TargetLink

Customization and Optimization Guide

For TargetLink 5.1

Release 2020-B – November 2020

dSPACE

How to Contact dSPACE

Mail:	dSPACE GmbH Rathenaustraße 26 33102 Paderborn Germany
Tel.:	+49 5251 1638-0
Fax:	+49 5251 16198-0
E-mail:	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.

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

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

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

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

Contents

About This Guide 15

Customizing the MATLAB Start-Up 19

How to Customize the MATLAB Start-Up 19

Customizing the TargetLink Environment 21

Modifying Environment Settings via the Preferences Editor 22
- Basics on Using the Preferences Editor 22
- How to Select a Code Editor 25
- How to Import and Export TargetLink Preferences via Command Line 26

Defining Search Paths for Customization Files, Demo Models,TSM Extension Packages and Instruction Set Simulators 28
- Defining Search Paths for Customization Files, Demo Models,TSM Extension Packages, and Instruction Set Simulators 28

Deriving Customization Files From Their Templates 33
- How to Define TargetLink's Search Path for DD-Related Customization Files 33
- How to Return TargetLink's Search Paths for DD-Related Customization Files 35
- How to Define TargetLink's Search Path for M-API-Related Customization Files 36
- How to Create Customization Files via the Create Customization Files Dialog 38
- How to Create Customization Files via the API 40

Target Simulation Module (TSM) Extension Packages 42
- How To Install TSM Extension Packages 42
- How to Clone Target/Compiler Combinations to Outside the TargetLink Installation 44

Specifying the Location of the Sources and Binaries of the TargetLink Fixed-Point Library 46
- Basics on Specifying the Location of the Sources and Binaries of the TargetLink Fixed-Point Library 46

Configuring the Code Generator for Production Code Generation 49

Basics on Configuring the Code Generator for Production Code Generation ... 49
Details on Preconfigured Code Generator Options (Objectives) ... 55
How to Overwrite Code Generator Options for Certain Objectives ... 58
Basics on Generating Code Compliant with C99/C11/C++ ... 61
How to Specify Custom Identifier Restrictions ... 67

Specifying the Location of Artifacts Generated or Used by TargetLink 71

Basics on Specifying the Location of Artifacts Generated by TargetLink ... 71
How to Specify the Location of Artifacts Generated by TargetLink ... 74

Adapting Code to Company Coding Styles 77

Generating Unique Names via Name Macros ... 78
Basics on Using Name Macros ... 78
Details on the Expansion of Name Macros ... 80

Working With Variable Classes ... 88

Customizing the Appearance of Variables via Variable Classes ... 88
Basics on Variable Classes ... 88
Details on Preconfigured VariableClass Objects in the Data Dictionary ... 90
How to Define New Variable Classes ... 93

Examples of Working with Variable Classes ... 95
Example of Making Block Outputs Observable ... 96
Example of Collecting Variables in Separate Modules ... 99
Example of Making Parameters Calibratable ... 101
Example of Changing Parameter Values on the ECU ... 104
Example of Including Externally Stored Parameter Declarations ... 109
Example of Changing TargetLink Subsystem Interfaces ... 112
Example of Specifying Block Parameters as Input Arguments of Functions ... 114
Example of Specifying Parameters as Preprocessor Macros ... 117
Example of Initializing Variables via Restart Functions ... 120
Example of Specifying Type Qualifiers via Type Prefixes ... 122

Example of Merging Variables..... 123
Example of Defining Variable Classes for Alias Variables..... 125

Working With Data Types..... 127

Customizing Data Types..... 127
Overview on Data Type Handling..... 127
Basics on TargetLink Base Types, Typedefs, and Header Files..... 128
How to Create New Type Definitions..... 134
How to Rename Base Types..... 136
How to Create User-Defined Data Types..... 139
Example of Controlling the Generation of the File Containing TargetLink Base Data Types..... 143

Working With Structures and Pointers to Structures..... 148

Arranging Data in Structured Variables..... 148
Basics on Arranging Data in Structured Variables..... 148
How to Explicitly Specify Structured Data Types..... 152
How to Scale the Leaf Struct Components of Array-Of-Struct Variables with Non-Standard Scaling..... 154
How to Create Explicitly Specified Structured Variables..... 155
How to Create Implicitly Specified Structured Variables..... 158
Example of Collecting Gain Parameters via Dynamic Components..... 161

Structuring Production Code via Pointers to Structs..... 163
Basics on Specifying Pointers to Structs..... 164
How to Create Pointers to Predefined Structs..... 167
How to Create Pointers to Predefined Structs Using Bus Port Blocks..... 170
How to Add Pointers to Structs to Function Signatures Manually..... 171
How to Create Pointers to Implicit Structs Using Bus Port Blocks..... 174
How to Specify Names for Structured Implicit Types..... 176

Customizing Production Code Generation via Hook Scripts..... 178
Basics on Using Hook Scripts..... 178

Decorating Production Code..... 183
Basics on Code Decorations..... 183
How to Specify Code Decorations in the Data Dictionary..... 186
Example of Mapping Variables to Memory Sections via Code Decorations..... 187
Example of Assigning Variables to Memory Sections When Using a Target Optimization Module..... 193

Customizing TargetLink's Code Formatting..... 197
Basics on Code Formatting..... 197

How to Customize the Code Formatting via Code Output Style Definition File..... 198
How to Select a Root Style Sheet..... 200

Controlling Objects and Properties Created by the Code Generator (Templates) 203

Basics on Using Templates..... 203
Details on Predefined DD Template Objects..... 209
How to Create DD Template Objects..... 213
Details on Variable Class Templates (VCTs)..... 214
Details on Scope Reduction and VCTs..... 217
Details on Function Class Templates..... 220
How to Specify VCTs Preserving Code Optimization Capabilities..... 222

Mapping of Subsystems and Functions 225

Structuring Large Models via Subsystems..... 226
Basics on Using Subsystems..... 226
How to Implement Subsystems as Functions..... 228

Customizing the Appearance of Functions via Subsystems and Function Classes..... 230
Basics on TargetLink Subsystems and Function Classes..... 230
How to Select Function Classes..... 233
How to Assign Function Classes to Implicit Functions..... 234
How to Specify Subsystem Interfaces..... 236

Defining the Function Interface..... 241
Basics on Function Interfaces..... 241
How to Specify Function Interfaces via Function Parameters..... 242

Centrally Specifying and Synchronizing Function System Signatures..... 245
Basics on Centrally Specifying Function System Signatures..... 245
Basics on Synchronizing Function System Signatures..... 252
How to Specify Function Block Data..... 258
How to Specify System Signature Data..... 260
How to Synchronize Function System Signatures..... 263

Encapsulating Access to Data Signals and Parameters via Access Functions..... 268

Introduction to Access Functions..... 268
Overview of Typical Use Cases for Access Functions..... 268
Basics on Access Functions..... 270

Implementing Access Functions ... 274
How to Create an Access Function Specification ... 275
Determining Access Function Kind According to the Use Case ... 276
Associating Access Functions With Variable Kinds ... 282
Changing Access Function Implementations Using Name Macros ... 286
Specifying Access Functions as C Functions ... 287
Specifying Access Functions as Preprocessor Macros ... 289
Mapping Access Functions to Modules ... 290
Controlling the Generation and Use of Local Copies (Access Functions) ... 292
Associating Access Functions with an API ... 295

Reusing C Functions and Variables for Identical Model Parts (Function Reuse) ... 297

Introduction to Function Reuse ... 297
Basics on Function Reuse ... 298
Basics on Reusing Variables of Preceding and Subsequent Blocks ... 303

Making Model Parts Reusable ... 306
How to Turn Reuseable Model Parts into Reusable System Definitions ... 307

Defining the Code Pattern of Reused Variables ... 308
How to Implement Instance-Specific Variables via Direct Reuse ... 308
How to Implement Instance-Specific Variables via Indirect Reuse ... 309
How to Implement Instance-Specific Variables as Function Arguments ... 311
How to Use Shared Variables ... 312

Specifying Different Initial Values for Instance-Specific Variables ... 313
How to Specify Instance-Specific Values for Library Subsystems ... 313
How to Specify Instance-Specific Values for Referenced Models ... 315

Managing Special Indirect Reuse Scenarios ... 316
How to Change the Default Variable Class of Main Reuse Structs ... 316
Example of Instance-Specific Variable Naming via MATLAB Scripts (Indirect Reuse) ... 317

Implementing Scaling-Invariant Functions ... 320
Basics on Scaling-Invariant Functions ... 320
Details on the Impact of Rescaling ... 322
Details on the Scaling Propagation Function ... 324
How to Specify Subsystems as Scaling-Invariant ... 329
How to Specify Stateflow Charts and Graphical Functions as Scaling-Invariant ... 331

Controlling Subsystem Execution ... 334
Basics on Controlling Subsystem Execution ... 334
Basics on Resettable Subsystems ... 335
Details on Implementing Externally Function-Call-Triggered Subsystems ... 336
Details on Implementing Function-Call-Triggered Subsystems ... 340
Details on Executing Enabled Subsystems ... 342
Details on Executing Triggered Subsystems ... 342
State Reset in Conditionally Executed Subsystems: Differences of Simulink and TargetLink ... 345

Controlling Function Inlining ... 350
Basics on Function Inlining ... 350
How to Inline Specific Functions During Code Generation ... 353
How to Flag a Function for Compiler Inlining ... 354
Example of Inlining a Function During Code Generation ... 355

Embedding External Code ... 358

Introduction ... 358
Overview of Methods for Embedding External Code ... 358

Embedding External Functions in a TargetLink Subsystem ... 360
How to Embed External Functions in a TargetLink Subsystem ... 360
Example of Bypassing Target-Specific Code for SIL Simulation When Embedding External Functions ... 365
Details on Controlling the Generation of Initialization, Termination and Restart Functions for External Step Functions by Using Function Classes ... 368

Embedding External Functions That Use Column-Major Order ... 369
Basics on Adapting Column-Major Functions for Use With TargetLink ... 369
How to Adapt a Column-Major Legacy Function Using C Code in a Custom Code Template ... 371
How to Adapt a Column-Major Legacy Function via Permute Dimensions Blocks in the Model ... 373

Including External Code via the Data Dictionary ... 376
Basics on Including External Code via the Data Dictionary ... 377
How to Build and Link External Modules via File Specifications ... 378
How to Include External Code Files via DD Modules ... 379

Decomposing Models for Distributed Development 381

Mapping Code Generation Units to Code (Module Ownership) ... 382
Principles of Partitioning Models and Code ... 382

Basics on Modules and Module Ownership..... 388
How to Create Module Specifications in the Data Dictionary..... 391
How to Specify a Model-Based CGU as the Specific Owner of a DD Module Object..... 393
How to Specify a DD-Based CGU as the Specific Owner of a DD Module Object..... 395

DD-Based Code Generation..... 397
Basics on Generating Code from the Data Dictionary..... 397
How to Create a DD CodeGenerationUnit Object for DD-Based Code Generation..... 400
Example of Specifying Calibratable Variables Used by Several Model-Based CGUs..... 401
Example of Specifying InterfaceVariables Used by Several Model-Based CGUs..... 403

Decomposing Models Logically (Incremental Code Generation)..... 406
Basics on Incremental Code Generation..... 406
How to Incrementally Generate Code for Function Subsystems..... 408
How to Extract Subsystems to a Test Frame in a Separate Model..... 411

Decomposing Models into Model Parts (Model Referencing)..... 413

Introduction to Model Referencing..... 413
Basics on Model Referencing..... 413
Details on Simulation and Logging of Referenced Models..... 416

Building Models with Model References from Scratch..... 419
How to Build a Referenceable Model from Scratch..... 420
How to Create Models with Model References..... 421

Building Models with Model References by Adapting and Converting Existing Models..... 422
How to Create Models with Model References by Adapting Existing Models..... 423

Generating Production Code in Combination with Model Referencing..... 424
Basics of Generating Production Code in Combination with Model Referencing..... 424
How to Generate Code in Combination with Model Referencing..... 426

Example of Adapting Existing Models for Model Referencing..... 430
Example Overview of Adapting Existing Models for Model Referencing..... 430
Example of Preparing a Subsystem for Conversion into a Referenced Model..... 431
Example of Defining DD Variable Objects for Mergeable Variables..... 432
Example of Defining Data Dictionary Module Objects..... 435

Example of Defining the Module Ownership of Globally Defined Variables..... 436
Example of Importing Code Generator Options into the Data Dictionary..... 439
Example of Creating a Model Reference from a Subsystem..... 440

Disabling and Restoring Model References..... 442
How to Disable Model References..... 442
How to Restore Model References..... 445

Defining Abstract Interfaces for Model-Components..... 447
Basics on Component-Based Development Using Abstract Interfaces (RDI Objects)..... 447
Basics on Modeling Flexible Parameters Using RDI Objects..... 455
How to Model Flexible Parameters Using RDI Objects..... 456
How to Map RDI Objects to Specific Implementation Variables..... 460
Basics on Modeling Efficient Interfaces..... 462
How to Model Efficient Interfaces Using RDI Objects..... 463

Customizing the Representation of Buses in Production Code 467

Mapping Buses and Subbuses to Implicit Structs in the Code..... 468
Basics on Implicit Structure Variables Resulting from Buses..... 468
How to Let TargetLink Create Implicit Structs for Buses..... 469
Examples of Implicit Structs for Nested Buses..... 471
Basics on Sort Sequences of Bus Elements in Implicit Structure Variables..... 475
How to Specify the Sort Criteria of Bus Elements..... 476

Mapping Entire Buses to Explicit Structs in the Code..... 479
Basics on the Compatibility of Buses and Predefined Structs..... 479
How to Map Entire Buses to Predefined Structure Variables..... 483
How to Map Entire Buses to Predefined Structure Type Definitions..... 485
Example of Mapping Bus Elements to a Custom C Struct..... 487

Specifying Bus Elements Individually..... 491
How to Create Separate Variables for Leaf Bus Elements..... 491
Details on Mapping Bus Elements to Function Arguments..... 492
Example of Leaf Bus Elements Specified Individually..... 493

Optimizing the Production Code 497

Introduction to Optimizing Production Code 498
Basics on Optimizing the Production Code 498
How to Enable Code Optimization 502

Optimizing Variables 504
Basics on Optimizing Variables 504
How to Optimize Variables 512
Basics on the Scope Reduction of Variables 513
How to Specify the Scope Reduction of Explicit Variables 515
Basics on Eliminating Temporary Variables 518
Basics on Optimizing Struct Variables 521

Optimizing Functions 524
Basics on Optimizing Functions via Property Values 524
Basics on Optimizing the Function Interface 528
How to Optimize Functions 531
Basics on the Scope Reduction of Functions 531
Basics on Merging External Functions 532

Optimizing Block-Specific Code 534
Basics on Optimizing Block Output Saturation 534

Optimizing Target-Specific Code 536
Basics on Target-Specific Optimization 536
Details on Code Optimization on Block and Block-Independent Level 537
Details on Look-Up Table Settings for Target-Optimized Code Generation 540

Optimizing Control Flows 543
Simplifying If And Switch Statements 543
Moving Code Into Conditional Branches 543
Merging Loops 545
Combining Code of If Statements 547

Optimizing via Range Propagation 551
Basics on Optimizing the Code via Range Propagation 551

Optimizing Bitfields for Booleans and Stateflow States 555
Basics on Collecting Booleans and Stateflow States in Global Bitfield Structures 555

Optimizing Code Efficiency via Memory-Specific Pointers 557

Basics on Memory-Specific Pointers..... 557
How to Enhance the AddressQualifiers XML object..... 559

Managing Variant Coding and Conditional Code Compilation 561

Introduction to Variant Coding and Conditional Code Compilation..... 562
Basics on Variant Coding and Conditional Code Compilation..... 562

Specifying Code Variants and Data Variants..... 564

Preparing the Data Dictionary for Code Variants and Data Variants..... 564
Basics on Variant Handling in the TargetLink Data Dictionary..... 564
How to Create Variant Items..... 568
How to Create Variant Configurations..... 570
How to Create Switch Variables for Data Variants..... 573

Specifying Code Variants..... 575
How to Specify Code Variants..... 575
How to Activate Code Variants..... 576

Specifying Data Variants..... 579
Basics on Coding Styles for Data Variants..... 579
How to Specify Data Variants..... 581
How to Activate Data Variants..... 583

Specifying Variable Vector Widths via Width Macros..... 586

Introduction to Variable Vector Widths..... 586
Basics on Variable Vector Widths..... 586
Details on Variable Vector Width Specification..... 589
Details on Variable Vector Width Implementation..... 592

Making Variables Width-Varying..... 594
How to Define Width Macros..... 596
How to Define the Width Values of Width-Varying Variables..... 598
How to Make Variables Width-Varying..... 600
How to Make Stateflow Variables Width-Varying..... 601
How to Terminate Width Propagation..... 603
How to Define Width Macros in Custom Header Files..... 604
How to Test Width-Varying Code..... 605
How to Generate a Report on the Propagation Failures..... 606

Examples of Width-Varying Variables..... 607
Example of Width-Varying Subsystems..... 607
Example of a Width-Varying Library..... 610
Example of a Width Macro Referenced in the Model..... 611
Example of Improved Code Readability due to Width Macros..... 614

Specifying Conditional Control Flows..... 616
Basics on Conditional Code Compilation via Preprocessor Directives..... 616
Details on Handling Objects in Preprocessor Directives..... 619
Details on Handling Logical Instructions in Preprocessor Directives..... 620
How to Specify Conditional Control Flows via Enabled Subsystems..... 621
How to Specify Conditional Control Flows via #If-#Else Statements..... 623
How to Include External Macros for Generating Conditional Control Flows..... 625

Glossary 629

Index 659

About This Guide

Contents

This guide presents the customization and optimization features of TargetLink. It outlines in detail, for example, how to adapt TargetLink settings so that the generated production code meets your specific requirements, for example, map models or model parts to code, minimize memory consumption and execution time, and reuse functions.

Orientation and Overview Guide For an introduction to the use cases and the TargetLink features that are related to them, refer to the 🕮 TargetLink Orientation and Overview Guide.

Required knowledge

This guide is most useful to software specialists. It is assumed that you know how to make models TargetLink-compliant, build simulation applications, and check the generated code. Knowledge in handling the host PC, the Microsoft Windows operating system and MATLAB is also presupposed.

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.

Customizing the MATLAB Start-Up

How to Customize the MATLAB Start-Up

Objective

To customize the MATLAB start-up, you must use custom scripts.

Execution order of the initialization phase

The execution order is:

- dSPACE initialization
- `dsstartup.m`
- `dspoststartup.m`

Method

To customize the MATLAB start-up

1. Add the necessary commands in the `dsstartup.m` file, for example, to automatically open a library or model whenever MATLAB is started. The `dsstartup.m` file has to be placed in the MATLAB search path.

   For more information, refer to dsstartup.m (📖 TargetLink File Reference).

   Tip

   To specify further commands to be executed after the dSPACE initialization, you can use `dspoststartup.m`.

Example

Suppose you want MATLAB to automatically change to your working folder `d:\work` and open the Simulink model `my_model.slx`. Create the dsstartup.m file and write the following commands to it:

```
cd d:\work
my_model
```

Customizing the MATLAB shutdown

You can customize the MATLAB shutdown by using the `dsfinish.m` file to carry out user-specific commands before MATLAB is shut down. Place this M file in the

MATLAB search path, for example, in MATLAB's working folder. You can specify several `dsfinish.m` files. All `dsfinish.m` files located in the MATLAB search path are executed before MATLAB is shut down.

Related topics

References

dsfinish.m (TargetLink File Reference)
dspoststartup.m (TargetLink File Reference)
dsstartup.m (TargetLink File Reference)

Customizing the TargetLink Environment

Where to go from here

Information in this section

Modifying Environment Settings via the Preferences Editor..... 22

Defining Search Paths for Customization Files, Demo Models,TSM Extension Packages and Instruction Set Simulators..... 28

Deriving Customization Files From Their Templates..... 33

Target Simulation Module (TSM) Extension Packages..... 42

Specifying the Location of the Sources and Binaries of the TargetLink Fixed-Point Library..... 46

Modifying Environment Settings via the Preferences Editor

Where to go from here

Information in this section

Basics on Using the Preferences Editor 22

How to Select a Code Editor 25

How to Import and Export TargetLink Preferences via Command Line 26

Basics on Using the Preferences Editor

Introduction

The Preferences Editor is a tool that lets you view and modify all user-specific preference settings after the installation was completed.

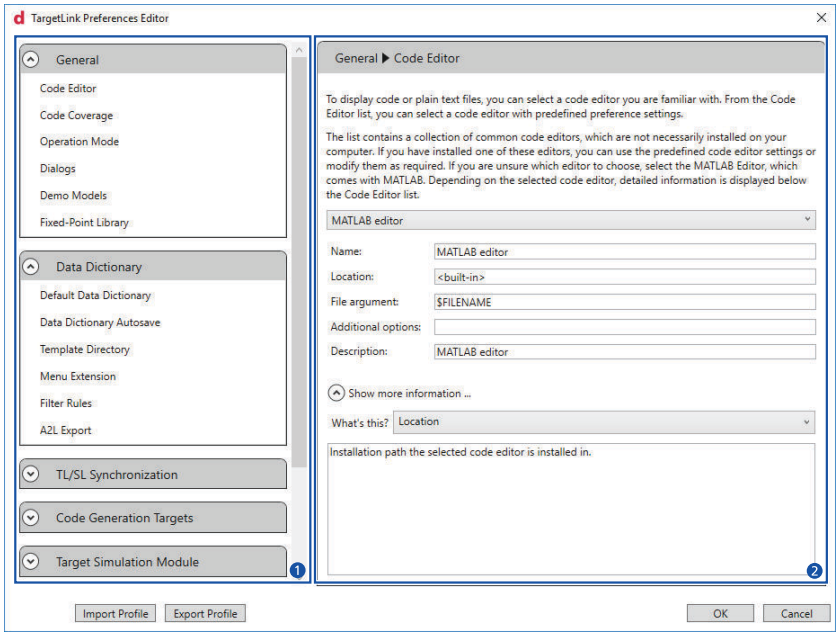
Customizing the TargetLink environment

Most of the steps for configuring the TargetLink environment are performed during the installation. A special installation program guides you through the installation and performs all of the necessary configuration settings. If required, you are prompted to confirm dialog messages or enter the location of associated programs, such as target-specific compilers.

You can access the Preferences Editor via the

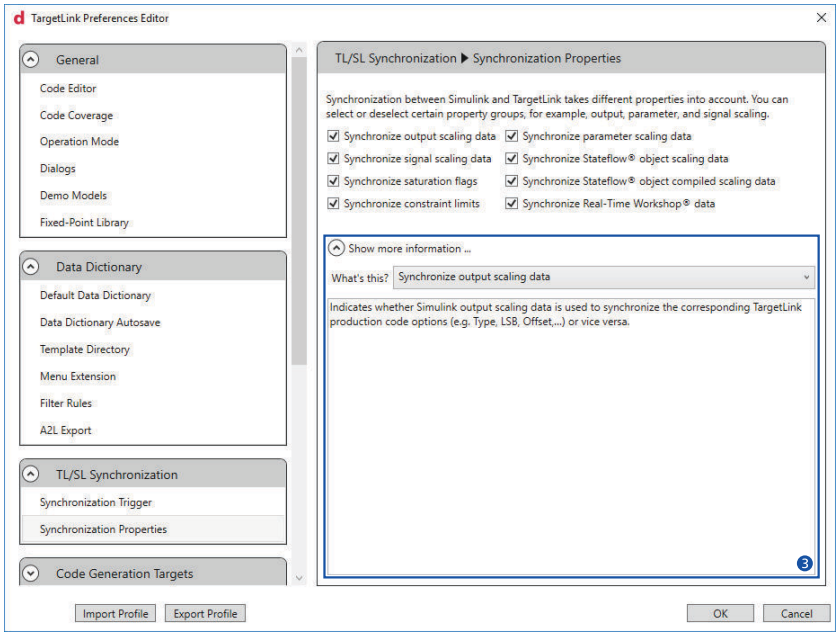
- **TargetLink Main Dialog**
- TargetLink menu in the Simulink Editor
- **TargetLink** shortcut in the Windows Start menu
- **`tl_pref`** API function.

When shipped, TargetLink comes with a set of default preferences. You can change these preferences according to your requirements via the TargetLink Preferences Editor. These preferences include the Code Editor to be used, the default Data Dictionary to be loaded, or the Code Generation Targets and the target simulation configurations that are to be displayed in the **TargetLink Main Dialog**, etc. The following illustration shows the panes of the TargetLink Preferences Editor, called Topic Navigator (1) and Preference Settings (2).



The Topic Navigator is divided into topic groups that can be expanded (collapsed) to display (hide) their topics. When you select a topic, all its preference settings are displayed in the Preference Settings pane. If required, you can change the preferences set during the installation, e.g., to customize your TargetLink environment. A general introduction to each topic is displayed at the top of the Preference Settings pane.

For some topics, the Preference Settings pane provides additional information for the currently selected control. Clicking the **Show more information** toggle button opens and closes the information area (3) for the currently selected control. You can also select an entry from the **What's this?** list to display help information.



Importing/exporting TargetLink configuration settings

You can import and export TargetLink configuration settings from and to a file. The whole set of TargetLink configuration settings is stored in multiple XML files:

- `TLPreferences.xml`
- `TLSerPort.xml`
- One `*.xml` file for each installed instruction set simulator (ISS). The xml file is named according to the following naming convention:

  `TLPIPEPort*.xml`.

  The placeholder will be replaced by the name of the ISS, e.g., `TLPIPEPortT32marm.xml` for the TRACE32 Simulator for ARM microcontrollers.

Importing and exporting TargetLink configuration settings is possible via:

- Preferences Editor (**Import Profile** and **Export Profile** buttons)
- DOS API (refer to How to Import and Export TargetLink Preferences via Command Line on page 26)

Related topics

Basics

Defining Search Paths for Customization Files, Demo Models,TSM Extension Packages, and Instruction Set Simulators..... 28

HowTos

How to Preselect the Default Base Data Type (📖 TargetLink Preparation and Simulation Guide)
How to Select a Code Editor 25

References

Preferences Editor (📖 TargetLink Tool and Utility Reference)
tl_pref (📖 TargetLink API Reference)
Topic Navigator (📖 TargetLink Tool and Utility Reference)
Topic Settings (📖 TargetLink Tool and Utility Reference)

How to Select a Code Editor

Objective

After production code was generated, you can inspect it with your preferred code editor. You can select an editor from the Code Editor list in the Preferences Editor. If you are unsure which editor to use, select the MATLAB editor which is distributed with MATLAB.

Method

To select a code editor

1. In the MATLAB Command Window, type `tl_pref` to open the Preferences Editor.
2. In the Preferences Editor, click General - Code Editor.
3. From the editor list, select an editor. By default, the MATLAB Editor is selected.
4. Click OK to confirm your changes and close the dialog.

Result

You selected a code editor.

Related topics

References

Common TargetLink Dialog Buttons (📖 TargetLink Model Element Reference)
Preferences Editor (📖 TargetLink Tool and Utility Reference)
tl_global_options (📖 TargetLink API Reference)

How to Import and Export TargetLink Preferences via Command Line

Objective

To modify the TargetLink preferences by an automatism, such as DOS API.

Precondition

`TlPreferences.exe` resides in `<TL_InstRoot>\PreferencesEditor\bin\`, which is the installation default.

Available command options

The `TlPreferences` command provides the following options:

Option	Description
`-Import:<path>`	Imports the TargetLink preferences from the folder specified by `<path>`.
`-Export:<path>`	Exports the TargetLink preferences to the folder specified by `<path>`.
`-Overwrite`	Allows overwriting existing files in the destination folder during export.
`-Log:<path>`	Creates a log file in the `<path>` folder. This option can be combined with any other option except `-Help`.
`-Reset`	Resets all the TargetLink preferences to the factory defaults.
`-Info`	Returns the user application settings path.
`-Help`	Lists the available command options.

Method

To import and export TargetLink preferences via command line

1. Open the Command Prompt window.
2. Switch to `<TL_InstRoot>\PreferencesEditor\Bin\`.
3. Enter the import or export command, for example:

   ```
   D:\TargetLink\PreferencesEditor\Bin>TlPreferences
   -Export:D:\global_tl_prefs -Overwrite
   ```

4. Press the Enter key.

   The command is executed. Existing files are overwritten. If the destination folder (export) does not exist yet, it is created.

Result

For the import, the settings stored in `TLPreferences.xml` and `TLSerPort.xml` are imported to the Preferences Editor.

Note

TargetLink automatically adjusts the paths for application data directories to the current user's user name.

For the export, the current settings, as shown in the Preferences Editor, are written to `TLPreferences.xml` and `TLSerPort.xml` in the destination folder.

Defining Search Paths for Customization Files, Demo Models,TSM Extension Packages and Instruction Set Simulators

Defining Search Paths for Customization Files, Demo Models,TSM Extension Packages, and Instruction Set Simulators

Introduction

You can separate user-specific or project-specific data from a particular TargetLink installation. To do so, you have to specify search paths.

You can define the search paths of the following items:

- Customization files
- Demo models
- TSM Extension Packages
- Instruction set simulators

This lets you configure TargetLink to work in an environment without administrator rights.

Customization files

TargetLink provides several template files which you can use to create customization files, such as hook scripts. This facilitates to separate project-specific data from a particular TargetLink installation by deriving project-specific customization files into user-specified folders. This also is useful if you are working in an environment without administrator rights.

Tip

To get a list of all available TargetLink customization file templates, use the `nameList = tlCustomizationFiles('GetList')` API function.

Creating customization files You can derive a customization file from its template as follows:

Customization File Template	Derive Customization File from Template Via ...	Refer To ...
DD templates	Data Dictionary Manager	How to Create DD Workspaces (📖 TargetLink Data Dictionary Basic Concepts Guide)
DD filter rules	`dsdd('CreateFilterRuleSet','...')`	How to Create Filter Rule Sets via DD Files (📖 TargetLink Data Dictionary Basic Concepts Guide)

Customization File Template	Derive Customization File from Template Via ...	Refer To ...
DD menu extensions A2L style sheets Hook scripts AUTODOC files[1)] Code output formatting[2)]	Customization Files dialog	How to Create Customization Files via the Create Customization Files Dialog on page 38
	`tlCustomizationFiles`	How to Create Customization Files via the API on page 40

1) e.g., `tldoc_pdf_customized`
2) e.g., `CodeOutputStyleDefinitionFile` and `CodeOutputStyleSheets`

Defining search paths You have to place the derived customization files in folders that reside on TargetLink's search path for customization files. You also have to add folders that contain M files on MATLAB's search path.

Tip

You can place TargetLink's customization files in different folders. This lets you place them as required by your project, for example, into into folders that are accessible without administrator rights. It is best to add the folders containing M files to the MATLAB path via a start script, which you can use to start a specific model.

You can define TargetLink's search paths as follows:

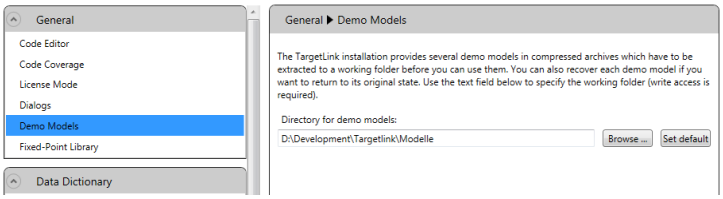
Customization File		Define TargetLink Search Path Via ...	Refer To ...
DD workspaces[1)]	`*.dd`	▪ Preferences Editor ▪ `tl_pref('Set', ...)`	How to Define TargetLink's Search Path for DD-Related Customization Files on page 33 If you do not specify a search path, MATLAB's current working directory is used by default.
DD filter rules	`*.xml`		
DD menu extensions	`*.xml`		
A2L style sheets	`*.xsl`		
Hook scripts AUTODOC files Libmap files	`*.m`[2)]	Edit `tl_get_config_path` (📖 TargetLink API Reference)	How to Define TargetLink's Search Path for M-API-Related Customization Files on page 36
Code output formatting[3)]	`*.xml` `*.xsl`	Refer to Code Formatting Customization Files (📖 TargetLink File Reference)	How to Customize the Code Formatting via Code Output Style Definition File on page 198 Basics on Code Formatting on page 197

1) DD templates become DD workspaces.
2) Must be placed in a folder whose *absolute* path is on the MATLAB and on the TargetLink search path.
3) `CodeOutputStyleDefinitionFile` and `CodeOutputStyleSheets`

Demo models

TargetLink lets you place user-specific copies of its demo models in a folder outside of the TargetLink installation folder. This allows you to use the demo models without having administrator rights.

Setting the demo model path It is best to set the path for the demo models in the Preferences Editor after completing the installation:



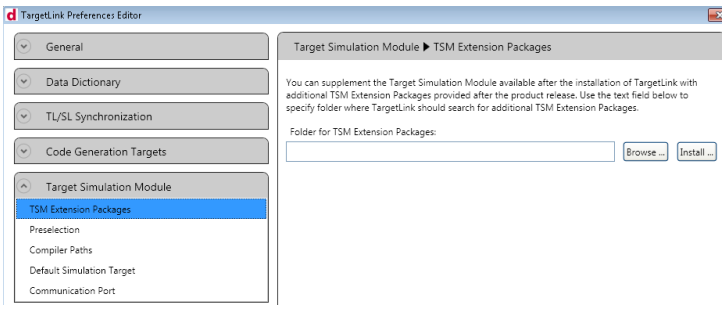
By default, TargetLink suggests the following folder: `<DocumentsFolder>\dSPACE\TargetLink\<Version>\Demos`.

If you do not specify the folder to the demo models within the Preferences Editor, TargetLink prompts you to specify it when you call `tl_demos` for the first time.

Restoring the demo models to their original state You can restore a demo model in the selected root folder by using the `tl_demos[<DemoName>] -restore` command. If you use the command without `<DemoName>`, all demo models are restored to their original state.

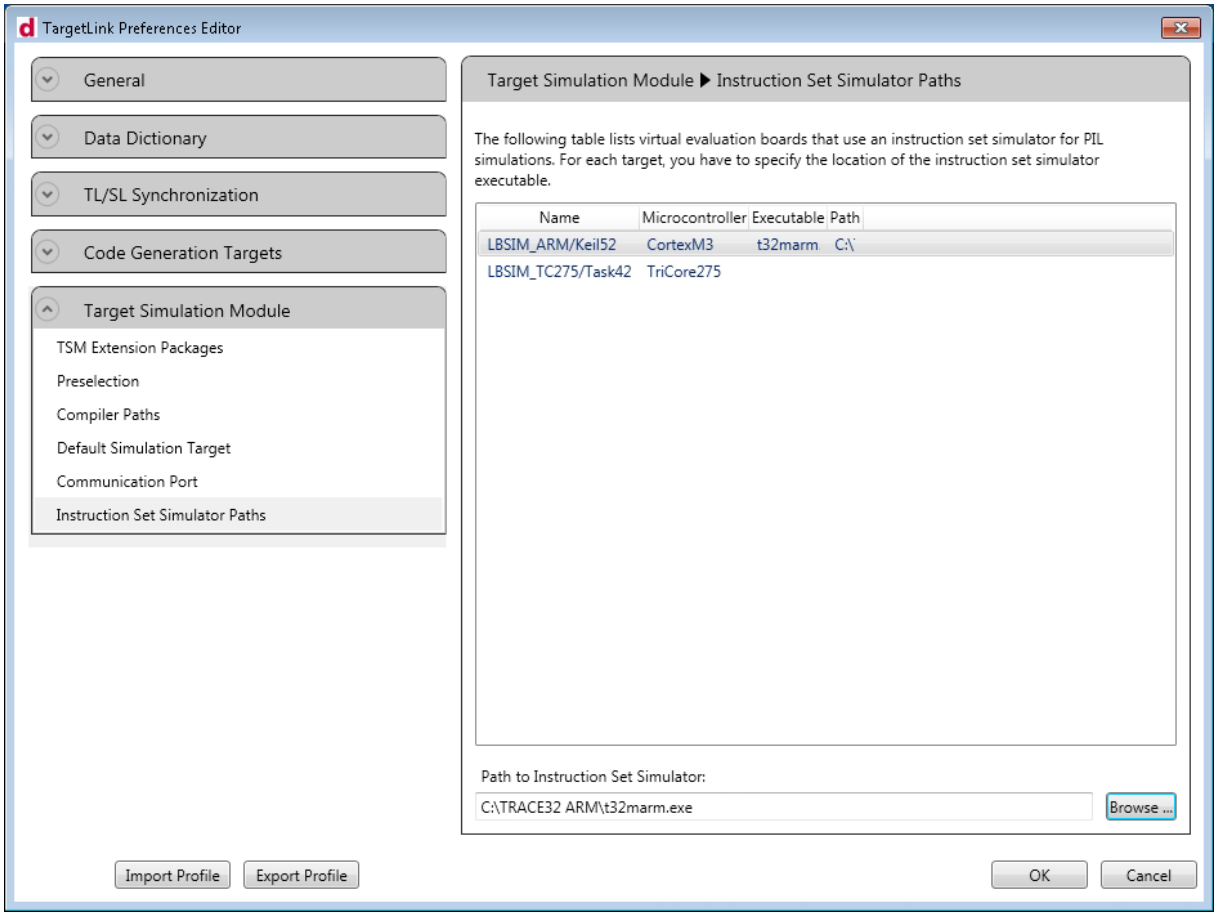
TSM Extension Packages

TargetLink lets you install Target Simulation Module (TSM) Extension Packages outside of the TargetLink installation folder. You can install TSM Extension Packages via the Preferences Editor, refer to How To Install TSM Extension Packages on page 42.



Instruction set simulators

TargetLink lets you perform PIL simulations on instruction set simulators (ISS). These can be installed outside the TargetLink installation folder, for example if you are working in an environment without adminstrator rights. For each available ISS, you can specify the installation path via the TargetLink Preferences Editor. This is necessary for the instruction set simulators you intend to use.



Cloning TSMs

TargetLink lets you clone target/compiler combinations. All the target/compiler files are then stored to the same folder as the TSM Extension Packages. You can clone TSMs by using the `TsmManagerUtility` command, which must be entered in a Command Prompt window. Refer to How to Clone Target/Compiler Combinations to Outside the TargetLink Installation on page 44.

Running multiple instances of TargetLink

You can run multiple instances of MATLAB independently, so multiple instances of TargetLink can run in parallel, but only one TargetLink instance per instance of MATLAB. This lets you e.g. compare the generated production code of different MATLAB/TargetLink installations simultaneously. You can define which installed TargetLink version is the preferred one during setup or via the Installation Manager.

Related topics

Basics

Basics on Code Formatting..... 197
Basics on Replacing Simulink Blocks via Libmaps (📖 TargetLink Preparation and Simulation Guide)
Create Customization Files Dialog (📖 TargetLink Tool and Utility Reference)

HowTos

How to Create Customization Files via the Create Customization Files Dialog..... 38
How to Create DD Workspaces (📖 TargetLink Data Dictionary Basic Concepts Guide)
How to Create Filter Rule Sets via DD Files (📖 TargetLink Data Dictionary Basic Concepts Guide)
How to Customize the Code Formatting via Code Output Style Definition File..... 198
How to Define TargetLink's Search Path for DD-Related Customization Files 33
How to Define TargetLink's Search Path for M-API-Related Customization Files..... 36
How To Install TSM Extension Packages..... 42

References

Code Formatting Customization Files (📖 TargetLink File Reference)
Preferences Editor (📖 TargetLink Tool and Utility Reference)
tl_demos (📖 TargetLink API Reference)
tl_pref (📖 TargetLink API Reference)
tlCustomizationFiles (📖 TargetLink API Reference)

Deriving Customization Files From Their Templates

Where to go from here

Information in this section

How to Define TargetLink's Search Path for DD-Related Customization Files 33

How to Return TargetLink's Search Paths for DD-Related Customization Files..... 35

How to Define TargetLink's Search Path for M-API-Related Customization Files..... 36

How to Create Customization Files via the Create Customization Files Dialog..... 38

How to Create Customization Files via the API..... 40

How to Define TargetLink's Search Path for DD-Related Customization Files

Objective

To store TargetLink's DD-related customization files to folders other than the current MATLAB working directory. TargetLink can then find and evaluate these files. This is especially helpful if you work with TargetLink's customization files without having administrator rights, because you can use storage folders outside the TargetLink installation.

DD-related customization files

The DD-related customization files for which you can specify a folder in the Preferences Editor are grouped as follows:

- DD workspaces
- DD filter rules
- DD menu extensions
- A2L style sheets

Possible methods

You can define the search paths as follows:

- Via Preferences Editor, refer to Method 1.
- Via the API, refer to Method 2.

Method 1

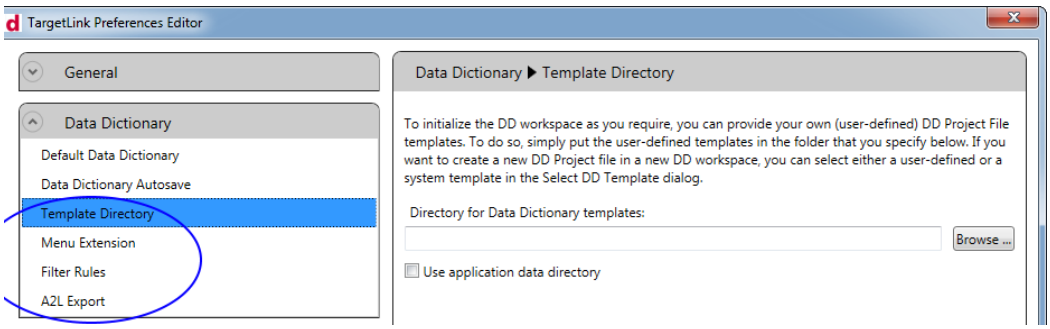
To define TargetLink's search path for DD-related customization files via the Preferences Editor

1 Open the Preferences Editor. You can do so via :
   - The `tl_pref` command in the MATLAB Command Window
   - The Preferences button on the TargetLink Main Dialog's Option page
   - The Windows Start menu:

     Start – All Programs – dSPACE TargetLink – TargetLink Preferences Editor

2 On the left in the Preferences Editor dialog, open the Data Dictionary tab.

3 Select one of the customization file groups marked in the following illustration:



4 On the right, browse for the folder, or select the Use application data directory checkbox.

> **Tip**
>
> TargetLink adjusts the paths for application data directories automatically to the current user's user name.

5 Repeat steps 3-4 for each customization file group you want to use.

6 Click OK.

Method 2

To define TargetLink's search path for DD-related customization files via the API

1 In the MATLAB Command Window, type the appropriate command:

Customization File Group	Command
DD templates	`tl_pref('set','DDTemplateDirectory', '<SearchPath>')`
DD filter rules	`tl_pref('set','DDFilterRuleDirectory', '<SearchPath>')`
DD menu extensions	`tl_pref('set','DDMenuExtensionDirectory', '<SearchPath>')`
A2L style sheets	`tl_pref('set','A2LStyleSheetDirectory', '<SearchPath>')`

Note

If you enter `%APPDATA%` as the `<SearchPath>`, the application data folder is used.

2 Press the Enter key.

Result

You added one or more folders that are to contain TargetLink's DD-related customization files to TargetLink's search path for customization files.

When you create a TargetLink customization file (refer to How to Create Customization Files via the Create Customization Files Dialog on page 38), this file is stored in the folder that relates to the specific customization file group.

Tip

To list all the folders that relate to the customization file groups, enter `prefStruct = tl_pref('get')` in the MATLAB Command Window and refer to the corresponding four entries of `prefStruct`.

Related topics

Basics

Defining Search Paths for Customization Files, Demo Models,TSM Extension Packages, and Instruction Set Simulators..... 28

HowTos

How to Create Customization Files via the Create Customization Files Dialog..... 38
How to Define TargetLink's Search Path for M-API-Related Customization Files..... 36

How to Return TargetLink's Search Paths for DD-Related Customization Files

Objective

To return the search path of a certain customization file group in the MATLAB Command Window so that it can be copied to the Clipboard. This is especially helpful if the folder is an application data folder, which is not easy to find in the file system.

DD-related customization files

The following DD-related TargetLink customization files require a TargetLink search path:

- DD workspaces
- DD filter rules

- DD menu extensions
- A2L style sheets

Method

To return TargetLink's search paths for customization files in the MATLAB Command Window

1 In the MATLAB Command Window, type the appropriate command:

Customization File Group	Command
DD templates	`tl_pref('get','DDTemplateDirectory')`
DD filter rules	`tl_pref('get','DDFilterRuleDirectory')`
DD menu extensions	`tl_pref('get','DDMenuExtensionDirectory')`
A2L style sheets	`tl_pref('get','A2LStyleSheetDirectory')`

2 Press the Enter key.

Result

The search path is displayed in the MATLAB Command Window. For example:

```
tl_pref('set', 'A2LStyleSheetDirectory', '%APPDATA%')
myA2LDir = tl_pref('get', 'A2LStyleSheetDirectory')

myA2LDir =

C:\Users\<user>\AppData\Local\dSPACE\<GUID>\TargetLink\A2LStyleSheets
```

Related topics

Basics

Defining Search Paths for Customization Files, Demo Models,TSM Extension Packages, and Instruction Set Simulators..... 28

HowTos

How to Define TargetLink's Search Path for M-API-Related Customization Files..... 36

How to Define TargetLink's Search Path for M-API-Related Customization Files

Objective

To work with TargetLink's M-API-related customization files like hook scripts, you have to derive them from their templates and place them in a folder that is on MATLAB's *and* TargetLink's search path.

Note

All the directories on TargetLink's search path must be on MATLAB's search path.
All paths must be defined as *absolute* paths. Do not use `..` in conjunction with MATLAB's `fullfile` and `addpath` API.

Precondition

The following preconditions must be fulfilled:

- You created one or more <CustomizationFileFolder> folders to contain TargetLink's M-API-related customization files as required by your project.
- You added the *absolute* path to the <CustomizationFileFolder> folders to MATLAB's search path, e.g., in your `start.m` script.

Method

To define TargetLink's search path for M-API-related customization files

1. In the MATLAB Command Window, enter the following command:

```
tlCustomizationFiles('Create','tl_get_config_path','<Customization
FileFolder>')
```

   TargetLink performs the following steps:

   1. Copies the `tl_get_config_path.m` to the specified folder
   2. Opens the `tl_get_config_path.m` file in the editor

2. In the `tl_get_config_path.m`, file add the *absolute* path to the <CustomizationFileFolder> folder(s). For example:

```
…
    'D:\user\MyPreCodeGenHooks'
    fullfile('D:\user', 'MyPostCodeGenHooks')
…
```

3. Save and close the `tl_get_config_path.m` file.

Result

You added one or more *absolute* paths of <CustomizationFileFolder> folders to TargetLink's search path. These folders can now contain TargetLink's M-API-related customization files.

Next step

You can derive M-API-related customization files from their templates and place them in the <CustomizationFileFolder> folders.

For instructions, refer to How to Create Customization Files via the Create Customization Files Dialog on page 38.

Related topics

Basics

Basics on Using Hook Scripts..... 178
Defining Search Paths for Customization Files, Demo Models,TSM Extension Packages, and Instruction Set Simulators..... 28

HowTos

How to Create Customization Files via the Create Customization Files Dialog..... 38
How to Define TargetLink's Search Path for DD-Related Customization Files 33

References

tl_get_config_path (📖 TargetLink API Reference)

How to Create Customization Files via the Create Customization Files Dialog

Objective

TargetLink provides several customization file templates that you can copy and convert to functional customization files.

Restriction

The Create Customization Files dialog can place customization files only in their default directory. Refer to Defining Search Paths for Customization Files, Demo Models,TSM Extension Packages, and Instruction Set Simulators on page 28.

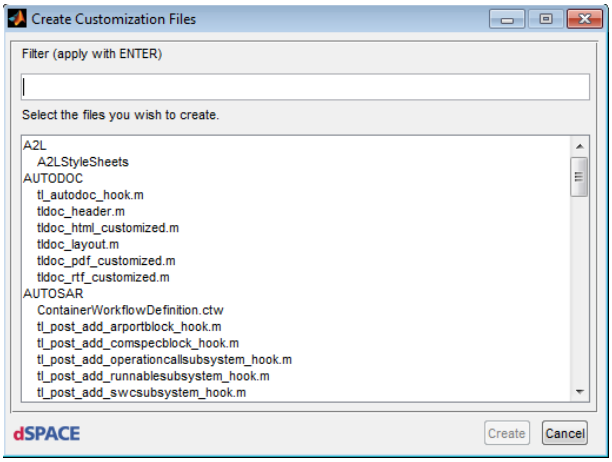
Method

To create customization files via the Create Customization Files dialog

1. In the MATLAB Command Window, type the following command (case-sensitive):

   ```
   tlCustomizationFiles
   ```

   The Create Customization Files dialog opens and lists all the available template files. You can apply a filter to the listed files only, not to the categories.



2 Select one or more template files and click **Create**.

TargetLink performs the following steps:

1. Copies the customization files to their default folder.

   Note

   The files are saved either to the folder specified in the Preferences Editor or, if no folder is specified, to the current MATLAB working directory.

2. Makes the customization files functional.
3. Opens the customization files in the editor.

3 Add the required functionality to the files.

4 Save and close the files.

Result

You created one or more customization files in their default folders and added the required functionality.

Related topics

Basics

Basics on Code Formatting..... 197
Defining Search Paths for Customization Files, Demo Models,TSM Extension Packages, and Instruction Set Simulators..... 28

HowTos

How to Customize the Code Formatting via Code Output Style Definition File..... 198
How to Define TargetLink's Search Path for DD-Related Customization Files 33
How to Define TargetLink's Search Path for M-API-Related Customization Files..... 36

References

Create Customization Files Dialog Description (📖 TargetLink Tool and Utility Reference)
tlCustomizationFiles (📖 TargetLink API Reference)

How to Create Customization Files via the API

Objective

TargetLink provides several customization file templates that you can copy and convert to functional customization files. You can place these customization files in any folder that is on MATLAB's and on TargetLink's search path.

Precondition

You specified the search path for the customization files. Refer to Defining Search Paths for Customization Files, Demo Models,TSM Extension Packages, and Instruction Set Simulators on page 28.

Method

To create customization files via the API

1 In the MATLAB Command Window type the following command:

```
tlCustomizationFiles('Create','<CustomizationFileName>',...
'<CustomizationFileFolder>')
```

TargetLink performs the following steps:

1. Copies the `<CustomizationFileName>` customization file template to the `<CustomizationFileFolder>` folder.
2. Makes the customization file functional.
3. Opens the customization file in the editor.

2 Add the required functionality to the file.

3 Save and close the file.

4 Repeat steps 1-3 for each customization file you want to use.

Tip

If you want to place several customization files in the same folder, specify <CustomizationFileName> as a cell array that contains the names of each customization file.

Result

You created one or more customization files in a custom folder and added the required functionality.

Related topics

Basics

Basics on Using Hook Scripts..... 178
Defining Search Paths for Customization Files, Demo Models,TSM Extension Packages, and Instruction Set Simulators..... 28

HowTos

How to Define TargetLink's Search Path for DD-Related Customization Files 33

Target Simulation Module (TSM) Extension Packages

Where to go from here

Information in this section

How To Install TSM Extension Packages..... 42

How to Clone Target/Compiler Combinations to Outside the TargetLink Installation..... 44

How To Install TSM Extension Packages

TSM Extensions

TSM Extensions extend the Target Simulation Module (TSM) with additional target/compiler combinations. All Target Simulation Module Extensions will be delivered as TSM Extension Packages. TSM Extension packages consist of a TSM container file with the file format `*.tsmz`.

Preconditions

The TSM container files you want to install must be available.

Workflow

The workflow consists of the following parts:

- Installing the TSM Extension packages, refer to Part 1 on page 42.
- Making the target/compiler combination available for code generation and PIL simulation, refer to Part 2 on page 43.

Part 1

To install TSM Extension Packages

1. Copy the TSM container file into a temporary directory.
2. Open the TargetLink Preferences Editor.
3. In the Topic Navigator pane, click **Target Simulation Module** and then **TSM Extension Packages**.
4. In the Preference Settings pane, next to the **Folder for TSM Extension Packages** edit field, click the **Browse...** button.

5 In the dialog that opens, create or select the folder in which you want to install the TSM Extension Packages.

Note

You must have write access for this folder. Therefore, you must choose a location for this folder outside the TargetLink installation.

Click OK to close the dialog.

6 In the Preference Settings pane, click the Install... button.

In the dialog that opens, select the TSM container file and click Open to start the installation.

Interim result

You installed the TSM Extension Packages.

Part 2

To make the target/compiler combination available for code generation and PIL simulation

1 In the Topic Navigator pane of the Preferences Editor, click Code Generation Targets and then Preselection.

2 In the Preference Settings pane, select the Code Generation Target for the TSM Extension Package you previously installed.

3 In the Topic Navigator pane, click Target Simulation Module and then Preselection.

4 In the Preference Settings pane, select the target/compiler combination.

5 If required, you can set the compiler path.

In the Topic Navigator pane, click Compiler Paths. In the Preference Settings pane, select the target/compiler combination and change the installation folder via the Installation folder: edit field or the Browse... button.

6 Depending on the specific target/compiler combination, it might be necessary to change the COM port or baud rate.

In the Topic Navigator pane, select Communication Port. In the Preference Settings pane, select the COM port required for the target/compiler combination from the Communication port list.

7 Click OK to close the TargetLink Preferences Editor.

Result

You installed TSM Extension Packages with additional target/compiler combinations and made them available for code generation and PIL simulation.

Related topics

Basics

Basics on Using the Preferences Editor..... 22
Defining Search Paths for Customization Files, Demo Models,TSM Extension Packages, and Instruction Set Simulators..... 28

HowTos

How to Clone Target/Compiler Combinations to Outside the TargetLink Installation..... 44

References

Combinations of Evaluation Boards and Compilers (📖 Evaluation Board Reference)

How to Clone Target/Compiler Combinations to Outside the TargetLink Installation

Objective

To modify TSMs without modifying the TargetLink installation.

Precondition

A TSM Extension Packages folder is specified. You can specify the folder via the API or via the Preferences Editor.

When shipped, `TlTsmManager.exe` resides in the following directory:

`<TL_InstRoot>\TsmInstallationManager\bin\`

Available command options

The `TlTsmManager` utility provides the following options:

Option	Description
`-Version`	Displays the version of the `TlTsmManager.exe` utility.
`-FrameworkVersion`	Displays the FrameworkVersion. This is the current format version of the TSM Extension Package (TSMZ file).
`-MinFrameworkVersion`	Displays the oldest possible FrameworkVersion. TSM Extension Packages that do not meet this minimum version cannot be processed.
`-Install`	Installs a specific target/compiler combination.
`-SourceFile:<path>`	Path of the TSM Extension Package (TSMZ file)
`-Uninstall`	Uninstalls a specific target/compiler combination.
`-Board:<name>`	Evaluation board
`-Compiler:<name>`	Compiler

Option	Description
`-Clone`	Clones a specific target/compiler combination. The TSM files are copied and pasted into the TSM Extension Packages folder specified in the Preferences Editor.
`-Board:<name>`	Evaluation board
`-Compiler:<name>`	Compiler
`-NewCompiler:<name>`	New compiler name (if you want to rename the compiler)
`-Location:Tl|Ext`	Tl: Processes only TSM files that are stored inside the TargetLink installation. Ext: Processes only TSM files that are stored in the TSM Extension Packages folder.
`-CreateContainer`	Creates a TSM Extension Package (TSMZ file).
`-Board:<name>`	Evaluation board
`-Compiler:<name>`	Compiler
`-TargetDir:<path>`	Target directory for the new TSM Extension Package (TSMZ file)
`-Log:<path>`	Creates a log file in the `<path>` folder. This option can be combined with any other option except `-Help`.
`-Help`	Lists the available command options.

Method

To clone a target/compiler combination

1. Open the Command Prompt window.
2. Switch to `<TL_InstRoot>\TsmInstallationManager\bin\`.
3. Enter the **`-Clone`** command. For example:

   ```
   <TL_InstRoot>\TsmInstallationManager\bin\TlTsmManager -Clone
   -Board:MPC5561EVB -Compiler:GHS2013 -NewCompiler:MyGHS2013 -
   Location:Tl
   ```

4. Press the Enter key.

Result

You have cloned the target/compiler combination. The target/compiler files are stored to the TSM Extension Packages folder (refer to Precondition on page 44).

Related topics

References

Preferences Editor (📖 TargetLink Tool and Utility Reference)

Specifying the Location of the Sources and Binaries of the TargetLink Fixed-Point Library

Basics on Specifying the Location of the Sources and Binaries of the TargetLink Fixed-Point Library

Introduction

TargetLink production code can contain macros and functions that belong to the Fixed-Point Library.

By default, TargetLink internally manages the macros and functions provided by the Fixed-Point Library.

However, you can specify where to store the corresponding sources and binaries, for example, to modify and manage them in an external version control system.

Tip

If you simply want to obtain a copy of the Fixed-Point Library files that are used in your production code to be able to build the production code outside of TargetLink, you can export them. Refer to How to Export Generated Files from the TargetLink Environment (TargetLink Interoperation and Exchange Guide)

Specifying the location of the sources

You can specify the location of the sources via the Preferences Editor or via the API:

Method	Description
Preferences Editor	You can specify the location in the **Fixed-Point Library** topic contained in the **General** topic group of the **Preferences Editor**.
API	You can specify the location via the following API function: `tl_pref('set', 'FixedPointLibrarySourcesDirectory', '<SomePath>')`

Note

After you specified the location of the sources, make sure to copy the files to this location via the **Copy** button in the **Fixed-Point Library** topic, which is contained in the **General** topic group of the **Preferences Editor**.

Directory structure The sources of the Fixed-Point Library are placed in different subdirectories, depending on whether they are generic or target-specific.

File Type	Location
Generic files	The generic versions are stored in `\Generic` and below.
Target-specific files	The target-specific versions are stored in `\<MicrocontrollerFamily>\<CompilerFamily>` and below. There are two kinds of target-specific header files: ▪ Header files with target-specific macro definitions and declarations of functions containing assembler instructions, named `dsfxp_a.h`. ▪ Header files with target-specific, ANSI C-compatible macro definitions and ANSI C function declarations, named `dsfxp_c.h`

Note

- Target-specific macros and functions that belong to the TargetLink Target Optimization Modules (TOMs) are non-ANSI C and can contain assembler instructions. If you are not allowed to use assembler instructions in code, you have to configure the TargetLink Code Generator accordingly. For more information, refer to the Code Generation page of the TargetLink Main Dialog Block.
- The target-specific libraries also contain all the generic elements. This lets you link against a single file.

The comments contained in the header and source files provide information that explains the macros and functions:

```
/*****************************************************************************
*
* FILE:
*   fit.h
*
* DESCRIPTION:
*   This file contains macro definitions for fitting an input value into an
*   output type.
*
*   The following applies to all macros:
*   ------------------------------------
*
*   The macro name represents the implemented operation:
*    C__<output_type>FIT<input_type>_SAT[(l|u)]
*
*    output_type            : U8, U16, U32,      I8, I16, I32,
*     input_type            : U8, U16, U32, U64, I8, I16, I32, I64
*
*
*   The parameters are classified with three keywords.
*
*    Parameter  Description                    Properties
*    ---------------------------------------------------------------------------
*    v          input                          rVars,  NoConsts, NoComplexExpr
*    v_h        upper part of a 64-bit input   rVars,  NoConsts, NoComplexExpr
*    v_l        lower part of a 64-bit input   rVars,  NoConsts, NoComplexExpr
*    csatval    saturation limit (positive),
*               the negative value is -csatval rVars,  Consts,   NoComplexExpr
*
*
```

Modifying the sources

If you are an expert user, you can modify the sources of the TargetLink Fixed-Point Library, once you copied its sources to the specified directory.

Changes you make to the sources of the Fixed-Point Library take effect after you build the library from the modified sources. This happens automatically during

the build process for SIL and PIL simulations, based on the dependencies of the simulation applications.

Building the library independently of the build process

If you want to build the sources of the Fixed-Point Library independently from the build process, you can use the `tlRebuildFixedPointLibrary` API function.

This can be useful in the following cases:

- To quickly verify that the sources still can be compiled.
- To build the sources using another compiler version or different compiler options.

Specifying the location of the binaries

You can specify the location of the binaries via the Preferences Editor or via the API:

Method	Description
Preferences Editor	You can specify the location in the **Fixed-Point Library** topic contained in the **General** topic group of the **Preferences Editor**.
API	You can specify the location via the following API function: `tl_pref('set', 'FixedPointLibraryBinariesDirectory', '<SomePath>')`

Related topics

Basics

Tracing TargetLink Fixed-Point Library Macros and Functions in Production Code (📖 TargetLink Preparation and Simulation Guide)

References

Fixed-Point Library Files (📖 TargetLink File Reference)
Preferences Editor (📖 TargetLink Tool and Utility Reference)
TargetLink Main Dialog Block (📖 TargetLink Model Element Reference)
tl_pref (📖 TargetLink API Reference)
tlRebuildFixedPointLibrary (📖 TargetLink API Reference)

Configuring the Code Generator for Production Code Generation

Introduction

To configure Code Generator options for producing C code for a code generation unit (CGU).

Where to go from here

Information in this section

Basics on Configuring the Code Generator for Production Code Generation 49

Details on Preconfigured Code Generator Options (Objectives) 55

How to Overwrite Code Generator Options for Certain Objectives 58

Basics on Generating Code Compliant with C99/C11/C++ 61

How to Specify Custom Identifier Restrictions 67

Basics on Configuring the Code Generator for Production Code Generation

Code Generator options

You can configure the Code Generator for production code generation by using Code Generator options.

Code generation units

The smallest unit for which code generation is possible is called code generation unit (CGU).

Code generation units can be model-based or DD-based:

Model-Based CGU	DD-Based CGU
TargetLink supports the following model-based CGUs: ▪ TargetLink subsystems ▪ Subsystems configured for incremental code generation ▪ Referenced models	TargetLink supports the following DD-based CGUs: ▪ DD CodeGenerationUnit objects

For information on working with different CGUs, refer to Decomposing Models for Distributed Development on page 381.

Storing Code Generator options

TargetLink can store Code Generator options at two locations that are independent of each other:

Model	Data Dictionary
At the root model.	In one or more DD CodegenOptionSet objects in the `/Pool/CodegenOptionSets/` subtree.
Relevant for *all* model-based CGUs contained in the *same* model hierarchy.	Relevant for the following kind of CGUs: ▪ DD-based CGUs that reference the CodegenOptionSet object via their CodegenerationOptionSetRef property ▪ Model-based CGUs in the model hierarchy of the root model whose TargetLink Main Dialog block references the CodegenOptionSet object via its Code Generator option set edit field.

For a complete overview of the Code Generator options, refer to Code Generator Options Dialog (📖 TargetLink Tool and Utility Reference).

Editing Code Generator options

You can edit Code Generator options as shown in the following table:

Code Generator Options	Access
Stored at the root model.	You can edit Code Generator options that are stored at the root model as follows: ▪ Via the All Options button on the Advanced page of the TargetLink Main Dialog Block ▪ Via the relevant controls on the Code Generation page and the Advanced page of the TargetLink Main Dialog Block Referencing a DD CodegenOptionSet object at the TargetLink Main Dialog block sets all the controls related to the Code Generator options as specified in the DD and inactivates the controls in the block dialog.
Stored in the Data Dictionary.	You can edit Code Generator options that are stored in the Data Dictionary as follows: ▪ By double-clicking a DD CodegenOptionSet object ▪ By selecting the **Edit Code Generator Options** (📖 TargetLink Data Dictionary Manager Reference) context-menu command

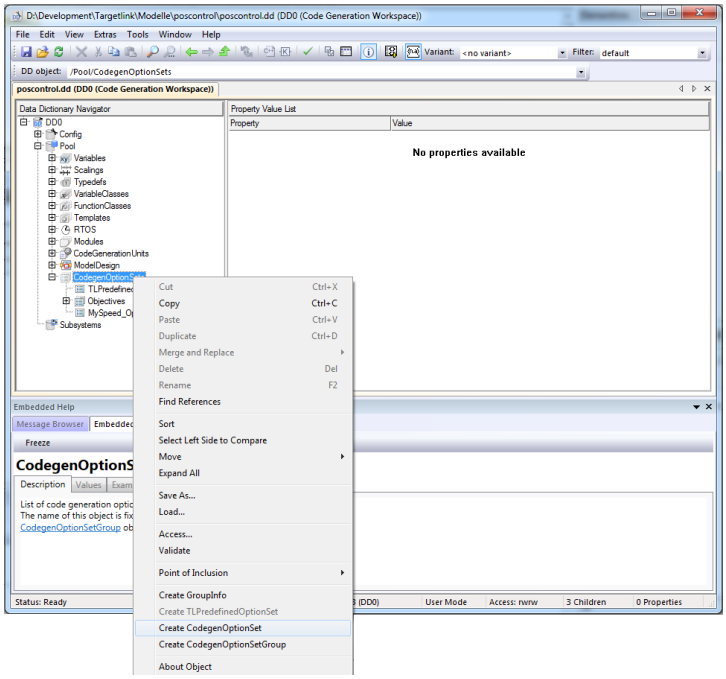
Hook scripts You can also change Code Generator options pragmatically via hook scripts that are executed prior to code generation.

For more information, refer to Customizing Production Code Generation via Hook Scripts on page 178 and Code Generation Hook Scripts (📖 TargetLink File Reference).

Defining DD Code Generator options

In the TargetLink Data Dictionary, you can define several CodegenOptionSet and CodegenOptionSetGroup objects. A CodegenOptionSet provides specific Code Generator options that are relevant for code generation. DD CodegenOptionSet objects can be grouped in DD CodegenOptionSetGroup objects and are located in `/Pool/CodeGenOptionSets`. In addition to your own DD CodegenOptionSet objects, TargetLink provides further CodegenOptionSet objects: the DD TLPredefinedOptionSet object and a CodegenOptionSetGroup object named Objectives.

Creating CodegenOptionSet objects You can create your own DD CodegenOptionSet objects in the TargetLink Data Dictionary.



Note

You are strongly recommended to configure DD CodegenOptionSet objects via the Code Generator Options Dialog dialog to avoid specifying invalid values.

If you close the Code Generator Options Dialog dialog by clicking the OK button, invalid values are reset to their defaults and deleted Code Generator options are restored.

Using the TLPredefinedOptionSet This option set contains general-purpose Code Generator options, i.e., options that are suitable for code generation in most cases. Keep in mind that its default values might change between Releases.

Using preconfigured Objectives If you create a new DD workspace via the dsdd_master_advanced.dd [System], the dsdd_master_autosar3.dd [System] or the dsdd_master_autosar4.dd [System] system templates, TargetLink provides the CodegenOptionSetGroup object named Objectives. It contains a number of predefined CodegenOptionSet objects, i.e., preconfigured Code Generator options optimized for a certain objective, e.g., MISRA compatibility, code readability, or resource consumption optimization. For more information, refer to Details on Preconfigured Code Generator Options (Objectives) on page 55.

Resetting TLPredefinedOptionSet to its default

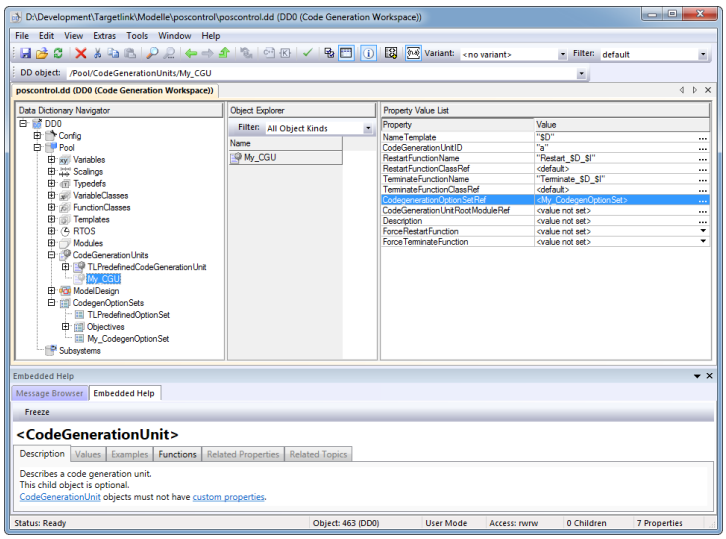
You can reset the DD `/Pool/CodegenOptionSets/TLPredefinedOptionSet` object via its Reset context menu command (works only for the TLPredefinedOptionSet object) or via the API:

```
dsdd('ResetToDefault','/Pool/CodegenOptionSets/<CodegenOptionSet>')
```

To reset other DD CodegenOptionSet objects, you can use only the API.

DD-based code generation

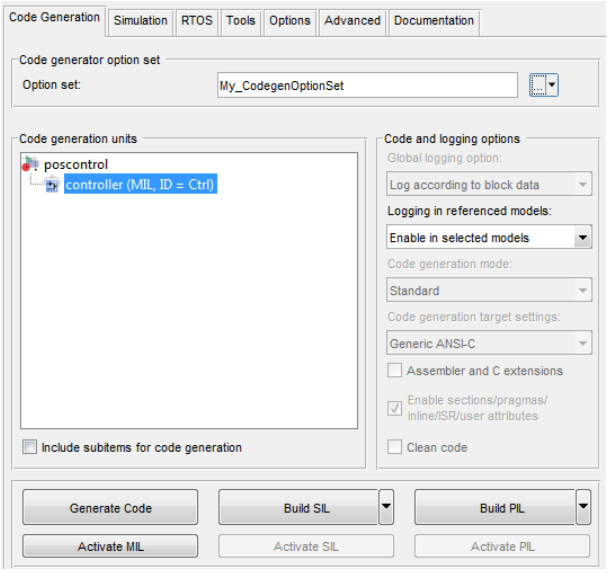
You can use a DD CodegenOptionSet object for DD-based code generation by referencing it at a DD CodeGenerationUnit object via its CodegenerationOptionSetRef property:



Model-based code generation

You can use a CodegenOptionSet object for model-based code generation by referencing it at a model's TargetLink Main Dialog block via its Option set edit field.

This edit field is located on the **Code Generation** page and the **Advanced** page of the **TargetLink Main Dialog Block**:



Note

If you reference a DD **CodegenOptionSet** object whose **Property Value List** does not contain all the available Code Generator options, TargetLink does the following:

- Uses the Code Generator options contained in the DD **CodegenOptionSet** object as specified
- Uses the Code Generator options not contained in the DD **CodegenOptionSet** object with their default value

DD-based and model-based code generation

In cases where you want to generate code for more than one CGU in one Code Generator run, make sure that all the CGUs use the same Code Generator options.

If the Code Generator options assigned to each CGU are inconsistent, TargetLink displays a message.

Note

It is usually best to reference the same DD **CodegenOptionSet** object at each CGU.
Consider transferring the Code Generator options stored in your model to a DD **CodegenOptionSet** object and reference this DD object at the **TargetLink Main Dialog** block.

Related API functions

You can use the following API functions in the context of Code Generator options:

API Function	Purpose
Exchanging Code Generator options between Model and DD	
`dsdd_export_optionset`	Writes the Code Generator options from a DD CodegenOptionSet object to the model.[1]
`dsdd_import_optionset`	Writes the Code Generator options from a model to a DD CodegenOptionSet object.[2]
`dsdd_compare_optionset`	Compares the Code Generator options specified at a model with the ones contained in a DD CodegenOptionSet object[3].
Determining the options used to generate production code	
`dsdd_get_creator_options`	Returns the Code Generator options that were used to generate the production code that is represented by the DD Subsystem object that you provided as the argument.

[1] You can call this function via the Export Options to Model <Model> context menu command of CodegenOptionSet objects or from the MATLAB Command Window.
[2] You can call this function via the Import Options from Model <Model> context menu command of CodegenOptionSet objects or from the MATLAB Command Window.
[3] You can call this function from the MATLAB Command Window.

Exchanging Code Generator options between Data Dictionaries

To exchange DD CodegenOptionSet objects between different DD files, you can use the Data Dictionary Manager's compare and merge functionalities.

For more information, refer to Comparing and Merging Data Dictionaries (📖 TargetLink Data Dictionary Basic Concepts Guide).

Related topics

Basics

Details on Preconfigured Code Generator Options (Objectives)..... 55

HowTos

How to Create DD Workspaces (📖 TargetLink Data Dictionary Basic Concepts Guide)
How to Overwrite Code Generator Options for Certain Objectives..... 58

References

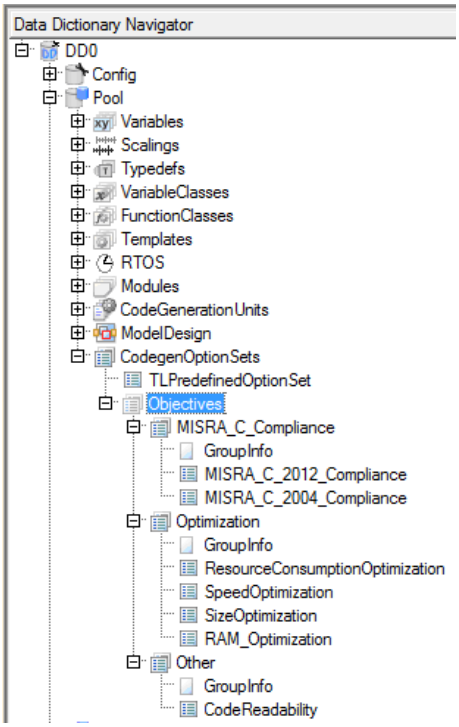
Code Generator Options Dialog (📖 TargetLink Tool and Utility Reference)
dsdd_compare_optionset (📖 TargetLink API Reference)
dsdd_export_optionset (📖 TargetLink API Reference)
dsdd_get_creator_options (📖 TargetLink API Reference)
dsdd_import_optionset (📖 TargetLink API Reference)
Edit Code Generator Options (📖 TargetLink Data Dictionary Manager Reference)
Effects of Code Generator Option Sets (📖 TargetLink Model Element Reference)
Export Options to Model <Model> (📖 TargetLink Data Dictionary Manager Reference)
Import Options from Model <Model> (📖 TargetLink Data Dictionary Manager Reference)
TargetLink Main Dialog Block (📖 TargetLink Model Element Reference)

Details on Preconfigured Code Generator Options (Objectives)

Introduction to Code Generator options

Code Generator options help you influence the optimization and the basic form of the generated code to achieve certain objectives. However, there might be conflicting objectives and underlying reasons that make it impossible to easily assess the effects of optimization options. TargetLink provides preconfigured Code Generator options (*Objectives*) that make it easier to handle Code Generator options optimized for a certain objective and minimize recurring decisions on prioritizing objectives.

Objectives are DD **CodegenOptionSet** objects optimized for a certain objective, e.g., MISRA compatibility, code readability, or resource consumption optimization. Objectives does not contain all available Code Generator options, but only Code Generator options relevant for this specific objective. These Code Generator options are available if you open a DD workspace that is based on either the **dsdd_master_advanced.dd [System]**, the **dsdd_master_autosar3.dd [System]** or the **dsdd_master_autosar4.dd [System]** system templates. If you do so, the following **Objectives** are available in `/Pool/CodeGenOptionSets`.



Tip

If you are working with different DD workspaces, you can use the Copy To (📖 TargetLink Data Dictionary Manager Reference) context menu command to copy DD objects to different DD workspaces.

Description of preconfigured Objectives

- MISRA C compliance
  - 2012

    Collects options and settings for generating production code with the fewest possible MISRA C: 2012 violations.

    The following options are not contained in this set but must still be considered, because they do not have a clear *optimum setting* but still have a considerable influence on MISRA compliance:
    - InsertVoidCastForUnusedFcnCallReturnValue
- Optimization
  - Resource consumption optimization

    Comprises options that affect code efficiency according to general optimization objectives and settings for balanced optimization according to TargetLink's optimization objectives, with the exception of high code readability.

The following options are not contained in this set but must still be considered, because they do not have a clear *optimum setting* but still have a considerable influence on code efficiency:

- AutoSelectFloatTypesForLocalMATLABVariables
- AutoSelectIntTypesForLocalMATLABVariables
- CombineControlFlowStatements
- ConsiderAllVariablesForBlockDiagramBasedSwitchOptimization
- ConsiderMergeableForBlockDiagramBasedSwitchOptimization
- DoNotUseAssignArithmeticForAccumulation
- EnableExternFunctionMerge
- ExploitRangesIndependentOfErasable
- GenerateIWriteForNonScalarIrvs
- LoopUnrollThreshold
- SupportSinglePrecisionLibraries

- Speed optimization

  Modifies resource consumption optimization wherever a decision can reduce execution time.

- Size optimization

  Modifies resource consumption optimization wherever a decision can reduce code size.

- RAM optimization

  Modifies resource consumption optimization wherever a decision can reduce the usage of static storage duration memory, i.e., RAM consumption.

- Other
  - Code readability

    Comprises options and settings that increase code readability at the cost of potentially higher resource consumption.

    The following options are not contained in this set but must still be considered, because they do not have a *readability setting* but still have a considerable influence on code readability:

    - ARDataPrototypeActualParamName
    - AvoidStaticLocalScope
    - CombineControlFlowStatements
    - DecimalConstants
    - ExploitComputeThroughOverflow
    - ForceBooleanOperandsInBooleanOperations
    - InliningThreshold
    - Optimization
    - UtilizeBitwiseShiftOperations

Additional information

You can find an application note providing additional information on the dSPACE website: http://www.dspace.com/go/TLCodeGenOpt.

Related topics

Basics

Basics on Configuring the Code Generator for Production Code Generation..... 49

HowTos

How to Overwrite Code Generator Options for Certain Objectives..... 58

References

Edit Code Generator Options (📖 TargetLink Data Dictionary Manager Reference)
Effects of Code Generator Option Sets (📖 TargetLink Model Element Reference)
Export Options to Model <Model> (📖 TargetLink Data Dictionary Manager Reference)
Import Options from Model <Model> (📖 TargetLink Data Dictionary Manager Reference)

How to Overwrite Code Generator Options for Certain Objectives

Objective

To overwrite existing (default) Code Generator options with preconfigured Code Generator options (**Objectives**).

Preconditions

Your current DD workspace contains the **CodegenOptionSetGroup** object named **Objectives**.

Preparation aspects for comparing Code Generator options

You can use **Objectives**, i.e., preconfigured subsets of Code Generator options to overwrite existing Code Generator options. This lets you optimize your code or make it compliant with coding guidelines, such as MISRA.

Suppose you already set Code Generator options at your model and want to apply preconfigured **Objectives** from the Data Dictionary, e.g., for test and evaluation purposes. You are recommended to save your original Code Generator options before overwriting them with the new ones. This also allows you to compare Code Generator options and their influence on the generated code afterwards.

Note

Because existing Code Generator options are always overwritten, it is crucial to add the most important Code Generator options last.

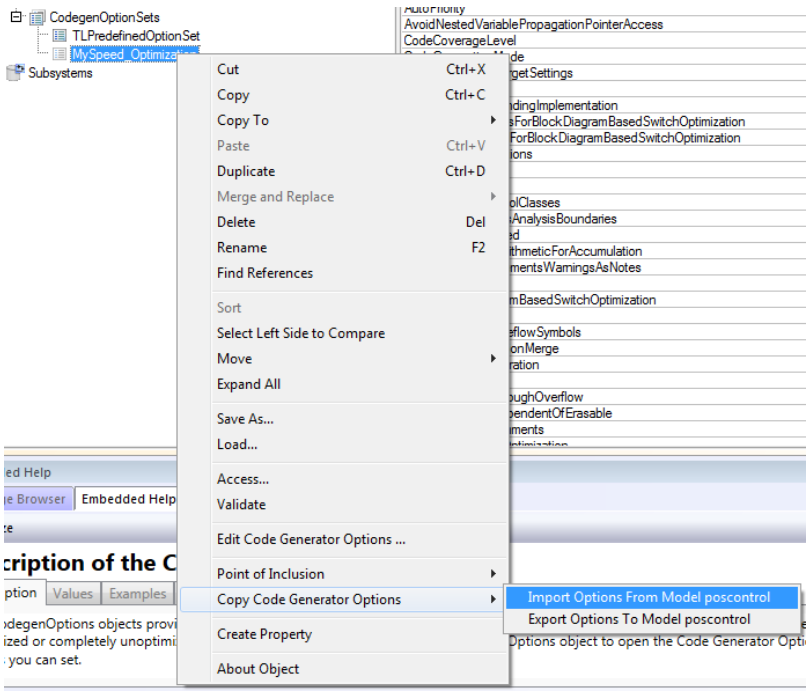
Method

To overwrite Code Generator options for certain objectives

1 Open the model providing the Code Generator option set you want to overwrite.

2 From the context menu of the DD **CodegenOptionSet** object, select **Create CodegenOptionSet** and enter a suitable name.

3 Import the Code Generator option set saved at your model into the Data Dictionary via the **Import Options From Model <model name>** context menu command.



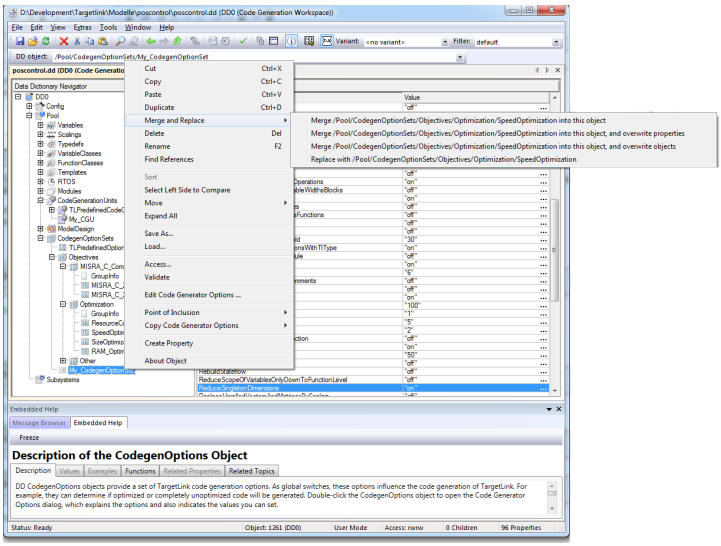
For more information, refer to Import Options from Model <Model> (📖 TargetLink Data Dictionary Manager Reference).

4 Create a copy of the **CodegenOptionSet** object (`Ctrl` + `D` - Duplicate (📖 TargetLink Data Dictionary Manager Reference)) and name the new DD object accordingly.

For more information, refer to Basics on Handling Data Dictionary Objects via API (📖 TargetLink Data Dictionary Basic Concepts Guide)

5 Select the **Objectives** object you want to apply and copy (`Ctrl` + `C`) it.

6 From the context menu of the CodegenOptionSet object you want to overwrite, select **Merge and Replace** and then the action you want to perform.



If you apply multiple **Objectives** objects one after the other, you have to start with the least important **Objectives** object and finish with the most important one.

7 Compare the modified Code Generator options with the original one.

For more information, refer to How to Compare DD Objects Using the DD Two-Way Comparison (📖 TargetLink Data Dictionary Basic Concepts Guide).

8 Check the assumptions on code generation and optimization for the preconfigured Code Generator options set to be applied.

If the assumptions on code generation and optimization made by TargetLink do not match the requirements of your development environment, modify them accordingly.

9 Reference the Code Generator option set to be used during code generation at a DD **CodeGenerationUnit** object via its **CodegenerationOptionSetRef** property (DD-based code generation), or in the **TargetLink Main Dialog** via its **Option set** edit field (Model-based code generation).

For information on referencing Code Generator options, refer to Basics on Configuring the Code Generator for Production Code Generation on page 49.

10 Generate production code.

Result

You have overwritten Code Generator options and generated code for the selected CGU.

Related topics

Basics

Basics on Configuring the Code Generator for Production Code Generation..... 49
Details on Preconfigured Code Generator Options (Objectives)..... 55

References

Effects of Code Generator Option Sets (📖 TargetLink Model Element Reference)

Basics on Generating Code Compliant with C99/C11/C++

Support of C and C++ standards

TargetLink supports the common subset of the C90/ANSI C standard and newer standards: TargetLink can be configured to generate code that complies with the stricter rules regarding identifiers of the newer standards, but does not use features added by these standards.

By default, TargetLink generates C90-compliant code.

The following standards are supported:

- C99
- C11
- C++11
- C++14
- C++17

In addition to the common subset, some features of the C99 standard are supported. Refer to Generating code compliant with C99/C11 on page 64.

To generate code compliant with the newer standards, the use of certain identifiers must be restricted.

Restricting identifiers via DD IdentifierRestriction objects

To restrict identifiers, you must use DD IdentifierRestriction objects. Via DD IdentifierRestriction objects, you can specify lists of identifiers so they are treated as one of the following:

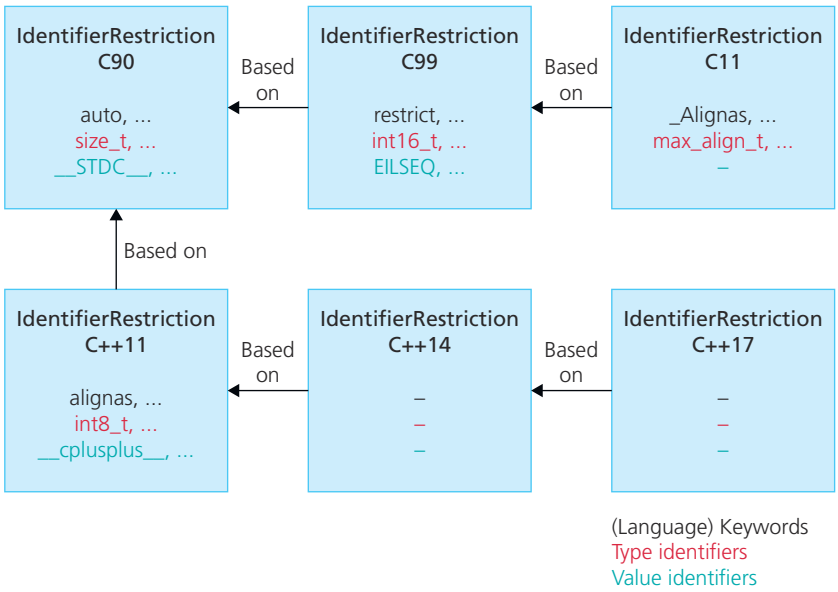
- Language keywords
- Typedefs or tag names
- Compile-time constants

A DD IdentifierRestriction object can reference another DD IdentifierRestriction object via the BasedOnRef property. This extends the referenced object with additional identifiers or stricter behavior for identifiers both objects have in common.

Identifier restrictions for the supported standards TargetLink provides a predefined DD IdentifierRestriction object for the C90/ANSI C standard that is read-only and always active. This ensures that TargetLink production code is always C90-compliant.

Additionally, TargetLink provides one preconfigured DD IdentifierRestriction object with all the identifiers required for each of the supported standards.

The preconfigured DD IdentifierRestriction objects are specified either as extensions of the C90 object (C99 and C++11 objects) or as extensions of other preconfigured DD IdentifierRestriction objects. For example, the identifier restrictions specified via the C99 object include all keywords and identifiers defined by the C99 standard that are not part of the C90 standard.



The preconfigured DD IdentifierRestriction objects are part of all system templates except the dsdd_master_basic.dd [System] system template.

Custom Identifier Restrictions You can create additional DD IdentifierRestriction objects to specify keywords and identifiers as required for your project.

The following approach is recommended:

1. Create a new DD IdentifierRestriction object.
2. Use the BasedOnRef property of the new object to specify a preconfigured or predefined DD IdentifierRestriction object whose restrictions you want to extend.
3. Specify all *additional* keywords and identifiers, as well as stricter categorizations for identifiers, to apply during code generation via the new object. For instructions, refer to How to Specify Custom Identifier Restrictions on page 67.

Different specifications for the same identifier Different specifications for the same identifier will be resolved to a consistent specification by TargetLink. The following rules apply:

Difference	Resulting Restriction
Severity	Highest specified severity
Categorization	Always keyword
Severity and categorization	Keyword and highest specified severity

TargetLink displays a message of the Advice type to inform you about the adjustments.

This is the case if, for example, the preconfigured DD IdentifierRestriction objects for the C99/C11 and the C++ 11/14/17 standards are active simultaneously. Because these objects represent the respective language standard, different categories for the same identifier between the language standards result in different specifications of the preconfigured DD IdentifierRestriction objects.

Additionally, the message severity for keyword use is different for the preconfigured C99/C11 and Cpp11/Cpp14/Cpp17 IdentifierRestriction objects.

All cases of differing specifications in the predefined and preconfigured DD IdentifierRestriction objects and the resulting specification are shown in the following table:

Identifier	C90/C99 IdentifierRestriction Object	Cpp11/Cpp14/Cpp17 IdentifierRestriction Object	Result
`bool`	Typedef (C99)	Keyword	Keyword
	Advice	Advice	Advice
`true`, `false`	Macro (C99)	Keyword	Keyword
	Advice	Advice	Advice
`wchar_t`	Typedef (C90)	Keyword	Keyword
	Warning	Advice	Warning
`inline`	Keyword (C99)	Keyword	Keyword
	Warning	Advice	Warning

TargetLink ensures a single, correct specification for each of these identifiers. The message A10747 is displayed only to inform you about the adjustments, i.e., it can be safely ignored.

If you want to avoid the message A10747, the following options are available:

Use Case	Options
C99/C11-compliant code	Use one of the following options: ▪ Set the IsActive property of all DD IdentifierRestriction objects that represent a C++ language standard to `off`. ▪ Remove the identifiers from the Keywords list of the Cpp11 DD IdentifierRestriction object. **Note** Removing these identifiers will not reduce the severity of the corresponding message. The message severity specified via the DD IdentifierRestriction objects for the C language standards is at least as high as the one specified via the DD IdentifierRestriction objects for the C++ standards.
Production code for integration in C++ projects	Remove the identifiers from the Keywords list of the Cpp11 DD IdentifierRestriction object. **Note** You are recommend to avoid changes to the ANSI_C, C99, and C11 IdentifierRestriction objects.

Generating code compliant with C99/C11

TargetLink supports the common subset of C90 and C99/C11. To generate code compliant with these standards, the following preconditions must be fulfilled:

- If you want to generate C99-compliant code, the IsActive property of the preconfigured C99 DD IdentifierRestriction object must be set to `on`.
- If you want to generate C11-compliant code, the IsActive property of the preconfigured C99 and C11 DD IdentifierRestriction objects must be set to `on`.
- The UtilizeBitwiseShiftOperations Code Generator option must be set to any value except `Automatic`.

In addition to the common subset, the following features of the C99 standard are supported:

C99 Feature	Implementation	Documentation
`_Bool` and C99 fixed-width integer data types with an exact width, such as `uint8_t`. TargetLink does not support the following: ▪ `_Bool` as a bitfield type ▪ `<stdbool.h>` with `bool`, `true`, and `false`	*Coded type* available via: ▪ **Code generation target settings** set to `Generic C99`. [1] ▪ Adjustment of the `TargetConfig.xml`.	▪ Example of Controlling the Generation of the File Containing TargetLink Base Data Types on page 143 ▪ How to Clone Target/Compiler Combinations to Outside the TargetLink Installation on page 44
`<math.h>` float type variants	Code Generator option: **SupportSinglePrecisionLibraries**	▪ Basics on Configuring the Code Generator for Production Code Generation on page 49 ▪ SupportSinglePrecisionLibraries (📖 TargetLink Model Element Reference)
`inline` function specifier	DD **CodeDecoration** objects.	Basics on Code Decorations on page 183
`__func__`	Variables or macros referencing DD **VariableClass** objects whose **Alias** property is set to `on`.	▪ Basics on Variable Classes on page 88 ▪ VariableClass
`restrict` type qualifier **Note** Because of the nature of the `restrict` type qualifier, TargetLink cannot validate the constraints and semantics of this qualifier. Using this feature is not recommended.	Specifying type qualifiers via the DD **TypePrefix** property of a DD **VariableClass** object.	Example of Specifying Type Qualifiers via Type Prefixes on page 122

[1] For simulation purposes, typedef declarations are generated according to the `TargetConfig.xml` of the selected simulation platform. Therefore, you must adjust the *Coded Type* property in the `TargetConfig.xml` to perform PIL simulation with the C99 data types. Refer to How to Clone Target/Compiler Combinations to Outside the TargetLink Installation on page 44. For SIL simulation, this is not possible.

Integrating production code in C++

You can integrate the C production code TargetLink generates into C++ projects via the following options:

- As `extern C`-encapsulated C code.
- As code that is compiled with a C++ compiler.

The **UtilizeBitwiseShiftOperations** Code Generator option must be set to any value except `Automatic`.

The production code uses only C90 and, optionally, some C99 features. Therefore, calling a C++ functionality from the production code usually requires adapter code that is not provided by TargetLink.

Generating Extern C Encapsulated Code To generate production code with extern C encapsulation, the following options are available:

- Code decorations. Refer to Basics on Code Decorations on page 183.
- EmitEncapsulationByExternC Code Generator option. Note that modules whose Language property is set to `Cpp` are not subject to this encapsulation.

Compiling as C++ Compiling the TargetLink production code with an external C++ compiler is possible if the following condition is fulfilled before code generation:

- The IsActive property of the C++11 DD IdentifierRestriction object is set to `on`.
- Optionally, the IsActive property of the C++14 and C++17 DD IdentifierRestriction objects can be set to `on` as well.
- Optionally, set the Language property of the DD ModuleInfo object to `Cpp`. This deactivates the `extern C` encapsulation and changes the file name extension to `.cpp`. You can change the default file name extensions for modules whose Language property is set to `Cpp` via the following Code Generator options:

Code Generator Option	Description	Default
DefaultCppHeaderFileExtension	Defines the default file name extension for header files containing C++ code.	h
DefaultCppSourceFileExtension	Defines the default file name extension for source files containing C++ code.	cpp

- Optionally, you can activate the EmitAllTagNameConflictsAsErrors Code Generator option.

The production code generated with these settings can be compiled as C++ code.

Note

TargetLink does not support SIL or PIL simulation for C++. Building and testing C++ code is out of scope for TargetLink.

Related topics

HowTos

How to Specify Custom Identifier Restrictions..... 67

References

EmitAllTagNameConflictsAsError (📖 TargetLink Model Element Reference)
EmitEncapsulationByExternC (📖 TargetLink Model Element Reference)
SupportSinglePrecisionLibraries (📖 TargetLink Model Element Reference)
UtilizeBitwiseShiftOperations (📖 TargetLink Model Element Reference)

How to Specify Custom Identifier Restrictions

Objective

To generate code compliant with standards other than C90, the use of identifiers must be restricted. To restrict the use of identifiers, you must use DD **IdentifierRestriction** objects.

Support of standards other than C90

TargetLink supports the common subset of C90 and select other standards. Refer to Basics on Generating Code Compliant with C99/C11/C++ on page 61.

Restrictions

This instruction focuses on custom restrictions for identifiers that you can specify in addition to the preconfigured DD **IdentifierRestriction** objects provided by TargetLink.

If you want to generate code compliant with the restrictions of one or more of the supported standards, but no additional restrictions, the use of the preconfigured DD **IdentifierRestriction** objects is sufficient. Refer to Basics on Generating Code Compliant with C99/C11/C++ on page 61.

Precondition

The following preconditions must be fulfilled:

- You opened the DD project file of the project whose identifiers you want to restrict.
- You merged the preconfigured DD **IdentifierRestriction** objects into the DD project file. Refer to How to Merge or Replace a DD Object Using the DD Two-Way Comparison (📖 TargetLink Data Dictionary Basic Concepts Guide).

Method

To specify custom identifier restrictions

1 In the Data Dictionary Navigator, right-click the `/Pool/IdentifierRestrictions/` object.

2 On the context menu, select **Create IdentifierRestrictionGroup**.

3 Right-click the **IdentifierRestriction** group and select **Create IdentifierRestriction**.

4 Rename the **IdentifierRestriction** object as required.

5 Optionally, specify the **BasedOnRef** property in the **Property Value list** to reference a suitable DD **IdentifierRestriction** object you want to extend.

> **Tip**
>
> We recommend extending the preconfigured DD **IdentifierRestriction** objects.

6 In the **Property Value List**, specify the following settings:

Property	Description
Keywords	Specify the identifiers you want TargetLink to treat as language keywords. The identifiers must be valid C identifiers. **Tip** Identifiers can be copied to and pasted from external sources via the **Advanced Edit String List** (📖 TargetLink Data Dictionary Manager Reference) dialog.
EmitKeywordUseAs	Choose the severity of the message TargetLink displays if a keyword specified via the **Keywords** property is used.
AllowedAsTypeIdentifier	Specify the identifiers you want TargetLink to treat as typedefs or tag names. The identifiers must be valid C identifiers.
EmitTypeIdentifierUseAs	Choose the severity of the message TargetLink displays if a typedef or tag name specified via the **AllowedAsTypeIdentifier** property is used outside of type identification.
AllowedAsValueIdentifier	Specify all identifiers you want TargetLink to treat as compile-time constants, i.e., as values. The identifiers must be valid C identifiers.
EmitValueIdentifierUseAs	Choose the severity of the message TargetLink displays if a compile-time constant specified via the **AllowedAsValueIdentifier** property is used outside of macro or enumeration constants.
IsActive	Set the value to **on** if you want to apply the identifier restrictions during code generation.

Result

You created a DD **IdentifierRestriction** object to specify restricted identifiers and the severity of the messages associated with them.

Related topics

Basics

Basics on Generating Code Compliant with C99/C11/C++..... 61

Specifying the Location of Artifacts Generated or Used by TargetLink

Where to go from here

Information in this section

Basics on Specifying the Location of Artifacts Generated by TargetLink..... 71

How to Specify the Location of Artifacts Generated by TargetLink..... 74

Basics on Specifying the Location of Artifacts Generated by TargetLink

Introduction

TargetLink creates several artifacts during code generation and when building the simulation application.

You can specify artifact locations in the Data Dictionary to determine their location in the file system. Artifact locations are grouped in a folder structure that is specified relative to a project folder.

Specifying the project folder

You specify a project folder via the DD **ProjectFolder** object, located in `/Pool/ArtifactsLocation/ProjectFolders`.

The path specified via its **RootPath** property is the root for each artifact location.

You can specify the path as follows:

- As an absolute path in your file system, e.g., `D:\MyProjectFolder`
- As a path relative to your current working directory, e.g., `.\SomeFolder\SomeOtherFolder\MyProjectFolder`

- Via DD environment variables created via the [envIdx] = dsdd('SetEnv',<envIdentifier>,<envValue>); API function

  For example: `$MyDDEnvVar$\MyProjectFolder`.

The ProjectFolder object can be globally used for all code generation units (CGUs) that do not have a project folder specification or locally for a specified CGU. This is shown in the following table:

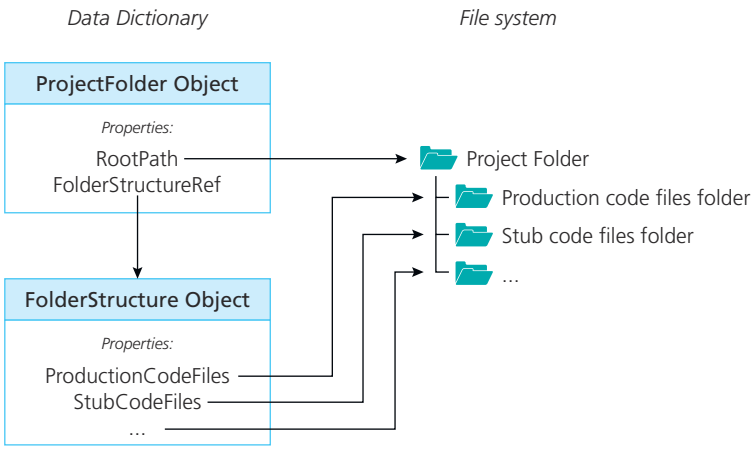
Value of DD IsDefaultProjectFolder Property	CGU Specification	Description
on	–	Uses this project folder for all CGUs that do not have a project folder specification.
off	The name of a model-based CGU provided via the DD SystemName property	Uses this project folder for the specified CGU.
	A DD CodeGenerationUnit object referenced via the DD CodeGenerationUnitRef property	

Default project folder The predefined DD ProjectFolder object called `TLPredefinedProjectFolder` provides the default project folder. In the context menu you can reset all properties to their default value.

Specifying the folder structure

You specify artifact locations as property values of a DD FolderStructure object.

Each artifact location must be relative to the root of the project folder, which is specified in the ProjectFolder object. The ProjectFolder also references this FolderStructure object in its FolderStructureRef property.



Default artifact locations The predefined DD FolderStructure object called `TLPredefinedFolderStructure` provides default locations for each kind of artifact. In the context menu, you can reset all properties to their default value.

Supported name macros TargetLink supports several name macros for specifying the artifact locations by means of the properties of the DD FolderStructure. Refer to Macro expansion for artifact locations on page 85.

Note

CGU-specific paths must contain name macros

The path to the folder must contain the `$(CGU)` name macro and, if you use code variants, the `$(Variant)` name macro.

Refer to the following predefined DD object:

`/Pool/ArtifactsLocation/FolderStructures/TLPredefinedFolderStructure`

TargetLink displays an error message if a mandatory name macro is missing.

Custom artifact locations You can specify additional artifact locations by adding custom string properties to the DD FolderStructure object. You can then use these artifact locations for tool chain automation via the API.

For an example, refer to the MODULAR_DEVELOPMENT demo.

Handling production code object files

When building the simulation application using a C compiler, TargetLink compiles object files from the generated production code. Via the ArchiveObjectFileHandling property of the ProjectFolder object, you can instruct TargetLink how to handle these object files.

Value of DD ArchiveObjectFileHandling Property	Description
`DoNotUseArchive`	Does not archive production code object files when building the simulation application.
`OverwriteArchive`[1]	Archives production code object files when building the simulation application. Object files in the folder that were specified via the ObjectFilesArchive property of the FolderStructure object are overwritten.
`UseArchive`[1]	Uses the production code object files in the folder that were specified via the ObjectFilesArchive property of the FolderStructure object when building the simulation application.

[1] The ObjectFilesArchive property of the FolderStructure object must be set.

Archiving DD Subsystem objects

When generating code, TargetLink stores metadata on the generated code in a CGU-specific DD Subsystem object.

Via the ProjectFolder.SaveCodeGenerationMetadata property you can instruct TargetLink to archive the DD Subsystem object corresponding to the CGU that uses the project folder as a metadata file.

Value	Description
on	Saves code generation metadata (DD **Subsystem** object) as a metadata file to the folder specified in the **MetadataFiles** of the **FolderStructure** object.
off	Does not save code generation metadata. (default)

Note

When specifying an artifact location for distributed development, you must set the **SaveCodeGenerationMetadata** property to **on** to facilitate the integration of the different model components within the integration model to avoid an unnecessary re-generation of the components' production code. For an example, refer to the MODULAR_DEVELOPMENT demo.

Limitations

The following limitations apply to artifact locations and project folder specification:

- Relative paths specified in **FolderStructure** objects must not contain double dots.
- UNC paths are not supported.
- Windows environment variables are not supported.
- Artifact locations as defined in **FolderStructure** objects must not be changed after code generation so that following processes in TargetLink can find them

Related topics

Basics

MODULAR_DEVELOPMENT (TargetLink Demo Models)

How to Specify the Location of Artifacts Generated by TargetLink

Objective

To specify artifact locations.

Workflow

This workflow consists of the following parts:

- Part 1 - Specifying the folder structure
- Part 2 - Specifying the project folder

Part 1

Specifying the folder structure

1 From the context menu of the DD **ArtifactsLocation** object, select **Expand All**.

2 From the context menu of the DD FolderStructures object, select Create FolderStructure and rename the resulting DD FolderStructure object to `<MyFolderStructure>`.

3 In its Property Value List, adjust each property as required.

Note

Several property values require name macros, refer to FolderStructure object.

Interim result

You specified the folder structure that comprises several artifact locations. Next you specify the project folder.

Part 2

Specifying the project folder

1 From the context menu of the DD ProjectFolders object, select Create ProjectFolder and rename the resulting DD ProjectFolder object to `<MyProjectFolder>`.

2 Via the FolderStructureRef property, reference the DD FolderStructure object called `<MyFolderStructure>`.

3 Specify the path to the project folder via the DD RootPath property.

4 Specify the ProjectFolder object's IsDefaultProjectFolder property as required:

- on - If you want to use this project folder for every Code generation unit (CGU) that does not have a project folder specification.

  Note

  Make sure that the IsDefaultProjectFolder property is set to on at exactly one DD ProjectFolder object.

- `off` - If you want to use this project folder for a CGU specified via the SystemName or CodeGenerationUnitRef properties.

5 Specify all other properties as required.

Note

If you want to use this project folder in the context of distributed development, set the DD SaveCodeGenerationMetadata property to on.

Result

You specified the location of the artifacts generated by TargetLink.

Related topics

Basics

Basics on Specifying the Location of Artifacts Generated by TargetLink..... 71
MODULAR_DEVELOPMENT (📖 TargetLink Demo Models)

Adapting Code to Company Coding Styles

Influencing code generation process

TargetLink provides various means for influencing the code generation process. This may be necessary or even mandatory to comply with your company modeling and coding style.

Where to go from here

Information in this section

Generating Unique Names via Name Macros..... 78
Working With Variable Classes..... 88
Working With Data Types..... 127
Working With Structures and Pointers to Structures..... 148
Customizing Production Code Generation via Hook Scripts..... 178
Decorating Production Code..... 183
Customizing TargetLink's Code Formatting..... 197

Information in other sections

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

Generating Unique Names via Name Macros

Introduction

TargetLink can generate the name of a variable (a block output, state or parameter) used in the generated production code by using name macros automatically. During code generation, each macro is replaced by the string it represents.

Where to go from here

Information in this section

Basics on Using Name Macros..... 78

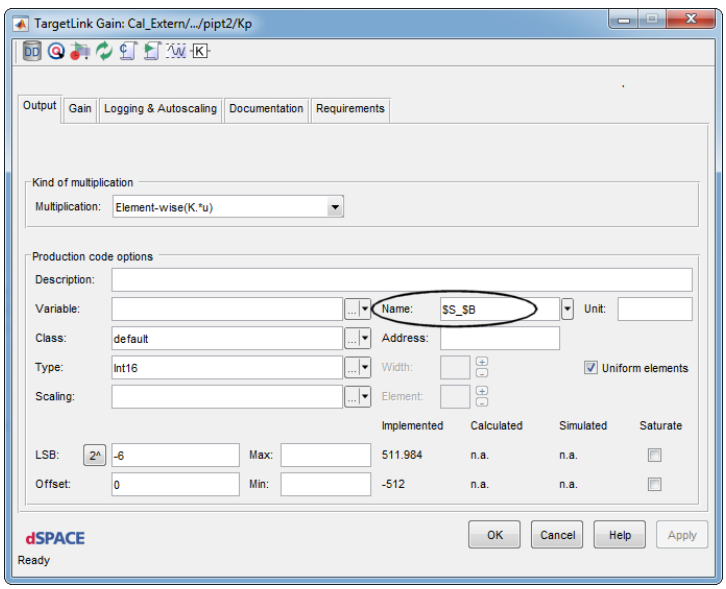
Details on the Expansion of Name Macros..... 80

Basics on Using Name Macros

Using 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.



The illustration above shows the **Output** page of the **Gain** block dialog. The name of the output is set to `$S_$B`. By default, block outputs in TargetLink are named `$S_$B`. This ensures a unique identifier because block names in Simulink are always unique in a subsystem, while a unique identifier for the Simulink subsystem is generated for $S.

Example Suppose you have a model with a TargetLink subsystem at the root level. If the subsystem contains a block `Kp` with $S_$B as its output name, the variable name in the production code generated for the TargetLink subsystem is `Sa1_Kp`.

The name is put together as follows:

- The letter `S`.
- `a` is the current TargetLink subsystem ID.
- Because the block resides at the topmost level in the TargetLink subsystem, it is identified by the number `1`.
- `Kp` denotes the block name.

Names of blocks inside a subsystem must be unique. If there is another block called `Kp`, the variable for the output of this block is:

- `Sa2_Kp` if the block resides in a Simulink subsystem encapsulated by the TargetLink root subsystem. The number `2` denotes the second subsystem level.
- `Sb1_Kp` if the block resides in a second TargetLink subsystem outside of the first subsystem. The letter b denotes the second TargetLink subsystem.

While unique code identifiers are always unique if you combine the `$S` and the `$B` macro, other combinations might introduce ambiguous variable identifiers that result in Code Generator error messages. Therefore, you have to specify reasonable variable names and name macros. This strict approach ensures that code identifiers are always generated as specified.

Note

C code identifiers can contain only letters, digits, and underscores. The first character must not be a digit. However, C file names can contain letters, digits, underscores, and hyphens, and they can start with any of these characters. You can specify C file names, for example, in the Function or Addfile block, or via the **Module** property of a DD object.

No such limitations apply for Simulink block names and signal labels. They can contain any arbitrary character. If the macros $B, $L, or $P are used with block names, the Code Generator replaces incompliant characters with underscores.

Further context-sensitive information is available:

- Details on the Expansion of Name Macros on page 80
- Block-Related Name Macros (📖 TargetLink Model Element Reference)
- Data Dictionary-Related Name Macros
- Stateflow-Related Name Macros (📖 TargetLink Model Element Reference)
- Configuring Names in Multirate Models (📖 TargetLink Multirate Modeling Guide)

Names of non-scalar variables

Fixed names are used for non-scalar variables (vectors and 2-D matrices) referencing a variable class whose DD Macro property is set to on. Because C semantics does not allow macros representing vectors or matrices, TargetLink will define a macro for each element of the vector or matrix using the fixed name with a suffix for the indices of the element:

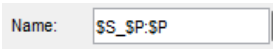
```
MyVector_0 … MyVector_n
MyMatrix_0_0 … MyMatrix_n_m
```

Function arguments

A special problem arises if a block output or parameter becomes a function argument. The Code Generator needs two names for the following:

- The formal function argument in the function
- The actual function argument in the caller function

To solve this problem, TargetLink lets you specify both names separated by a : in the Name edit field of the block dialog, for example:



This means the name of the formal argument is $S_$P and the name of the actual argument is $P.

Related topics

Basics

Configuring Names in Multirate Models (📖 TargetLink Multirate Modeling Guide)
Details on the Expansion of Name Macros..... 80

References

Block-Related Name Macros (📖 TargetLink Model Element Reference)
Stateflow-Related Name Macros (📖 TargetLink Model Element Reference)

Details on the Expansion of Name Macros

Macro expansion during code generation

During code generation, the name macros are replaced by the strings they describe. This is shown in the following tables:

Frequently Used Name Macros		
Macro	**Replaced By**	**Example**
$B	Block name. Special characters that are not valid in a C variable name are replaced by underscores.	Expands to Gain for a block named Gain.

Frequently Used Name Macros		
Macro	**Replaced By**	**Example**
`$C`	Unique identifier for the Stateflow state. It consists of the character C, the associated TargetLink subsystem ID, and an integer number that identifies the object. It also enables automatic name expansion to resolve name ambiguities and make it possible to place the variable in instance structures for function reuse.	Expands to `Ca5` for the 6th Stateflow object in a TargetLink system.
`$D`	TargetLink Data Dictionary object name. If there is no DD object, it is interpreted as being DD_$S_$B for variables and functions from blocks, and DD_$C_$O for Stateflow objects.	Expands to `x` for a **TargetLink Data Dictionary** object named x.
`$f`	The function name as is (case-sensitive).	Expands to `Fcn` for a function called `Fcn`.
`$F`	The function name in capital letters.	Expands to `FCN` for a function called `Fcn`.
`$H`	The name of the hierarchy state level in which the Stateflow object is placed.	Expands to `Running_Slow` for a Stateflow object placed in Running/Slow.
`$I`	Expands to the ID specified in one of the following locations: ▪ Via the **Subsystem ID** edit field on the **Options** page of the **Function Block** or for the TargetLink Root system in the Main Dialog. ▪ For DD-based Code generation unit (CGU)s in the **CodeGenerationUnitID** property of DD **CodeGenerationUnit** objects.	-
`$L`	Name of the signal line. If the block output has no label, the block name is used.	Expands to label of the signal. If no label exists, expands to $B.
`$M`	The model name.	Expands to `pipt1` for a model file named `pipt1.slx`.
`$N`[1)]	The name[2)] of the code generation unit (CGU). For DD-based code generation units, `$N` expands to `<NameOfDDbasedCGU>_$I`.	Expands to `controller` for a TargetLink subsystem named controller in the Simulink model.
`$O`	The name of the selected Stateflow object, which can be: ▪ The state/event name ▪ The input or output ▪ Local, constant, temporary or workspace data ▪ The chart/machine name ▪ The name of a,graphical function	Expands to `x` for a Stateflow object named `x`.
`$P`	The block parameter name. If a parameter is specified as a MATLAB expression, special characters are replaced by underscores.	Expands to `GainParam` for a gain value named GainParam that is evaluated in the MATLAB workspace.

Frequently Used Name Macros		
Macro	**Replaced By**	**Example**
`$R`	Used to enable automatic name expansion to resolve name ambiguities and to make it possible to place the variable in instance structures for function reuse. Has no effect if there is no name conflict with another variable.	Expands to '' (empty string) for an instance #1, and to '_a' for an instance #2.
`$S`	The Simulink subsystem in which the block resides. The replacement string consists of the following elements: ▪ The character `S` ▪ The associated subsystem ID `$I` ▪ An integer number `<n>` that identifies the subsystem It also enables automatic name expansion to resolve name ambiguities and make it possible to place the variable in instance structures for function reuse.	Expands to `Sa1` for the first subsystem in a TargetLink system.
`$T`	The shortcut of the base type specified in the TargetLink Data Dictionary at the DD Typedef object.	Expands to `I16` for a variable of base type Int16 if Int16.BaseTypeCut = 'I16'.
`$v`	The variable name as is (case-sensitive).	Expands to `Ki` for a variable named `Ki`.
`$V`	The variable name in capital letters.	Expands to `KI` for a variable named `Ki`.

1) Must not be used for function names, inputs, and outputs of subsystems configured for incremental code generation. During code generation, the $N of the incremental function and of the superordinate subsystem are replaced by different subsystem names. Thus, the function interface does not match.

2) The maximum number of characters is 58.

Further Name Macros		
Macro	**Replaced by**	**Example**
`$(BaseType)`	Name of the base types specified in the TargetLink Data Dictionary.	Expands to `Int16` for a variable of Int16 base type.
`$(Component)`	Evaluates to the name of the software component that the variable or function belongs to. (AUTOSAR-specific)	-
`$(DataVariantName)`	Expands to the name of the DD DataVariant object.	–
`$(Dim1Width)`, `$(Dim2Width)`	▪ `$(Dim1Width)` - Replaced by the number of elements in the first dimension (row of a 2-D variable). ▪ `$(Dim2Width)` - Replaced by the number of elements in the second dimension (column of a 2-D variable). No effect for scalar variables. For vector variables, only `$(Dim1Width)` is applicable.	–
`$(FunctionKind)`	This macro expands to the function kind which consists of the detection kind and the parameter kinds.	–

Further Name Macros		
Macro	**Replaced by**	**Example**
`$(FunctionName)`	Expands to the original name template specified at a Stateflow object or the original name of a look-up table function.	–
`$(Group)`	This macro is replaced either by the name of the DD group in which it is located (without path) or, for structure components, by the name of the group in which the structure resides. If there is no DD object that belongs to the name template, this macro value is empty. All characters unequal to 'a'…'z', 'A'…'Z', '0'…'9' are replaced by '_' before the macro value is inserted into the name template.	Expands to `MyVariables` for a DD Variable that resides in the variable group MyVariables.
`$(Module)`	The name of the module in which the object is defined.	Expands to `MyModule` for a module named MyModule.
`$(PageNumber)`	Expands to an integer number counting the pages.	–
`$(Prm)`	The evaluation of this macro depends on the AUTOSAR release selected for code generation: ▪ Evaluates to `Calprm` for an AUTOSAR release prior to 4.0. ▪ Evaluates to `Prm` for an AUTOSAR release later than 4.x.	–
`$(ReplaceableDataItem)`	Expands to the name of the DD ReplaceableDataItem object implemented by this variable.	–
`$(Section)`	Expands to the number of a sub reuse struct.	–

`$(SPath)`	The path from the root structure to the component, excluding root structure and component
	▪ If $(SPath) is used in a macro body, the substructures are separated by dots.
	▪ If $(SPath) is used in a C identifier, the substructures are separated by underscores.
		Expands to
		▪ `.MySubStruct1.MySubStruct2` in macro bodies
		▪ `_MySubStruct1_MySubStruct2` in C identifiers
		if it is used for a component of the following structure description:
		`struct SubStruct2{`
		`  int  a;`
		`  float  b;};`
		`struct SubStruct1{`
		`  int  i;`
		`  struct SubStruct2  MySubStruct2;};`
		`struct StructT{`
		`  double  F[3];`
		`  struct SubStruct1  MySubStruct1;};`
		`typedef struct StructT StructType;`
		`StructType MyStruct;`

`$(SType)`	The value of $(Type) for the root structure variable.	Expands to `MyStructType` for a structure type named MyStructType.

Further Name Macros		
Macro	**Replaced by**	**Example**
`$(Sv)`	Variable name of the root structure variable.	Expands to `Mystruct` for a structure named MyStruct.
`$(SV)`	Variable name of the root structure variable in capital letters.	Expands to `MYSTRUCT` for a structure named MyStruct.
`$(Type)`	▪ Name of the user-defined typedef, if the IsBaseType property is set to `off` ▪ Name of the base type, if the IsBaseType property is set to `on`.	Expands to `x` for a DD typedef named x.
`$(VarAccess)`	This macro is allowed in access function templates to specify macro bodies and gives correct access to the relevant variable, as and when required: For non-struct component variables, it is replaced in the same way as `$v`, for struct component variables, it is replaced in the same way as `$(Sv)$(SPath).$v`.	–
`$(VariantItemId)`	Expands to the value of the ID property of the DD VariantItem object.	–
`$(VariantItemName)`	Expands to the name of the DD VariantItem object.	–
`$E(expr)`	The actual value (representation) of *expr* that is evaluated in the MATLAB workspace. It must evaluate to a string. The string *expr* can contain name macros (even `$E`(expr1)) when evaluated. `$E` can be used in address fields of TargetLink blocks, or in combination with the `$THIS` macro (refer to $E, $THIS, and $CHART on page 84).	Expands to foo if the expression is `b` and a variable b = 'foo' exists in the MATLAB workspace.

$E, $THIS, and $CHART

You can use the `$E` and `$THIS` macros to pass block handles to M scripts. For example, `$E(MyFunction($THIS))` passes the handle of the block, Stateflow object, or MATLAB object in which this name macro is used to to `MyFunction.m`.

Additionally, you might have to use the `$CHART` name macro. This is the case, if the following conditions apply:

- You use a library that contains Stateflow objects or ML Fcn objects.
- Your model contains several instances of the library objects.
- You want to generate different names for the library object instances in the production code.

`$CHART` expands to the handle of the library object instance. To address an object within the instance of a library object, you can use `$THIS` again.

Consider the following example of a ML Fcn Input and two instances:

- `Subsystem/Instance1/MyMATLABFunction.in1`
  - `$THIS` expands to the ID of `in1` contained in the library, e.g., `42`.
  - `$CHART` expands to the handle of the instanced library object `Subsystem/Instance1/MyMATLABFunction`, e.g., `31.14`.
- `Subsystem/Instance2/MyMATLABFunction.in1`
  - `$THIS` expands to the ID of `in1` contained in the library, e.g., `42`.
  - `$CHART` expands to the handle of the instanced library object `Subsystem/Instance2/MyMATLABFunction`, e.g., `53.14`.

Note

The M script must reside on a MATLAB search path.

Macro expansion for artifact locations

The following table shows the expansion of name macros for artifact locations:

Macro	Replaced by	Example
`$(Board)`	Expansion depends on the simulation mode: - SIL - Name macro expands to `HostPC64`. - PIL - Name macro expands to the name of the evaluation board. Supported in the value of the DD ObjectFilesArchive property.	-
`$(CGU)`	Expansion depends on the kind of code generation unit (CGU): - Model-based CGU - The name macro expands to `<CGUName>`. - DD-based CGU - The name macro expands to `<CGUName_CGUID>`. Mandatory in the following properties of the FolderStructure object. - ProductionCodeFiles property - StubCodeFiles property - CodeGenerationReportFiles property For artifacts that are not CGU-specific, this macro expands to an empty string. In the resulting path, TargetLink replaces `//` with `/`.	-
`$(Compiler)`	Expands to the name of the compiler used to build the simulation application. Allowed only in the value of the DD ObjectFilesArchive property.	-
`$(Variant)`	Expands to the code variant that is active during code generation. If no code variant is active, it expands to an empty string. In the resulting path, TargetLink replaces `//` with `/`.	-

Macro expansion during AUTOSAR export

The following table shows the expansion of name macros during AUTOSAR export:

Macro	Replaced by	Example
`$(Component)`	Expands to the name of the software component that the following element kinds belong to: ▪ `ConstantSpecification_ADT` ▪ `ConstantSpecification_IDT` ▪ `ConstantSpecificationMappingSet` ▪ `PerInstanceMemoryArtifact`	-
`$(ComponentAutosarFileName)`	Expands to the file name of the DD SoftwareComponent object as defined via its GroupInfo.AutosarFileName property.	-
`$(ComponentPackage)`	Expands to the package of the DD SoftwareComponent object as defined via its GroupInfo.Package property.	-
`$(ElementContext)`	Expands to the name of the object that references an element of the following element kind: ▪ `ConstantSpecification_ADT` ▪ `ConstantSpecification_IDT` ▪ `DataConstraint` ▪ `PerInstanceMemoryArtifact` (AUTOSAR-specific)	For an ExportRule object with the ElementKind property set to `DataConstraint`, the NameTemplate property set to `DC_$(ElementContext)`, and an application data type named `MyADT_Scalar`, the macro expands to **`DC_MyADT_Scalar.`**
`$(ElementContextAutosarFileName)`	Expands to the file name of the object that references an element of the following element kind: ▪ `ConstantSpecification_ADT` ▪ `ConstantSpecification_IDT` ▪ `DataConstraint`	-
`$(ElementContextPackage)`	Expands to the package of the object that references an element of the following element kind: ▪ `ConstantSpecification_ADT` ▪ `ConstantSpecification_IDT` ▪ `DataConstraint`	-

Related topics

Basics

Basics on Using Name Macros..... 78

HowTos

How to Implement Instance-Specific Variables via Direct Reuse..... 308

References

Block-Related Name Macros (📖 TargetLink Model Element Reference)
Function Block (📖 TargetLink Model Element Reference)
Stateflow-Related Name Macros (📖 TargetLink Model Element Reference)

Working With Variable Classes

Where to go from here

Information in this section

Customizing the Appearance of Variables via Variable Classes..... 88

Examples of Working with Variable Classes..... 95

Customizing the Appearance of Variables via Variable Classes

Introduction

You can use variable classes to customize the appearance of variables in the generated production code.

Where to go from here

Information in this section

Basics on Variable Classes..... 88

Details on Preconfigured VariableClass Objects in the Data Dictionary..... 90

How to Define New Variable Classes..... 93

Basics on Variable Classes

Defining the appearance of variables

DD **VariableClass** objects provide all the properties you can use to define the appearance of variables in the generated production code, i.e., to instruct the Code Generator how to generate variables associated with the variable class. The names used for **VariableClass** objects must be valid C identifiers that consist of letters, digits and underscores. Via the context menu of the selected **VariableClass** object, you can apply various commands, for example, **Add custom property**.

Basics

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.

Note

A variable can be optimized if it can be removed from the source code without changing the simulation behavior of the model at its interface. To enable the Code Generator to eliminate a variable, its variable class must be set to **default** or the Optimization property of the selected class must be set. For TargetLink, the variable class name *default* is a keyword and must therefore not be used in the name of a variable class you create. In this context, *default* means that the Code Generator automatically selects the most efficient implementation.

Controlling the appearance of variables

It is essential to define the memory allocation of variables that cannot be eliminated. You generally have to control how a variable appears in the generated code because:

- The variable must be accessible to a calibration system.
- Variables from external user-supplied sources must be applied.
- The variable must be allocated to a module or a specific memory section.
- A variable must be initialized in a special RESTART function.
- One variable must be used by multiple blocks.
- Parameters must be passed via a function interface.
- The generated code must comply with company-specific coding guidelines for variable declarations.

TargetLink variable classes can be applied to meet these and related objectives.

Related topics

Basics

Details on Preconfigured VariableClass Objects in the Data Dictionary..... 90

HowTos

How to Define New Variable Classes..... 93

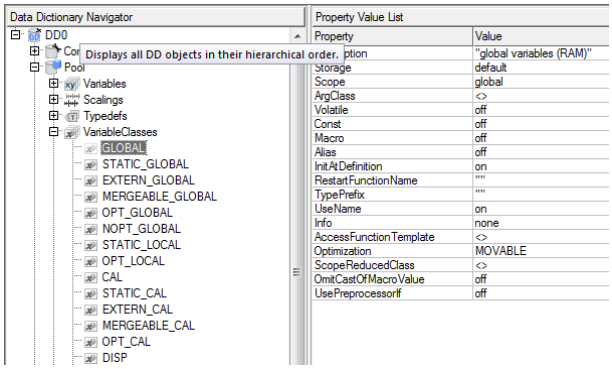
Examples

Example of Changing TargetLink Subsystem Interfaces..... 112
Example of Making Block Outputs Observable..... 96
Example of Making Parameters Calibratable..... 101
Example of Specifying Block Parameters as Input Arguments of Functions..... 114
Example of Specifying Parameters as Preprocessor Macros..... 117

Details on Preconfigured VariableClass Objects in the Data Dictionary

Introduction

Data dictionaries based on the DD templates that are provided by TargetLink contain various preconfigured variable classes.



Note

The preconfigured DD VariableClass objects are only examples that might not match your use cases.

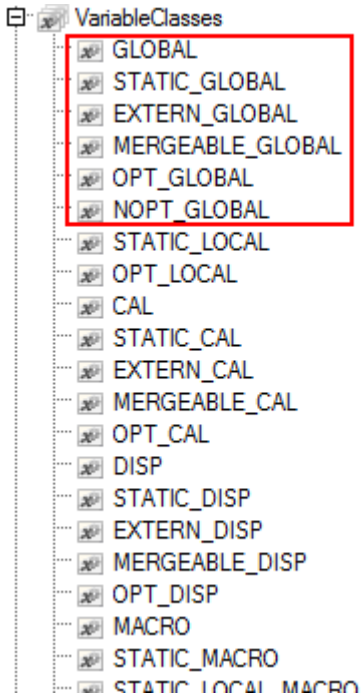
It is strongly recommended you create your own DD VariableClass objects that are suitable for your actual use cases. Duplicate (**CTRL + D**) any of the preconfigured objects as a starting point. Do not change the objects themselves.

However, preconfigured VariableClass objects that belong to the VariableClassGroup objects AUTOSAR or RealTimePlatformSupport must not be duplicated and must be used unchanged.

Each VariableClass object has a fixed set of properties that describe how the Code Generator must generate a variable associated with the class.

Categories of preconfigured variable classes

Variable classes differ with regard to their property values and can roughly be categorized according to their tasks. A category generally consists of several variable classes, each of which provides a specific property value that deviates from the variable class that heads the category. The following screenshot shows variable classes with global scope as an example:



To make the selection of variable classes in the TargetLink block dialogs easier, all preconfigured variable classes are named according to their tasks, which in this case is calibration. The following categories are available:

Variable Class Objects	Description	Typical Memory Location
GLOBAL	Normal global variables	RAM
CAL	Global calibratable parameters	ROM
DISP	Global observable variables	RAM
MACRO	Global preprocessor macros	–
CONST	Global constants	ROM
FCN_ARG	Function arguments (non-constant, call-by-value)	STACK
FCN_PARAM	Function parameters (constant, call-by-value)	STACK
FCN_RETURN	Function return value	STACK
EXTERN_GLOBAL_ALIAS	Global variables, which are declared in a custom header file	–
EXTERN_CAL_ALIAS	Calibratable global variables, that are declared in a custom header file	–
RTOS_ACCESS_FUNCTION	Apply an access function to read/write the variable	–

All variable classes of a category have specific properties in common that define the general behavior of the category.

Example All variable classes of the calibratable category have their Info set to `readwrite` and their Scope property set to `global`. This enables an external calibration tool to dynamically change (read/write) parameters.

Example2 The preconfigured DD VariableClass object called `DISP` has its Optimization property set to `MOVABLE`.

This is the case, because it is configured to be used for variables that are to be measured in *optimized code* at one of the following times:

- The calculation that includes the variable is performed
- The variable is used
- The variable is actualized

This might not meet your expectations. However, to determine which properties of a DD **VariableClass** to set, you first have to determine what you want to do and when. The following table shows exemplary considerations for measurable variables:

What	When
What do I want to measure? ▪ An input ▪ A preliminary result ▪ An output	When is the measurement to be made? 1. On each execution 2. Only in optimized code and only when it is used[1)] 3. Only when the variable is required or calculated[2)]

[1)] Assumed use case for **DISP**
[2)] Assumed use case for **OPT_DISP**

If you determined the what and the when, you should create a DD **VariableClass** that matches your needs.

Tip

You can use the Data Dictionary's compare functionality to visualize the properties in that they differ. Refer to Comparing and Merging Data Dictionaries (📖 TargetLink Data Dictionary Basic Concepts Guide).

Preconfigured derived variable classes

Other variable class properties, for example, *Optimization* and *Storage*, are used to define specific variants that describe the variable class with regard to access and optimization. For the MERGEABLE_CAL variable class, for example, the Optimization property is set to `MERGEABLE`. This instructs the Code Generator to generate, one variable representing parameters of multiple blocks in the generated code, if possible. The following derived variable classes are preconfigured:

Variable Class Objects	Description
STATIC_xxx	Static variables are accessible only in the file in which they are defined.
EXTERN_xxx	External variables are only declared (not defined) in a file so that you can use them in that file. They are defined in another file.
MERGEABLE_xxx	Mergeable variables are merged during code generation, if possible. This means multiple definitions of a variable are identified by their name and mapped to one variable. They can be referenced from multiple blocks in the model.
OPT_xxx	Global variables that are fully optimizable, i.e., if possible they are removed.
NOPT_xxx	Global variables which are not optimizable.

Variable Class Objects	Description
STATIC_LOCAL_xxx	Static local variables are accessible only in the function they are defined in (function-local). Unlike normal local variables, they are not placed on the stack but in the RAM or ROM.
FCN_REF_ARG	Function arguments (non-constant, call-by-reference)
FCN_REF_PARAM	Function parameters (constant, call-by-reference)

Related topics

Basics

Basics on Variable Classes..... 88

HowTos

How to Define New Variable Classes..... 93

How to Define New Variable Classes

Objective

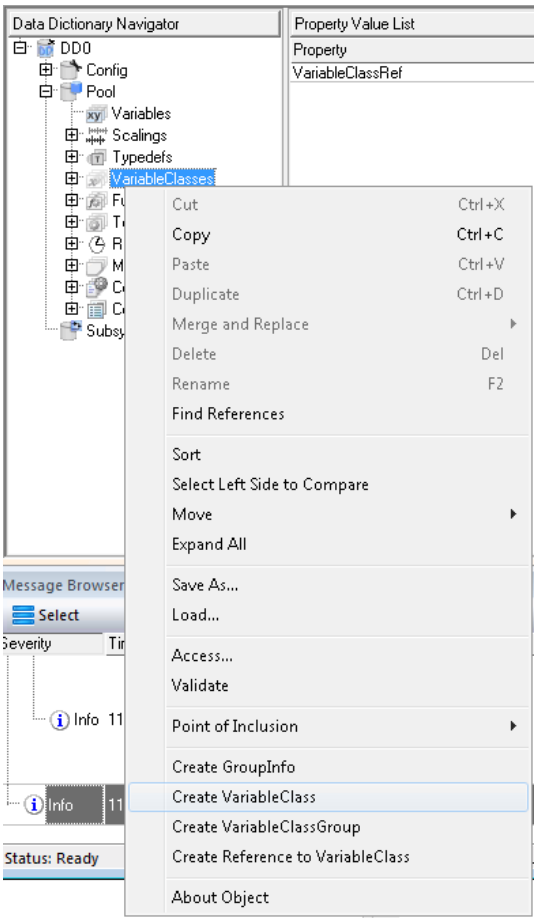
If the preconfigured variable classes do not match your requirements, you can create a new variable class that is tailored to your requirements.

Method

To define a new variable class

1 In the `/Pool` area of the Data Dictionary Navigator, right-click VariableClasses.

2 From the context menu, select **Create VariableClass**.



A new variable class object named **new_VariableClass** is created below the preconfigured VariableClass objects. All variable class properties are set to default values.

Note

You can create a VariableClassGroup object if you want to keep similar variable classes in one subtree, for example, all calibratable variable classes.

3 Enter a unique name for the new VariableClass object.

The variable class names must be valid C identifiers, which consist of letters, digits, and underscores. It is useful to append the name of the category so that you will always get unique and identifiable variable class names.

4 In the property value list, set the property values according to your requirements.

5 Right-click the VariableClass object you created.

6 From the context menu, select **Validate**.

The validity of the selected variable class is checked.

Result

You created a new VariableClass object, which can be referenced in a block dialog. For classes that are organized in subgroups, the DD path must be specified relative to `/Pool/VariableClasses`, which means it includes the group name, for example, **Calibratables/STATIC_CAL**.

Related topics

Basics

Basics on Variable Classes..... 88
Details on Preconfigured VariableClass Objects in the Data Dictionary..... 90

Examples of Working with Variable Classes

Follow up examples

Some of the following examples are based on previous examples.

Where to go from here

Information in this section

Example of Making Block Outputs Observable..... 96
Example of Collecting Variables in Separate Modules..... 99
Example of Making Parameters Calibratable..... 101
Example of Changing Parameter Values on the ECU..... 104
Example of Including Externally Stored Parameter Declarations..... 109
Example of Changing TargetLink Subsystem Interfaces..... 112
Example of Specifying Block Parameters as Input Arguments of Functions..... 114
Example of Specifying Parameters as Preprocessor Macros..... 117
Example of Initializing Variables via Restart Functions..... 120
Example of Specifying Type Qualifiers via Type Prefixes..... 122
Example of Merging Variables..... 123
Example of Defining Variable Classes for Alias Variables..... 125

Example of Making Block Outputs Observable

Creating global variables

Suppose you want to make a block output observable, for example, because you want to read its current value via a calibration tool. To do so, you have to create a global variable for it.

Precondition

This example makes use of the preconfigured DD VariableClass object called `DISP`.

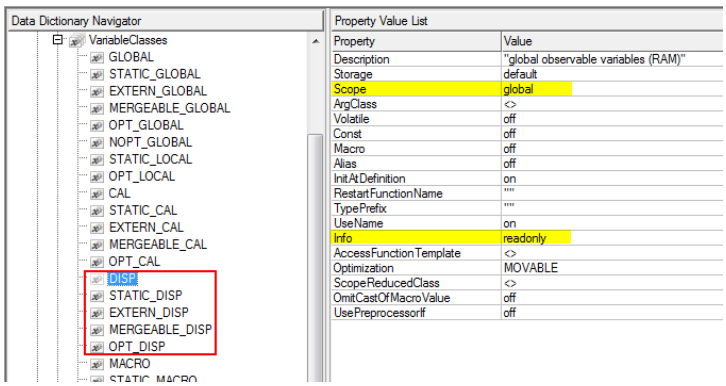
> **Note**
>
> The preconfigured DD VariableClass objects are only examples that might not match your use cases.
>
> It is strongly recommended you create your own DD VariableClass objects that are suitable for your actual use cases. Duplicate (**CTRL + D**) any of the preconfigured objects as a starting point. Do not change the objects themselves.
>
> However, preconfigured VariableClass objects that belong to the VariableClassGroup objects AUTOSAR or RealTimePlatformSupport must not be duplicated and must be used unchanged.

For details, refer to Example2 on page 91.

DISP variable class

This is possible if you assign a variable class to the corresponding output, where the Scope property is global and the Info property is read-only.

The variable classes highlighted in the illustration below meet these requirements.



The read-only property value does not influence the generated C code but lets the variable appear as a MEASUREMENT element in the description file for the calibration tool (for example, in an ASAM MCD-2 MC file).

Information property	ASAM MCD-2 MC File	Predefined Class	Remark
`none`	no entry	GLOBAL	The variable will not appear in A2L files exported from TargetLink. It will therefore not be accessible in a calibration and measurement system later on.
`readonly`	MEASUREMENT	DISP	This specification is typically used for measurable signals, i.e., variables that are available for read access in a calibration and measurement system. The variables are marked as *MEASUREMENTS* in an A2L file. Access must be possible via a fixed address. Therefore, the scope is reduced only to module-local.
`readwrite`	CHARACTERISTIC	CAL	This specification is typically used for calibration parameters, i.e., variables that have to be available for read and write access in a calibration and measurement system. Variables with read/write access appear as *CHARACTERISTICS* in A2L files. Access must be possible via a fixed address. Therefore, the scope is reduced only to module-local. Calibration engineers often modify calibration parameters so that the control algorithm is ideally tailored to the system. This means that variable values can change outside the production code. Therefore, TargetLink evaluates only specified value ranges. Code optimization ensures that their values are not propagated.
`adjustable`	no entry	n.a.	Capable of changing values between (writing and reading) accesses, i.e., TargetLink assumes that variable values can change outside the production code. Therefore, TargetLink evaluates only specified value ranges. The variable will not appear in the A2L file. Code optimization ensures that their values will not be propagated.
`offline_readwrite`	CHARACTERISTIC	n.a.	The variable is not part of the A2L file. Therefore, variables can be flashed but not calibrated during ECU run time (e.g., for safety-relevant parameters that must not be changed while driving). Access must be possible via a fixed address. Therefore, the scope is reduced only to module-local.

Note

In the generated production code, block outputs can be merged into the associated operation (i.e., eliminated) and are therefore not represented by variables.

```
...
/* Sum: picontroller/sU
 # combined # Gain: picontroller/Kp
 # combined # Gain: picontroller/Ki */
 Sa1_sU = (Int16) (((UInt16) (Int16) ((((Int32) Sa1_e) *
  ((Int32) P_Sa1_Kp)) >> 7)) + ((UInt16)(Int16) ((((Int32) Sa1_x)
  * ((Int32) P_Sa1_Ki)) >> 6)));
...
```

This piece of code represents the blocks sU, Kp, and Ki. There are no variables for the outputs of the Ki and Kp **Gain** blocks, which reduces stack consumption and execution time.

Model location

The following instructions show you how to apply the DISP variable class for the output of the Kp Gain block in the PIPT1 demo model. The model is located in `<DocumentsFolder>\dSPACE\TargetLink\<Version>\Demos\pipt1`.

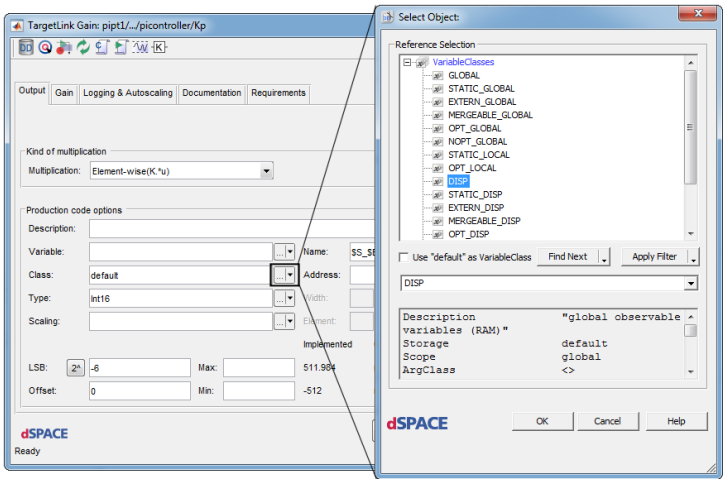
Method

To make a block output observable

1 Open the PIPT1 demo model and then the associated DD project file.

2 In the picontroller subsystem, double-click the desired block, for example, the Kp Gain block, to open the block dialog.

3 On the Output page, click the Class Browse button to open the Select Object dialog.



4 Select the DISP VariableClass object and click OK.

5 In the Name edit field, set the name of the variable to be generated to *y_Kp* or any other valid C identifier if you want to overwrite TargetLink's default naming convention.

6 Click the Apply button to confirm your settings and close the dialog.

7 Change to the TargetLink Main Dialog and generate production code.

Result

The corresponding code fragment in `picontroller.c` now reads:

```
/* Gain: picontroller/Kp */
y_Kp = (Int16) ((((Int32)Sa1_e) * ((Int32)P_Sa1_Kp)) >> 7);
```

A variable that represents the block output of Kp is explicitly evaluated in each computation step. The corresponding variable declaration can be found in the declaration part of this file and reads:

```
DISP Int16  y_Kp;
/* LSB: 2^-6 OFF:  0 MIN/MAX:  -512 .. 511.984375 */;
```

- As the UseName property of the DISP VariableClass object is set to on, the DISP macro is used in the generated production code. Its macro definition can be found in the `tl_defines_<id>.h` header file (`#define DISP`), which is

included in the `picontroller.h` production code header file (`#include "tl_defines_a.h"`):

The contents of the `description` property of the variable class are inserted as a code comment.

- `/Subsystems/picontroller/picontroller/Variables/` in the TargetLink Data Dictionary contains the `y_Kp` variable. Its **Class** property identifies it as being read-only for a calibration tool. The **TargetLink Data Dictionary**'s ASAM MCD-2 MC export interface takes this into account and also retrieves the address of `y_Kp` from the associated linker map file to determine the list of MEASUREMENTS.

  For details on the TargetLink Data Dictionary's ASAM MCD-2 MC export interface, refer to Preparing the A2L Export (📖 TargetLink Interoperation and Exchange Guide).

Next step

You can generate a separate code file for `y_Kp`, for example, if you want to have all calibratable variables in a separate file. For details, refer to Example of Collecting Variables in Separate Modules on page 99.

Related topics

Basics

Basics on Variable Classes..... 88
Preparing the A2L Export (📖 TargetLink Interoperation and Exchange Guide)

Examples

Example of Collecting Variables in Separate Modules..... 99

Example of Collecting Variables in Separate Modules

Generating separate modules

You can instruct the Code Generator to create a separate module (code file) if you want to keep all the variables of one class in a separate file.

Options for assigning a variable to a module

In TargetLink, you have several options for assigning a variable to a module:

- You can assign a single variable to one module.
- You can assign a whole group of variables to one module.

- You can assign all the variables of a function and the function body to one module.

Note

Static variables are visible in only one module. They cannot be separated from the module containing the function body.

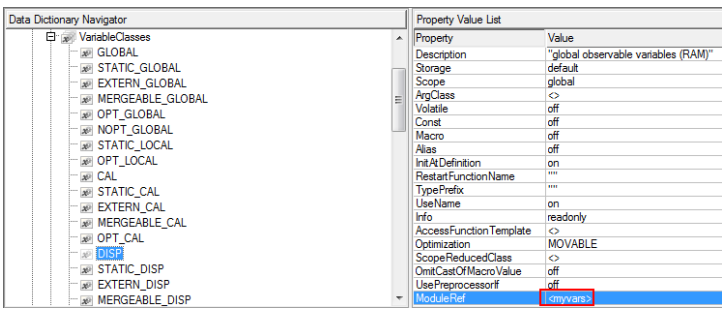
Precondition

- You must have followed the Example of Making Block Outputs Observable on page 96.
- You must have created a DD module object, for example, myvars.

Method

To collect variables in a separate module

1. Open the PIPT1 demo model and then the associated DD project file in the Data Dictionary Manager.
2. Select the `/Pool/VariableClasses/DISP` object.
3. In the property value list, click the Browse button of the ModuleRef value and select a DD module object, for example, *`myvars`*, to modify the specification of the DISP variable class.



4. Change to the TargetLink Main Dialog and generate production code.

Result

The Code Generator generates the additional source modules `myvars.c` and `myvars.h`. The `myvars.c` file contains the definition of *`y_Kp`*:

```
DISP Int16  y_Kp;
/* LSB: 2^-6 OFF:  0 MIN/MAX: -512 .. 511.984375 */
```

Related topics

Basics

Basics on Variable Classes..... 88

Examples

Example of Making Block Outputs Observable..... 96

Example of Making Parameters Calibratable

Creating read-/writable global variables

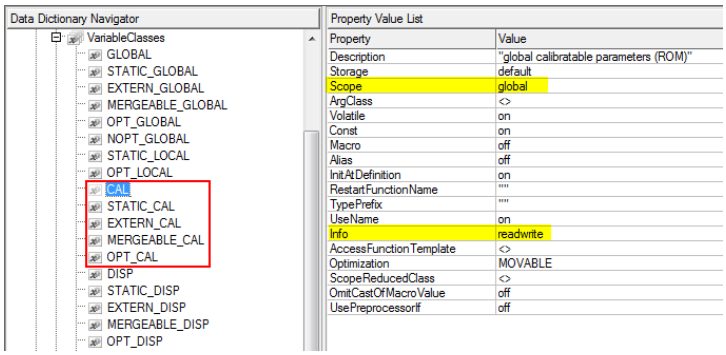
Suppose you want to make a variable calibratable, for example, because you want to change the parameter via a calibration system. To do so, you have to create a global variable that can be read and written by the calibration system.

Basics

Calibration systems access variables directly by their addresses in the ECU memory. Calibratable variables must have global scope, otherwise their addresses cannot be determined from the linker map file. For the calibration system, it is not sufficient to know only the name and the address of a variable. It also requires the data type and scaling information.

CAL variable class

Making parameters calibratable is possible if you assign a variable class to the parameter where the Scope property is **global** and the Info property is **readwrite**. The variable classes highlighted in the illustration below meet these requirements.



In TargetLink, the Info property of a variable class determines if a variable is accessible by a calibration system or not. If you select a variable class with `Info = readwrite` (for example, the predefined class CAL), you instruct the Code Generator to prepare variables associated with the variable class for a calibration system. The variables are written to the ASAM MCD-2 MC file.

Note

For most target compilers, calibratable variables must have global scope, otherwise their addresses in the ECU memory cannot be determined.

Demo location

The following example shows you how to apply the CAL variable class to make a parameter calibratable. The example uses the PIPT1 demo model, which is located in `<DocumentsFolder>\dSPACE\TargetLink\<Version>\Demos\pipt1`.

Method

To make a parameter calibratable

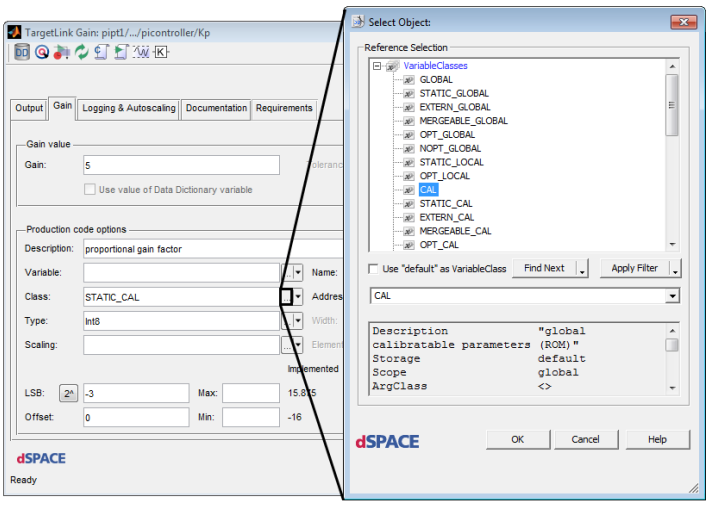
1 Open the PIPT1 demo model and then the associated DD project file.

2 In the picontroller subsystem, double-click the desired block, for example, the KpGain block, to open the block dialog.

Tip

By default, the gain value of the Kp block does not occur as a separate variable in the generated code, but is eliminated, i.e. inlined in the associated operation.

3 On the Gain page, click the Class Browse button to open the Select Object dialog.



4 Select the CAL variable class and click OK.

The scaling controls are enabled to allow variable scaling.

5 In the Name edit field, set the name of the variable to be generated to *Kp* or any other valid C identifier if you want to overwrite TargetLink's default naming convention.

6 Scale the gain parameter according to the range it has to cover or click the **Scaling** Browse button to select a scaling object via the **Select Object** dialog.

7 Click the **Apply** button to confirm your settings and close the dialog.

8 Open the **TargetLink Main Dialog** and start the code generation process.

Result

The corresponding code fragment in `picontroller.c` now reads:

```
...
 # combined # Gain: picontroller/Kp
 # combined # Gain: picontroller/Ki */
   Sa1_sU = (Int16) (((UInt16) (Int16) ((((Int32) Sa1_e) *
     ((Int32) Kp)) >> 7)) + ((UInt16)
     (Int16) ((((Int32) Sa1_x) * ((Int32) P_Sa1_Ki)) >> 6)));
     LOG_VAR(a, _Sa1_sU, Sa1_sU);
...
```

The corresponding variable declaration can be found in the declaration part of this file and reads:

```
CAL Int8 Kp = 40 /*5.*/ /*
   Description: proportional gain factor
   LSB: 2^-3 OFF:  0 MIN/MAX:  -16 .. 15.875 */;
```

- As the UseName property of the CAL variable class is set to **on**, the CAL macro is used in the generated production code. Its macro definition can be found in the header file `tl_defines_<id>.h`, which is included in the production code header file `picontroller.h`:

  ```
  ...
  #define CAL const volatile
  #define STATIC_CAL static const volatile
  ...
  ```

  The contents of the description property of the variable class are inserted as a code comment.

- `/Subsystems/picontroller/picontroller/Variables/` in the **TargetLink Data Dictionary** contains the *Kp* variable. Via its **Class** property, it is implicitly identifiable as a `readwrite` variable. The **TargetLink Data Dictionary**'s ASAM MCD-2 MC export interface takes this into account and also retrieves the address of *Kp* from the associated linker map file.

  For details on the **TargetLink Data Dictionary**'s ASAM MCD-2 MC export interface, refer to Preparing the A2L Export (📖 TargetLink Interoperation and Exchange Guide).

Next step

You can include *Kp* from a separate source code file. For details, refer to Example of Including Externally Stored Parameter Declarations on page 109.

Related topics

Basics

Basics on Variable Classes..... 88
Preparing the A2L Export (TargetLink Interoperation and Exchange Guide)

Examples

Example of Including Externally Stored Parameter Declarations..... 109

Example of Changing Parameter Values on the ECU

Accessing calibratable parameters on the ECU

Suppose you want to change, i.e., calibrate, parameter values on your electronic control unit (ECU) via a calibration system. This requires access to the parameters to be calibrated. To access parameters, you have to assign a variable class where the Scope property is **global** and the Info property is **readwrite**.

Basics

When you have defined calibratable variables, you can force TargetLink to collect them during code generation and to place them in a structure variable (struct). TargetLink supports multiple instances of structs (data pages) required by a calibration system such as ControlDesk, in a form that is independent of target hardware features, for example, memory overlay.

Only one page is visible to the ECU at a time. This means when you change parameters on one page, you can make these changes available to the ECU via a single page switch.

Switching parameter sets

You can initiate the generation of data pages by adding a predefined option to the hook script derived from the **`tl_pre_codegen_hook`** (TargetLink File Reference) hook script template. You can use this option to specify whether to generate data pages and the number of data pages to be generated. The content of all data pages is identical until you change parameter values. The first data page is the reference page, which is used as the default page.

Use TargetLink's `tlCustomizationFiles('Create',...)` API function or the **Create Customization Files** dialog to derive customization files from their templates. You can open the dialog by executing the `tlCustomizationFiles` command without arguments. Refer to Deriving Customization Files From Their Templates on page 33.

To switch pages, a switch function, a switch variable, and an active page pointer are also generated. The switch function is defined as *global* and can be called externally, for example, via the XCP service in the service interface layer. The XCP service provides the `..._set_cal_page` function, which you can use

to change the switch variable's value and thus the destination of the active page pointer. You can also use it to insert the function for the switch function.

Considered variables

If a variable was moved to a data page, all occurrences of the variable are replaced by accessing this data page. Access is achieved by referencing the active page pointer.

Variables are not moved to data pages if they are:

- Address-mapped
- Macros
- 64-bit variables
- Static local
- Extern
- Extern macros
- Variables using a variable class whose property alias is set to 'on'
- Structs (and components of structs) not representing a look-up table map

Note

When used with data pages, structs that represent look-up tables lead to structs of structs in the generated code. This is error-prone and might cause problems when calibrating data and switching pages. It is therefore recommended to clear the **Generate map structure** option to generate code for look-up tables without map structures when working with data pages.

Sometimes it is useful to control the generation of data pages, for example, if data pages have to be placed in different files to assign them to different memory sections. If so, you can use the *Variables* template, the *Functions* template, or the *Structs* templates provided by the **TargetLink Data Dictionary**.

Note

If you want to use data pages in combination with code generation from the Data Dictionary, all information of the data pages has to be stored in the TargetLink Data Dictionary. This means that:

- All variables that become part of the data page must be specified in the Data Dictionary and must reference a DDBased module. All variables means variables in the model and in the Data Dictionary with exception of the variables that fulfill one of the criteria mentioned above, for example, address-mapped variables.
- Variables and functions must have fixed names independent of model elements or DD objects, i.e., do not use CGU-dependent name macros, such as `$I`.
- Function, Variable and Struct templates (see below) must exist in order to generate the elements in a DDBased module. Function templates allow to directly reference a module via the ModuleRef property in the Settings. The Variable and Struct templates have to reference a variable class that again references a DDBased module.

To influence the generation of Data Pages, Active Page Pointer, Switch Variable, and Switch Function, you first have to set their filter criteria:

Variable templates for page switching:

Filter Criteria		
VariableKind	**VariableClassSpec**	**Description**
ActivePagePointer	SLGlobal	Template to influence the generation of the active page pointer
PageSwitchVariable	SLGlobal	Template to influence the generation of the switch variable

Struct templates for page switching:

Filter Criteria	
StructSpec	**Description**
DataPage0, DataPage1, etc. or DataPage* (for all pages)	Template to influence the generation of data pages

Function templates for page switching:

Filter Criteria	
FunctionName	**Description**
PageSwitchFunction	Template to influence the generation of the Switch Function

For details on filter criteria, refer to Data Dictionary-Related Name Macros.

Precondition

Your model contains calibratable variables.

Method

To change parameter values on the ECU

1 In the MATLAB Command Window, type the following command:

```
tlCustomizationFiles
```

TargetLink derives the `tl_pre_codegen_hook.m` hook script from its template and opens it in the editor.

2 Add the predefined Code Generator variable `tlcgOptions` as shown below.

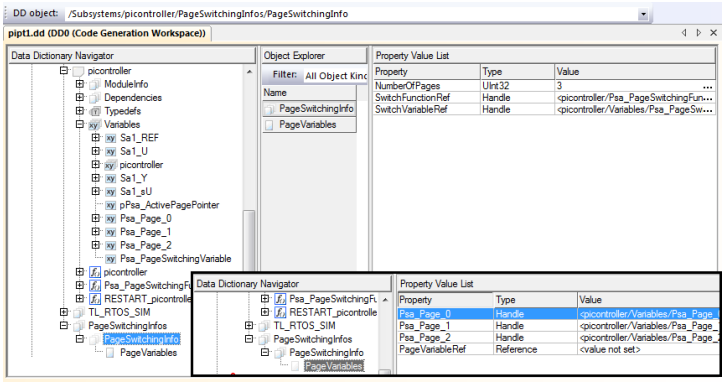
```
tlcgOptions = { …
… tlcgOptions{:} …
……'EnableDataPages','on', …
……'NumberOfDataPages', 3 };
```

You can specify up to 256 data pages. A valid number is a positive integer. The default value is 2. If so, you can omit the `NumberOfDataPages` entry.

3 Save and close the `tl_pre_codegen_hook.m` file.

4 Open the TargetLink Main Dialog of your model and generate production code.

As a result, the specified number of data pages that contain calibratable variables is created. If you want to check the results, you can do so via the Data Dictionary Navigator.



5 On the Tools page of the TargetLink Main Dialog, click Generate ASAP2 file to generate an calibration information file (`*.a2l`).

For each calibratable variable listed in the TargetLink Data Dictionary, corresponding ASAP2 information similar to the following is generated automatically.

```
…
/begin CHARACTERISTIC
   P_Sa1_Ki   /* Name */
   ""   /* LongIdentifier */
   VALUE   /* Type */
   0x3f981d   /* Address */
   SBYTE_COL_DIRECT   /* Deposit */
   0   /* MaxDiff */
   EQ_LSB_0_015625   /* Conversion */
   -2   /* LowerLimit */
   1.98438   /* UpperLimit */
   BYTE_ORDER MSB_FIRST
   /begin IF_DATA ASAP1B_CCP
      DP_BLOB 0x0 0x3f981d 1
   /end IF_DATA
/end CHARACTERISTIC
…

/begin CHARACTERISTIC
   P_Sa1_Kp   /* Name */
   ""   /* LongIdentifier */
   VALUE   /* Type */
   0x3f981e   /* Address */
   SBYTE_COL_DIRECT   /* Deposit */
   0   /* MaxDiff */
   EQ_LSB_0_125   /* Conversion */
   -16   /* LowerLimit */
   15.875   /* UpperLimit */
   BYTE_ORDER MSB_FIRST
   /begin IF_DATA ASAP1B_CCP
      DP_BLOB 0x0 0x3f981e 1
   /end IF_DATA
/end CHARACTERISTIC
…
```

Result

You have carried out all preparation steps for page switching. You must now integrate the results in your calibration service.

Related topics

Basics

Basics on Variable Classes..... 88
Deriving Customization Files From Their Templates..... 33

Examples

Example of Making Parameters Calibratable..... 101

References

Create Customization Files Dialog Description (📖 TargetLink Tool and Utility Reference)
EnableDataPages (📖 TargetLink Model Element Reference)
Hook Scripts (📖 TargetLink File Reference)
NumberOfDataPages (📖 TargetLink Model Element Reference)
tlCustomizationFiles (📖 TargetLink API Reference)

Example of Including Externally Stored Parameter Declarations

Including externally declared parameters

You can instruct the Code Generator to include parameters that are declared in a separate C file. If you include your own source files with parameter declarations, TargetLink declares the corresponding variables as external symbols and uses them as they are.

Basics

Often, the code generated by TargetLink is integrated into an existing ECU software environment. The environment already contains numerous variables, which must be declared as external symbols in the production code generated by TargetLink.

Note

Simulation with your own variables and functions is possible only if you link the corresponding code files to the generated simulation application, for example, with the TargetLink **Addfile** block. The initial values of the parameters in your code file must be identical to the values you specified in the TargetLink blocks.

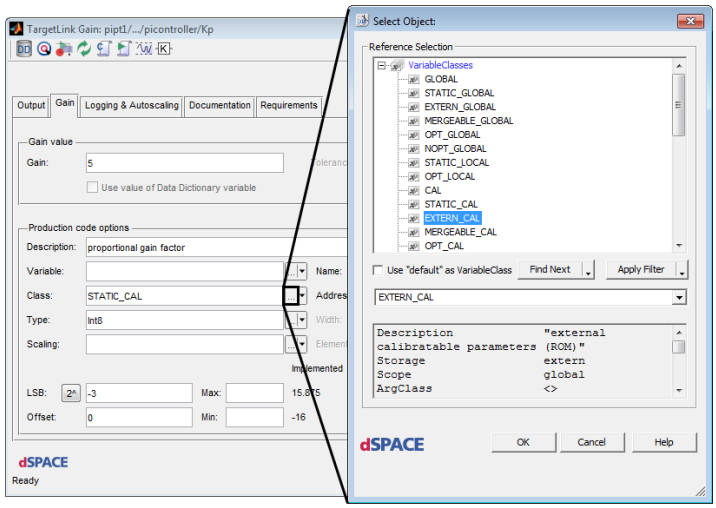
Method

To include externally stored parameter declarations

1 Open the PIPT1 demo model and then the associated DD project file.

2 In the picontroller subsystem, double-click the desired block, for example, the Kp Gain block, to open the block dialog.

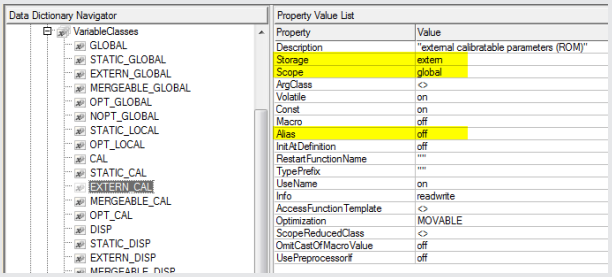
3 On the Gain page, click the Class Browse button to open the Select Object dialog.



4 Select the EXTERN_CAL variable class and click OK.

Tip

By default, this variable class is specified as shown below, i.e., with Storage set to extern and Scope set to global. Alternatively, you can set the new variable class property Alias to on. TargetLink does even not generate an external declaration for Alias variables. Instead, the variable declaration is imported from a header file you specify with the Addfile block. For details, refer to Example of Defining Variable Classes for Alias Variables on page 125.

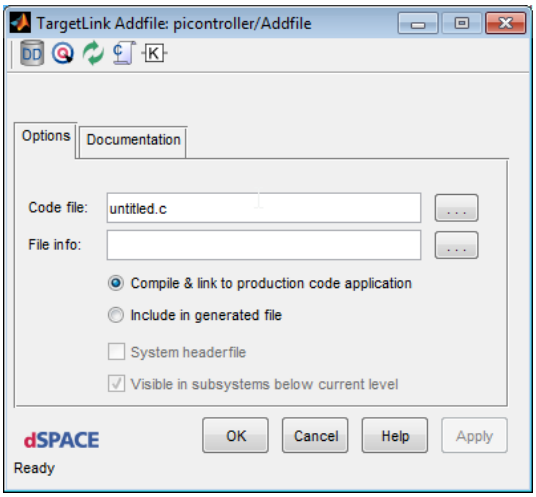


5 Use an editor to create a new C file named `extvars.c` containing the variable declaration of *Kp*.

The initial value, data type, and properties, such as Const and Volatile, must be identical to what you specified in the block dialog.

6 From the TargetLink block library, drag & drop an **Addfile** block to the TargetLink subsystem **picontroller**.

7 Open the block dialog and set the block parameters.



You have to decide whether you want to include the C code of the referenced file or simply compile and link it. The code of the `extvars.c` file you provide reads, for example:

```
#include "tl_basetypes.h"
UInt8 Kp = 160 /* 5. */;
```

Because the `UInt8` data type must be defined to make the file compilable, you have to include `tl_basetypes.h`. During code generation, a warning message is displayed in the MATLAB Command Window, reminding you to provide the *Kp* variable manually.

8 Open the **TargetLink Main Dialog** and start the code generation process.

Result

The parameters you provided via the `extvars.c` file are compiled and linked to the production code S-function for SIL simulation, and to the target executable for PIL simulation.

Related topics

Basics

Basics on Variable Classes 88

Examples

Example of Defining Variable Classes for Alias Variables 125
Example of Making Parameters Calibratable 101

Example of Changing TargetLink Subsystem Interfaces

Default behavior for inputs and outputs

In TargetLink, the function interface, i.e., the *inputs* and *outputs* of a TargetLink subsystem, which is specified as a function, is generated via global variables if the **default** variable class is selected. You then get global variables for TargetLink inports or outports.

Variable classes GLOBAL and EXTERN_GLOBAL

Generally, your model consists of several subsystems that are interconnected. For example, one output signal can be the input signal for an arbitrary number of subsystems. If so, it is important to clearly specify the function interface to reduce the number of variables created.

You can do this by using the GLOBAL variable class for TargetLink outports and EXTERN_GLOBAL for TargetLink inports, and also by using fixed variable names instead of name macros.

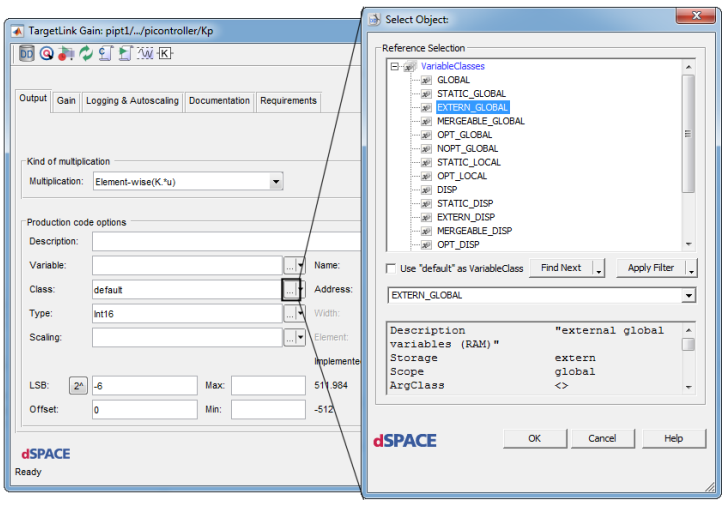
Demo location

The following example shows you how to apply these variable classes to the respective TargetLink inports and outports using the PIPT1 demo model. The model is located in `<DocumentsFolder>\dSPACE\TargetLink\<Version>\Demos\pipt1`.

Method

To specify interface variables to be generated as global

1. Open the PIPT1 demo model and then the associated DD project file.
2. In the **picontroller** subsystem, double-click the desired block, for example, the **REF InPort** block, to open the block dialog.
3. On the **Output** page, click the **Class** Browse button to open the **Select Object** dialog.



4. Select the **EXTERN_GLOBAL** VariableClass object and click **OK**.

5 In the **Name** edit field, set the name of the variable to be generated to *Ref_In* or any other valid C identifier to overwrite TargetLink's default naming convention.

6 Click the **Apply** button to confirm your settings and close the dialog.

7 Set the **EXTERN_GLOBAL** variable classes and names of all other TargetLink **InPort** blocks in your model and then set the **GLOBAL** variable class and names for the TargetLink **OutPort** blocks accordingly.

8 Change to the **TargetLink Main Dialog** and generate production code.

Result

The function prototype of picontroller in `picontroller.h` now reads:

```
…
#ifndef _PICONTROLLER_H_
#define _PICONTROLLER_H_
…
#include "tl_defines_a.h"
#include "tl_basetypes.h"
…
EXTERN_GLOBAL UInt16 Ref_In
   /* LSB: 2^-11 OFF:  0 MIN/MAX:  0 .. 31.99951171875 */;
EXTERN_GLOBAL UInt16 Y_In
   /* LSB: 2^-11 OFF:  0 MIN/MAX:  0 .. 31.99951171875 */;
…
extern GLOBAL Int16 U_Out
   /* LSB: 2^-6 OFF:  0 MIN/MAX:  -512 .. 511.984375 */;
…
extern void picontroller(void);
…
#endif/*_PICONTROLLER_H_ */
…
```

Because the input and output variables are either `global` or `extern_global`, picontroller is now a function without parameters. The variable declaration of `U_Out` can be found in the declaration part of `picontroller.c`:

```
…
GLOBAL Int16 U_Out
   /* LSB: 2^-6 OFF:  0 MIN/MAX:  -512 .. 511.984375 */;
…
STATIC_CAL Int8 P_Sa1_Ki = 26 /*0.4*/ /*
   Description: integral gain factor
   LSB: 2^-6 OFF:  0 MIN/MAX:  -2 .. 1.984375 */;
…
```

There is no declaration for the extern variables `Ref_In` and `Y_in` in `picontroller.c`.

Note

If the storage property of the variable class used for the inports and outports is set to static, production code is generated but cannot be compiled and linked to the TargetLink simulation frame. It therefore cannot be tested in the TargetLink environment.

Related topics

Basics

Basics on Function Interfaces 241
Basics on Variable Classes 88
Optimizing the Production Code 497

HowTos

How to Specify Function Interfaces via Function Parameters 242

References

Function Block (📖 TargetLink Model Element Reference)

Example of Specifying Block Parameters as Input Arguments of Functions

Specifying input arguments of functions

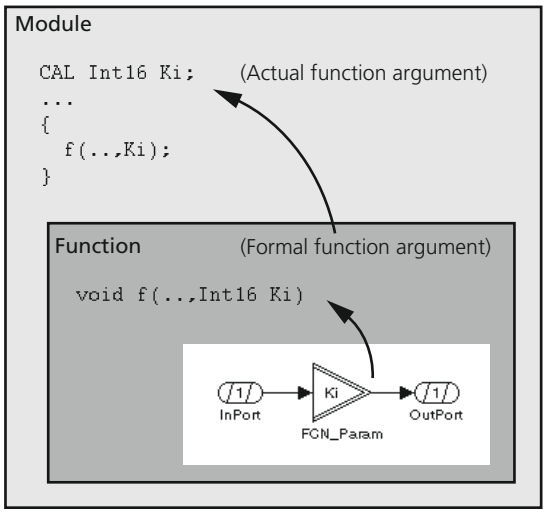
Suppose you want to specify the gain parameter of the Ki block as a parameter of the `controller` function. To do so, you have to specify it as a function parameter.

Basics

A variable associated with the FCN_PARAM variable class is specified as an input argument of the generated function. The kind of the actual function argument that is created in the calling function depends on the **ArgClass** property you select.

By default, **ArgClass** specifies the name of the variable class that is used to create the actual function argument in the calling function. If you use the default setting **ArgClass = CAL**, the actual function argument becomes a calibratable global variable in the module of the calling function as shown in the following

illustration. For details on specifying the name of the actual function argument, refer to Basics on Using Name Macros on page 78.



FCN_PARAM variable class

You can specify input arguments of functions by assigning a variable class where Scope is set to value_param and ArgClass is set to CAL.

Specifying function interfaces

TargetLink provides several variable classes you can use to specify the function interface. The following table gives an overview of the variable classes you can use for this purpose:

Variable Class	Formal Argument (Inside the Function)	Actual Argument (in the Calling Function)	Example
FCN_PARAM	call by value	CAL - calibratable global	f(calVar)
FCN_REF_PARAM	call by reference	CAL - calibratable global	f(&calVar)
FCN_ARG	call by value	default - local variable	f(var)
FCN_REF_ARG	call by reference	default - local variable	f(&var)
FCN_RETURN	return value	default - local variable	var=f()

Demo location

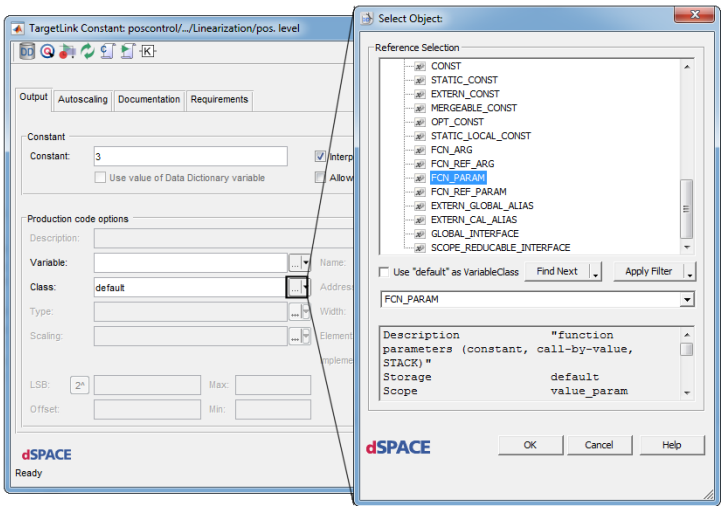
The following instructions show you how to apply the FCN_PARAM variable class to specify a block parameter as a function parameter. This is shown using the POSCONTROL demo model, which is located in `<DocumentsFolder>\dSPACE\TargetLink\<Version>\Demos\poscontrol`.

Method

To specify a block parameter as an input argument of a function

1. Open the POSCONTROL demo model and then the associated DD project file.
2. In the controller/Linearization subsystem, double-click the desired block, for example, the pos. level constant block, to open the block dialog.

3 On the **Output** page, click the **Class** browse button to open the **Select Object** dialog.



4 Select the **FCN_PARAM** variable class and click **OK**.

Tip

If you want to have a call-by-reference parameter or the function return value, you can select the **FCN_REF_PARAM** (Scope = ref_param) or the **FCN_RETURN** (Scope = fcn_return) variable class for the gain parameter.

5 Scale the gain parameter according to the range it is to cover.

The current value is 3, so it is a good choice to select **UInt16** for the data type and 2^{-13} for the LSB value. If you enter constrained limits (**Max**, **Min**), you guarantee that the range is not exceeded when calibrating the parameter.

6 In the **Name** edit field, set the name of the variable to be generated to *pos_level* or any other valid C identifier if you want to overwrite TargetLink's default naming convention.

7 Click the **Apply** button to confirm your settings and close the dialog.

8 Change to the **TargetLink Main Dialog** and generate production code.

Result

The implementation of the Linearization function in `controller.c` now reads:

```
CAL UInt16 pos_level = 24576 /*3.*/
   /* LSB: 2^-13 OFF:  0 MIN/MAX:  0 .. 7.9998779296875 */;
…
static void Sa2_Linearization(const UInt16 pos_level_a)
{
…
}
…
void controller(void)
{
…
   Sa2_Linearization(pos_level);
…
}
```

The `pos_level` variable is generated as a call-by-value parameter. If you have selected FCN_REF_PARAM in step 4, it is generated as a *call-by-reference* parameter. In this case, the function prototype of `Linearization` in `controller.h` reads:

```
…
extern void Linearization(Int16 * Sa1_UPI, const UInt16 * pos_level);
…
```

Note

For non-scalar (vector/2-D matrix) variables, a ***ref_param*** variable class results in the same code as a ***value_param*** class.

Related topics

Basics

Basics on Function Interfaces 241
Basics on Using Name Macros 78
Basics on Variable Classes 88

References

Function Block (📖 TargetLink Model Element Reference)

Example of Specifying Parameters as Preprocessor Macros

Usage of preprocessor macros

Suppose you want to improve the readability of the code by using a symbolic name for a constant value, e.g., PI instead of 3.14. In handwritten C code, symbolic names are often realized via preprocessor macros.

MACRO variable class

In TargetLink, preprocessor commands (C macros) are generated, if you use variable classes whose Macro property is set to on.

All variable classes where the Macro property is set to on are not permissible for:

- Block outputs, states and all variables that can be modified during run time.
- Table data, state-space matrices and transfer function or filter coefficients, since this could result in a large number of macro definitions for each coefficient.

For the EXTERN_MACRO variable class the Storage property is set to `extern`, which is why this class can be used for all kinds of variables, block parameters, outputs, and states.

Note

- TargetLink does not create these EXTERN_MACROs itself but applies them in the generated code only when associated variables are accessed. Use the Addfile block to provide the actual macro definition in a header file that is included in the model.

  You can create your own macros for general read/write access purposes with the following mechanism:

  ```
  #define  GET_X    get_var(strct.x)
  #define  Z_VALUE    myStruct.subStruct.z
  ```
- Suppose a variable is assigned the EXTERN_MACRO variable class and you want to access the variable via access macro. If you defined a specific body (implementation) for the access macro, TargetLink cannot build a simulation frame for SIL or PIL simulation. A warning message is displayed.

Demo location

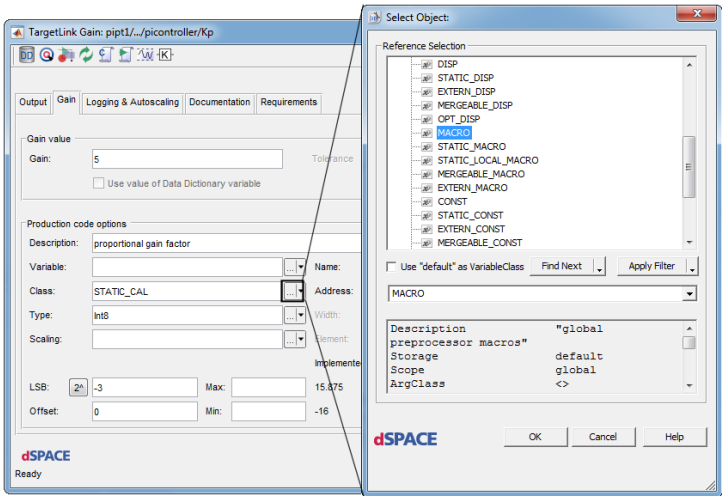
The following instructions show you how to apply the MACRO variable class to let TargetLink generate block parameters as preprocessor commands (C macros). For this example, use the PIPT1 demo model, which is located in `<DocumentsFolder>\dSPACE\TargetLink\<Version>\Demos\pipt1`.

Method

To generate a block parameter as a preprocessor command

1. Open the PIPT1 demo model and then the associated DD project file.
2. In the picontroller subsystem, double-click the desired block, for example, the Gain Kp block, to open the block dialog.

3 On the **Gain** page, click the **Class** Browse button to open the block selection dialog.



4 Select the **MACRO** variable class and click **OK**.

5 In the **Name** edit field, set the name of the variable to be generated to *P_Kp* or any other unique C identifier in the **Name** edit field. From this setting, the Code Generator creates a preprocessor command such as the following:

```
#define  MACRO_NAME    ((type) value)
```

Note

Make sure that neither a macro nor a variable is named identically to prevent naming conflicts. Otherwise, an error message is issued.

6 Scale the gain parameter according to the range it has to cover.

If the `Info` property is set to **none**, the **Max** and **Min** edit fields are disabled, because the macro value is fixed. If it is set to **adjustable**, the **Max** and **Min** values are interpreted as a possible value range. As a result, the macro value can be modified within this range later on, and the Code Generator takes this into account when propagating signal ranges from one block to another.

The macro definition contains a cast to the required type.

```
#define   P_Kp   ((Int8) 40)
```

7 Click the **Apply** button to confirm your settings and close the dialog.

8 Change to the **TargetLink Main Dialog** and generate production code.

Result

The results of code generation are as follows:

```
...
#define P_Kp ((Int8) 40) /*5.*/
...
extern UInt16 Sa1_REF /* LSB: 2^-11 OFF:  0 MIN/MAX:  0 .. 31.99951171875 */;
extern Int16 Sa1_U /* LSB: 2^-6 OFF:  0 MIN/MAX:  -512 .. 511.984375 */;
extern UInt16 Sa1_Y /* LSB: 2^-11 OFF:  0 MIN/MAX:  0 .. 31.99951171875 */;
```

```
...
extern void picontroller(void);
#endif /*_PICONTROLLER_H_ */
...
```

Related topics

Basics

Basics on Variable Classes..... 88

Examples

Example of Specifying Block Parameters as Input Arguments of Functions..... 114

Example of Initializing Variables via Restart Functions

Initializing a variable via the restart function

Suppose you want to initialize a variable when the restart function is called and not when the variable is defined. To do so, you have to set the initialization properties *InitAtDefinition* and *RestartFunctionName* accordingly.

Basics

Initialized variables, such as Unit-Delay states or Stateflow variables, are usually set to their initial values when they are defined or via the corresponding code of the restart function.

In most cases, it is not necessary to initialize variables twice. To avoid this, TargetLink provides the InitAtDefinition and RestartFunctionName properties for each variable class. You can modify these to define the initialization method to be used with each variable class.

Variable initialization during simulation

Even if both initialization methods are disabled, the variables are initialized in the simulation frame to enable SIL simulations and PIL simulations. When you export the generated files and integrate them into an ECU application, you have to supply the variable initialization. During production code generation, the Code Generator displays a warning message that reminds you to supply the correct variable initialization.

Initializing the model

To establish well-defined model behavior when the simulation is started, the following initializations are also performed:

- Each time the model is initialized for SIL simulation, the generated S-function is cleared from the MATLAB memory to force reloading the S-function.
- In PIL simulation, a special command is used for reinitialization so that the start-up code is executed each time a new simulation is started.

Using the restart function

If you want to initialize a variable only by means of the corresponding code in the restart function, specify the **RestartFunctionName** property of its variable class.

Method

To initialize a variable by using the restart function

1. Open the PIPT1 demo model and then the associated DD project file.
2. In the Data Dictionary Navigator, select the variable class you want to modify.
3. In the property value list, set the **InitAtDefinition** property to `off`.

   By default, **InitAtDefinition** is set to `on`. This means that variables associated with this variable class are initialized at their definition statements. You can set the InitAtDefinition property to `off` to prevent initialization.
4. In the property value list, click `RestartFunction` and enter the name of the function that is to be used.

   Via the RestartFunctionName property, you can assign separate restart functions to different variable classes. In addition, you can apply the name macros ($M, $N, $I) to specify the name of the initialization function.

   Note

   By default, the restart function is generated/placed into the same module as the VariableClass.
5. Click **OK** to confirm the settings and close the dialog.
6. Change to the **TargetLink Main Dialog** and generate production code.

Result

After code generation, all variables associated with this variable class are not initialized when they are defined but when the restart function is called.

Note

If a variable class with a restart function is selected for a state variable that resides inside an enabled system and for which *States when enabling: reset* is set, a second, separate initialization in the corresponding INIT function is called to reset the states.

Related topics

Basics

Basics on Variable Classes..... 88
VARIABLE_INITIALIZATION (📖 TargetLink Demo Models)

Example of Specifying Type Qualifiers via Type Prefixes

Type qualifiers

In C code, type qualifiers are keywords used to modify the properties of variables. Type qualifiers cannot be discarded. For example, passing the address of a const-qualified variable to a function expecting a pointer to the respective unqualified type leads to a compilation error.

Type qualifiers in TargetLink

In TargetLink, type qualifiers are specified via the DD **TypePrefix** property of a DD **VariableClass** object.

TargetLink supports the following type qualifier keywords:

- `const`
- `volatile`

Other type qualifiers are compiler-specific, language-specific, or target-specific. Depending on your target/compiler combination, you have to add other type qualifiers in the `TargetConfig.xml`. Refer to How to Enhance the AddressQualifiers XML object on page 559.

Example

In this example, the `STATIC_GLOBAL` variable class of the PIPT1 demo model is declared by the `const` type qualifier via the Data Dictionary.

1. Select the `STATIC_GLOBAL` DD **VariableClass** object.
2. Set the DD **TypePrefix** property to `const`.
3. In the picontroller subsystem, select the **Kp** Gain block and assign the `STATIC_GLOBAL` variable class as the output variable class.

Related topics

Basics

Basics on Memory-Specific Pointers..... 557
Basics on Variable Classes..... 88

HowTos

How to Enhance the AddressQualifiers XML object..... 559

Example of Merging Variables

Merging variables

You can merge multiple signals to one variable in the generated code.

Note

Merge variables very carefully. If you merge variables, it is your responsibility to consider data dependencies that are not reflected by the model. The Code Generator does not take these data dependencies into account, for example, when determining the execution order of blocks. At the very least, you have to check whether signals are merged in safety-related projects.

Merging variables faciliates the following:

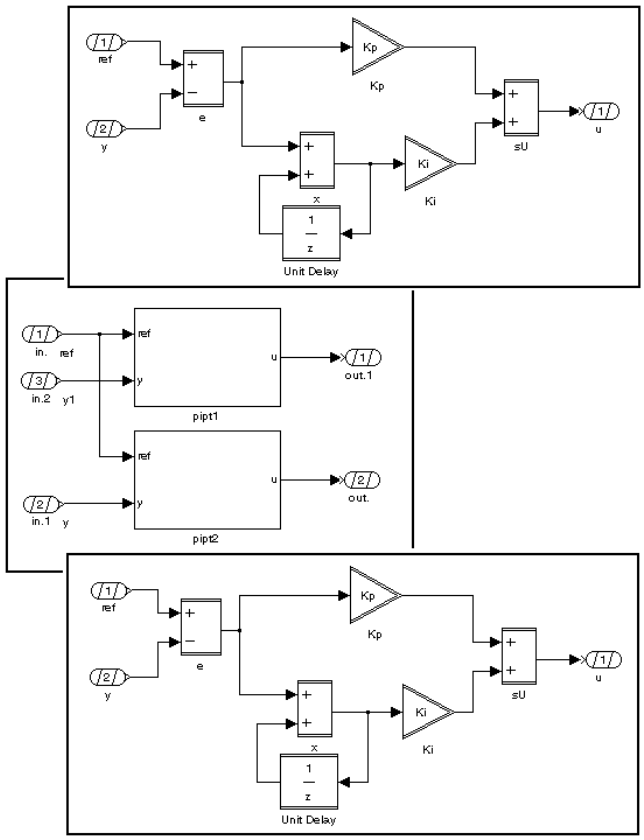
- Avoiding name ambiguity errors.
- Using system parameter variables more than once in the model. This is especially important for calibratable parameters.
- Reducing memory consumption.

To merge variables, you have to assign each variable to a DD VariableClass object whose Optimization property contains `MERGEABLE`, for example, the predefined MERGEABLE_CAL or MERGEABLE_GLOBAL variable classes.

Merging multiple parameters or identical table axis vectors

You can merge multiple parameters or identical table axis vectors to one variable in the generated production code.

Example Your model contains two instances of the PIPT1 demo model in which the value of the gain parameters Ki and Kp are identical and have to be calibratable.



To generate one calibratable variable for the gain parameters in the code, you have to assign the MERGEABLE_CAL variable class to the parameters.

You can assign variable classes by modifying the block properties as follows:

- How to Modify Block Properties via Block Dialogs (📖 TargetLink Preparation and Simulation Guide).
- How to Multi-Edit Property Values (📖 TargetLink Preparation and Simulation Guide).

The generate code fragment reads as follows:

```
UInt16 X_Sa3_Unit_Delay = 0 /* LSB: 2^-6 OFF:  0 MIN/MAX:  0 .. 1023.984375 */;
MERGEABLE_CAL Int8 P_Sa3_Kp = 40 /*5.*/ /*
   Description: proportional gain factor
   LSB: 2^-3 OFF:  0 MIN/MAX:  -16 .. 15.875 */;
```

Tip

It is easier to identify block variables mapped to the same variable in the code if you assign a DD **Variable** object referencing a DD **VariableClass** object whose **Optimization** property contains `MERGEABLE`. Refer to How to Reference DD Variable Objects to Block Variables (📖 TargetLink Preparation and Simulation Guide)

Whenever possible, use initialization values directly from the **Data Dictionary** to ensure consistent initialization values for all merged variables. Refer to How to Reference Values from DD Variable Objects to Block Variables (📖 TargetLink Preparation and Simulation Guide).

Related topics

Basics

Basics on Variable Classes 88

HowTos

How to Modify Block Properties via Block Dialogs (📖 TargetLink Preparation and Simulation Guide)
How to Multi-Edit Property Values (📖 TargetLink Preparation and Simulation Guide)
How to Reference DD Variable Objects to Block Variables (📖 TargetLink Preparation and Simulation Guide)
How to Reference Values from DD Variable Objects to Block Variables (📖 TargetLink Preparation and Simulation Guide)

References

Optimization (📖 TargetLink Data Dictionary Reference)
Value (📖 TargetLink Data Dictionary Reference)

Example of Defining Variable Classes for Alias Variables

Avoiding variable declaration

TargetLink provides the Alias property in the variable classes. If you set the Alias property to on, the Code Generator does not declare and does not initialize variables associated with the variable class. The Code Generator assumes that the alias variable has already been declared in a custom header file, which is specified using the **Addfile** block.

In contrast to extern global variables, TargetLink does not generate an extern declaration into a header file. It just uses the symbol as it is in the algorithmic part of the code.

Method

To define a variable class for alias variables

1. In the Data Dictionary Navigator, select `/Pool/VariableClasses`.
2. Copy an existing variable class or create a new one from the context menu.
3. In the property value list, set the Alias property to `on`.
4. Check the other properties of the new variable class.

Result

The Code Generator does not declare nor initialize variables associated with the variable class.

Working With Data Types

Customizing Data Types

Customizing data types for code generation

Before you start the code generation process, you have to specify the data types to be used, for example, for block outputs, parameters, and states. If necessary, you can customize data types used with TargetLink.

Where to go from here

Information in this section

Overview on Data Type Handling 127

Basics on TargetLink Base Types, Typedefs, and Header Files 128

How to Create New Type Definitions 134

How to Rename Base Types 136

How to Create User-Defined Data Types 139

Example of Controlling the Generation of the File Containing TargetLink Base Data Types 143

Overview on Data Type Handling

Contexts of data type handling in TargetLink

Contexts of data type handling in TargetLink

Context	Documentation
Mapping Simulink® data types to TargetLink data types	Details on Synchronizing Simulink and TargetLink Data (📖 TargetLink Preparation and Simulation Guide)
Stateflow® data types	Basics on Stateflow Data Types and Variable Scaling (📖 TargetLink Preparation and Simulation Guide)
TargetLink data types and typedefs	Basics on TargetLink Base Types, Typedefs, and Header Files on page 128
Classic AUTOSAR data types	Basics on Classic AUTOSAR Data Types in the Data Dictionary (📖 TargetLink Classic AUTOSAR Modeling Guide)

Context	Documentation
Adaptive AUTOSAR data types	Introduction to Data Types in Adaptive AUTOSAR (TargetLink Adaptive AUTOSAR Modeling Guide)

Basics on TargetLink Base Types, Typedefs, and Header Files

Base types, data types and typedef names

When working with data types in TargetLink, it is important to distinguish between the following:

- TargetLink base types
- TargetLink data types
- C data types
- C typedef names

They are all specified via DD **Typedef** objects.

Note

According to the C language standard, a typedef declaration creates a typedef name, not a new data type. As a consequence, a non-structured TargetLink data type specified via a DD **Typedef** object does not correspond to a C data type.

The following table shows the different options for the specification of DD Typedef objects and the resulting C code representation:

TargetLink	DD Typedef Object Specification	C Code Representation, Example, and Default Header File	Documentation
Base type	▪ IsBaseType set to on ▪ BaseTypeRename specified ▪ CreateTypedef set to on	typedef name that is mapped to platform-specific data type: `typedef signed short int Int16;` default header file: `tl_basetypes.h`	TargetLink base types on page 130
Non-structured data type	▪ BaseType references a non-structured base type ▪ CreateTypedef set to on	typedef name that is mapped to the specified base type: `typedef Int16 MyInt16;` default header file: `udt_$I.h`	TargetLink data types on page 131
	▪ BaseType references a non-structured base type ▪ CreateTypedef set to `off`	- No typedef declaration is generated. The appropriate base type is used instead. default header file: -	

Explicitly specified structured data type	▪ BaseType set to `Struct`
	▪ CreateTypedef set to on
		new data type and typedef name:
		`typedef struct MyStruct_tag {`
		`  // Components`
		`} MyStruct;`
		default header file: `udt_$I.h`
			How to Explicitly Specify Structured Data Types on page 152

	▪ BaseType set to `Struct`
	▪ CreateTypedef set to `off`
		new data type:
		`struct MyStruct_tag {`
		`  // Components`
		`};`
		default header file: `udt_$I.h`	

Implicitly structured data type	▪ BaseType set to `Struct`
	No Components specified
	▪ CreateTypedef set to on
		new data type and typedef name:
		`typedef struct MyStruct_tag {`
		`  // Components generated by Code Generator`
		`} MyStruct;`
		default header file: `udt_$I.h`
			▪ How to Create Implicitly Specified Structured Variables on page 158
			▪ How to Let TargetLink Create Implicit Structs for Buses on page 469

	▪ predefined DD IMPLICIT_STRUCT Typedef object	new data type:
		`struct anImplicitStruct {`
		`  // Components generated by Code Generator`
		`} ;`
		default header file: `udt_$I.h`	

TargetLink base types

Base types are specified by DD **Typedef** objects whose **IsBaseType** and **BaseTypeRename** property are set. All base types must either be a predefined TargetLink base type or a rename of a predefined base type.

All **DD project file templates** include predefined base types that you can use out-of-the-box.

Usage of base types TargetLink uses base types for the following:

- Explicitly specified uses of the base types
- Variables that are not explicitly specified
- All casts to numerical types

This can lead to a cast to a base type even if the typedef names of all variables of an arithmetic operation are user-defined.

Some of the base type properties directly affect the generated code:

- The size in bytes affects the order of struct components of implicitly generated struct types as TargetLink tries to avoid alignment gaps.
- The AtomicAccess flag specified in the `TargetConfig.xml` indicates whether a memory access can be interrupted by an RTOS interrupt.

Other properties also help TargetLink provide required information for other tools, for example, ByteOrder, which is important for ASAM MCD-2 MC file generation.

By default, base types are mapped to basic C types whose width is platform-dependent as shown in the following example:

```
typedef signed short int Int16;
```

Customizing base types The base types are mapped to basic C types. This mapping is controlled by the **Code generation target settings** and can be adjusted via the following options:

- Selecting a different code generation target.
- Adjusting the *Coded Type* of the corresponding `TargetConfig.xml`. Refer to How to Clone Target/Compiler Combinations to Outside the TargetLink Installation on page 44.

The names of the base types can be adjusted via the **BaseTypeRename** property of the DD **Typedef** object. Refer to How to Rename Base Types on page 136.

Customizing base type header files By default, the typedef declarations for the base types are generated into the `tl_basetypes.h` header file . You can customize the header files as follows:

Customization	Documentation
You can rename the header file.	Renaming header files on page 132
You can use a custom header file instead of the generated one.	Example of Controlling the Generation of the File Containing TargetLink Base Data Types on page 143

Customization	Documentation
You can activate the generation of a central header file. This lets you change the mapping to platform-specific types in a single file rather than multiple files, if necessary.	Central header file on page 133

TargetLink data types

TargetLink data types are specified via DD **Typedef** objects whose **IsBaseType** property is set to `off`. These data types only result in new C data types if they are a structured data type, i.e., the **BaseType** property is set to **Struct**. However, a C typedef name can be generated regardless of the referenced base type.

You can create new TargetLink data types and corresponding C typedef names or C data types as follows:

- For structured data types, refer to How to Explicitly Specify Structured Data Types on page 152 or How to Create Implicitly Specified Structured Variables on page 158.
- For non-structured data types, refer to How to Create User-Defined Data Types on page 139.
- With the exception of AUTOSAR arrays, arrays are not specified via DD **Typedef** objects but via one of the following:
  - the width of the signal in the model
  - the **Width** property of DD **Variable** objects
  - the **Width** property of components of DD **Typedef** objects that specify a structured data type.

  For AUTOSAR arrays refer to How to Define Array Implementation Data Types (Classic AUTOSAR) (📖 TargetLink Classic AUTOSAR Modeling Guide).

  For array-of-struct variables, refer to How to Specify Array-of-Struct Variables (📖 TargetLink Preparation and Simulation Guide).

Controlling the generation of typedef declarations By default, TargetLink does not generate typedef declarations for DD **Typedef** objects. The corresponding base type, or in case of structured data types the struct and struct tag, is used in the production code instead.

If you set the **CreateTypedef** property of the DD **Typedef** object to **on**, TargetLink generates typedef declarations.

The corresponding typedef declaration maps the typedef name for the TargetLink data type to the typedef name of the referenced base type or to the user-defined struct type.

By default, the typedef declarations are generated into the `udt_$I.h` header file .You can customize the header files as follows:

Customization	Documentation
You can rename the header file.	Renaming header files on page 132
You can select a module whose header file will contain the typedef declaration.	Changing the module on page 133

However, depending on your model and optimization settings, a typedef declaration might not be generated.

Referencing data types in block dialogs In the TargetLink block dialogs, you can select only DD Typedef objects and not the data types, basetypes or their properties directly.

This lets you reference already specified combinations of data types and properties instead of having to combine them for each reference or selecting a reference from a list of all possible combinations.

Reserved identifiers

TargetLink generates the following header files with reserved identifiers in addition to the header files `tl_basetypes.h` and `udt_$I.h`:

- `tl_dsfxp_types.h` for the data types and the corresponding typedef declarations used for the fixed-point library
- `tl_sim_types.h` for the data types and the corresponding typedef declarations used for the simulation code
- `tl_std_types.h` if the UsePlatformTypeAbstractionLayer (TargetLink Model Element Reference) Code Generator option is set to on.

These data types and the resulting typedef names are reserved for use by TargetLink.

Using any of the reserved typedef names will lead to an error.

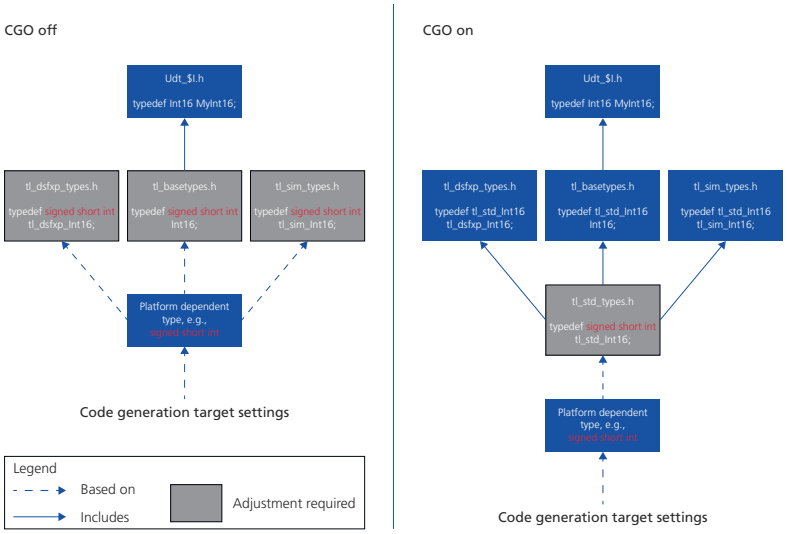
DsFxP	Simulation code	Abstraction layer
`tl_dsfxp_Bool`	`tl_sim_Bool`	`tl_std_Bool`
`tl_dsfxp_Float32`	`tl_sim_Float32`	`tl_std_Float32`
`tl_dsfxp_Float64`	`tl_sim_Float64`	`tl_std_Float64`
`tl_dsfxp_Int8`	`tl_sim_Int8`	`tl_std_Int8`
`tl_dsfxp_Int16`	`tl_sim_Int16`	`tl_std_Int16`
`tl_dsfxp_Int32`	`tl_sim_Int32`	`tl_std_Int32`
`tl_dsfxp_UInt8`	`tl_sim_UInt8`	`tl_std_UInt8`
`tl_dsfxp_UInt16`	`tl_sim_UInt16`	`tl_std_UInt16`
`tl_dsfxp_UInt32`	`tl_sim_UInt32`	`tl_std_UInt32`

Header files

Renaming header files You can rename the generated header files by changing the NameTemplate property of the relevant DD Module object. Refer to ModuleInfo.

Changing the module For user-specified DD Typedef objects that do not specify a renamed base type, you can use the ModuleRef property to reference a DD Module object. In this case, the typedef declaration is generated into the header file of the module.

Central header file You can use the UsePlatformTypeAbstractionLayer (📖 TargetLink Model Element Reference) Code Generator option to generate an *additional* header file. This header file maps central typedef names, by default starting with the `tl_std_` prefix , to platform-specific data types. This lets you change the mapping between platform-specific data types and typedef names for the production code, simulation code, and fixed-point library code in a single file.



Reserved prefixes

The identifier prefix `tl_` is reserved for use by TargetLink. Using the prefix in an identifier for one of the following will lead to a warning:

- Data types
- Variables
- Functions
- EnumValues
- Macros

Using the prefix for a struct tag or struct component leads to an advice.

Related topics

HowTos

How to Clone Target/Compiler Combinations to Outside the TargetLink Installation..... 44
How to Create Implicitly Specified Structured Variables..... 158
How to Create User-Defined Data Types..... 139
How to Explicitly Specify Structured Data Types..... 152
How to Let TargetLink Create Implicit Structs for Buses..... 469
How to Rename Base Types..... 136
How to Specify Array-of-Struct Variables (📖 TargetLink Preparation and Simulation Guide)

Examples

Example of Controlling the Generation of the File Containing TargetLink Base Data Types..... 143

References

ModuleInfo (📖 TargetLink Data Dictionary Reference)
Typedef (📖 TargetLink Data Dictionary Reference)

How to Create New Type Definitions

Objective

Defining your own DD Typedef objects is useful if you want to introduce synonyms for base types which have been declared somewhere else. The synonyms are then equivalent to the base type and are often used to tailor the program to the target platform.

Basics

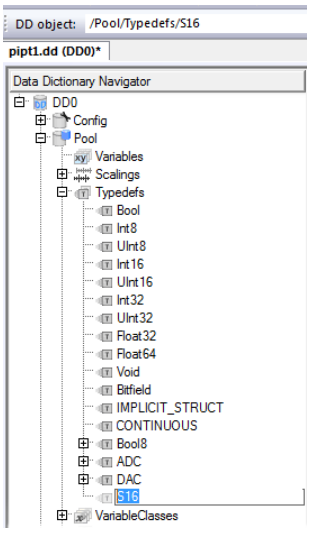
Once you introduce a typedef, it can be referenced for variable definitions. DD Typedef objects are thus useful in implementing company-specific coding styles. However, setting up DD Typedef objects is rarely the province of day-to-day coding and usually remains a task for administrators.

Method

To create new type definitions

1. In the `/Pool` area of the Data Dictionary Navigator, right-click **Typedefs**.
2. From the context menu, select **Create Typedef**.
   A new typedef object named **new_Typedef** is created.

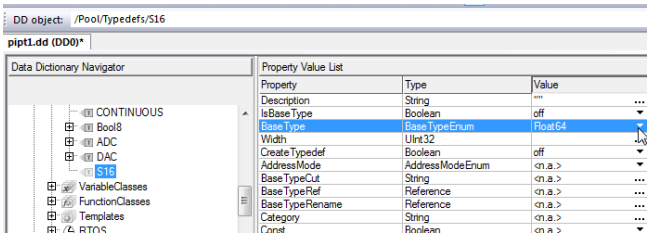
3 Enter a unique name, for example, S16.



Note

The type name must be a valid C identifier, consisting of letters, digits and underscores.

4 In the property value list, click the arrow to open the list of the available base types.



5 Select the base type you want to assign to the new typedef.

Result

You have created a new DD Typedef object and assigned a suitable base type.

Related topics

Basics

Basics on TargetLink Base Types, Typedefs, and Header Files..... 128

HowTos

How to Create User-Defined Data Types..... 139
How to Rename Base Types..... 136

How to Rename Base Types

Objective

Your company-specific coding style usually also requires specific names for the data types used in the generated production code. Base types are hardware-related, which is why you cannot rename them directly. To bypass this restriction, TargetLink provides a renaming mechanism that allows you to define synonyms that match your company-specific coding style.

Suppose your company-specific coding style requires using S16 for a signed 16-bit integer variable instead of Int16, which is TargetLink's default.

Process steps

You will have to perform the following process steps:

1. Create a typedef named S16 (if not already done) first.
2. Reference the S16 typedef in the typedef representing the base type you want to rename by setting its BaseTypeRename property. For example, if Int16 is to be renamed, set the BaseTypeRename property of `/Pool/Typedefs/Int16` to S16.

Note

- Both the typedef representing the original base type and the renamed base type must always exist, and they must exist as a pair.
- If you enter the original base type in any block dialog after you have created the pair of typedefs, the original base type, for example, Int16, is automatically replaced by its associated synonym S16. In addition, the original base type is also no longer displayed and therefore cannot be selected in any selection dialog.

Base type synonyms

Once you finished the process described above, the new base type names (synonyms) are used in all declarations, function prototypes, and casts in the C code and header files. You can also reference them in block dialogs, generated documents and so on.

Tip

TargetLink also supports shortcuts for type names, which are especially useful if you want to include data type information in the variable name. You can define short type names via the BaseTypeCut property of the type definition concerned. To apply the BaseTypeCut, add the name macro $T in the edit field of the variable concerned. For details on name macros, refer to Basics on Using Name Macros on page 78.

Precondition

The following precondition must be fulfilled:

- The desired DD project file (`<name>.dd`) is open.

Method

To rename base types

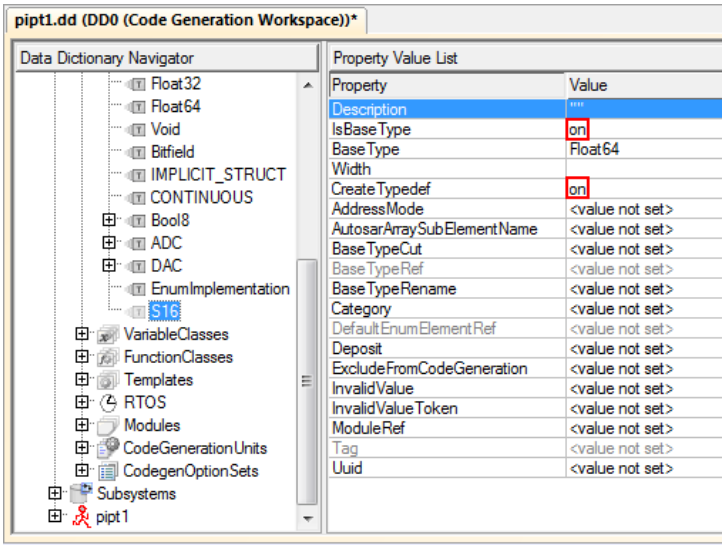
1 Create a new type definition, for example, S16.

For details, refer to How to Create New Type Definitions on page 134.

2 In the Property Value list of the S16 typedef, set first IsBaseType and then CreateTypedef to on.

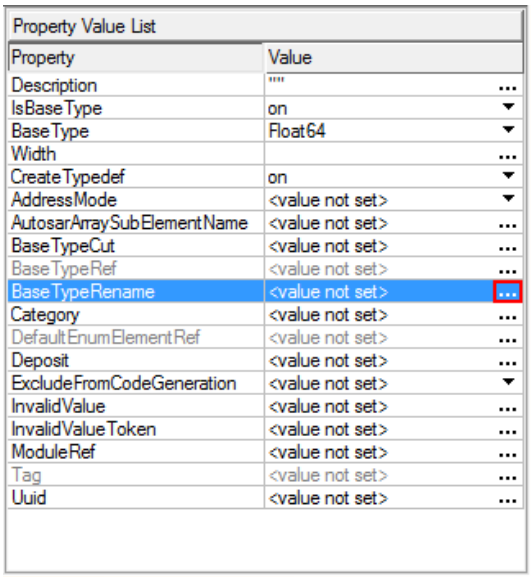
3 Click the BaseTypeRename button and select S16 in the Select Object dialog.

The new type definition now looks similar to the one shown below.

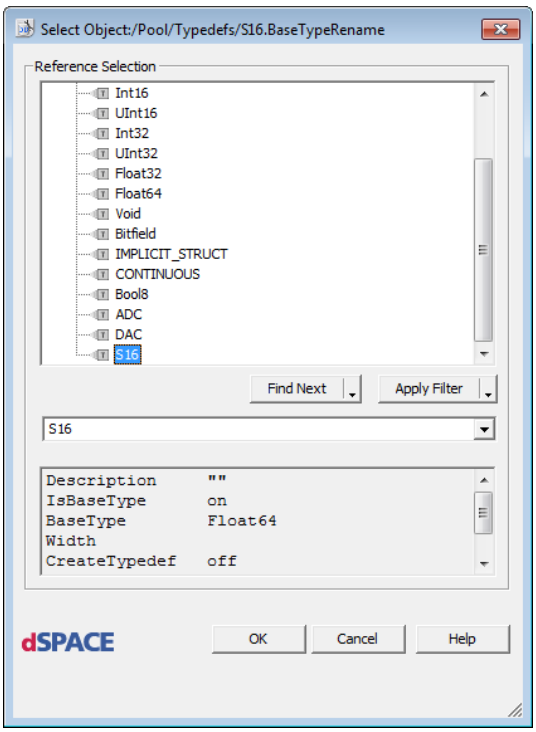


4 In the Data Dictionary Navigator, select the typedef you want to rename, for example, Int16.

5 In the Property Value List, click the **BaseTypeRename** browse button to open the **Select Object** dialog.



6 Select the **S16** typedef to reference it and click **OK**.



You have now finished all the necessary preparation steps. If you also want to apply short type names in variable names, proceed with the following instructions.

7 Open the block dialog where you want to include data type information in the variable name.

8 Add the $T name macro in the **Name** edit field and click **Apply**.

9 Start the code generation process.

Result

The Code Generator generates production code where all occurrences of Int16 are replaced with S16, and short cut types are inserted for the variables you specified a name macro for.

The generated code for the demo model POSCONTROL looks like this:

```
CAL S16 P_Sa1_Kp_S16 = 128 /* 2. */
   /* LSB: 2^-6 OFF:  0 MIN/MAX:  -512 .. 511.984375 */;
...
void controller(void)
{
   /* SLLocal: Default storage class for local variables | Width: 16 */
   S16 Sa1_e /* LSB: 2^-10 OFF:  0 MIN/MAX:  -32 .. 31.9990234375 */;
   S16 Sa1_si /* LSB: 2^-9 OFF:  0 MIN/MAX:  -64 .. 63.998046875 */;
...
   Sa1_e = (S16) (((UInt16) Sa1_REF) - ((UInt16) Sa2_LIN_POS));
   /* Sum: controller/si */
   Sa1_si = (S16) (((UInt16) (S16) (Sa1_e >> 1)) + ((UInt16) X_Sa1_xi));
...
}
```

The `tl_basetypes.h` file will contain `typedef signed short int S16; /* 16 bit signed integer basetype */` instead of `typedef signed short int Int16; /* 16 bit signed integer basetype */`.

Related topics

Basics

Basics on TargetLink Base Types, Typedefs, and Header Files..... 128
Basics on Using Name Macros..... 78

HowTos

How to Create New Type Definitions..... 134
How to Create User-Defined Data Types..... 139

How to Create User-Defined Data Types

Objective

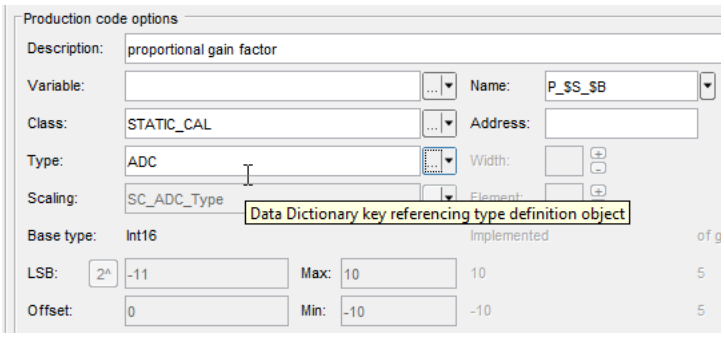
User-defined data types improve the code readability and are also useful if you often have to assign identical constraints (fixed scaling parameters and min/max values) to multiple signals of one kind in your model.

Note

A user-defined data type is a derivative of a typedef (type definition) and usually establishes a combination of base types and constraints.

DD Typedefs objects

You can define user-defined data types in the `/Pool/Typedefs` subtree in the TargetLink Data Dictionary. If a variable (block output, parameter or state) is associated with a user-defined data type, it inherits the data type's base type and, optionally, the constraints (fixed-point scaling parameters and constrained limits (Max, Min)), for example:



The Code Generator also generates a user-defined data type if the data type of a variable is set to IMPLICIT_STRUCT.

User-defined data types and code generation

During production code generation, the information provided by the TargetLink Data Dictionary concerning user-defined data types is used to generate code with the specified data type names, scalings and constrained ranges. In the generated production code, the declaration of the block output variable will read:

```
void picontroller(void)
{
   ...
   Int16 Sa1_Ki /* LSB: 2^-6 OFF:  0 MIN/MAX:  -512 .. 511.984375 */;
   ADC Sa1_Kp /* LSB: 2^-11 OFF:  0 MIN/MAX:  -10 .. 10 */;
   Int16 Sa1_e /* LSB: 2^-10 OFF:  0 MIN/MAX:  -32 .. 31.9990234375 */;
   ...
}
```

The name of the user-defined header file is set to `udt_$I.h`, by default. You can change the name of the user-defined header file via the NameTemplate property of the /Pool/Modules/TLPredefinedModules/UserDataTypes module object.

Note

You can use the UseRootFileNameAsUserDefinedTypesHeaderBasename Code Generator option to achieve the behavior of previous TargetLink versions ≤3.0. For these TargetLink versions, the name of the header file is built from the TargetLink subsystem name and the extension `_udt`, for example, `picontroller_udt.h`.

If you take a closer look at this file, you will only find the type definitions that are actually used in the particular code module. The definition reads:

```
#ifndef _PICONTROLLER_UDT_H
#define _PICONTROLLER_UDT_H
#include "tl_basetypes.h"
#ifndef _ADC_TYPE
#define _ADC_TYPE
typedef Int16 ADC; /*
   User-defined data type ADC. Imported from TL 1.3
   LSB: 2^-11 OFF:  0 MIN/MAX:  -16 .. 15.99951171875 */
#endif /* ADC_TYPE_ */
#endif/*_PICONTROLLER_UDT_H_ */
```

Note

- For non-scalar (vector/2-D matrix) variables, selecting a user-defined data type means that you set the data type, LSBs, constraints and offsets of all vector elements to the values associated with the user type.
- For the PreLook-Up Index Search block, the LSB of the fraction (f) depends on the width of the selected data type. For example, if UInt8 is selected, the LSB of the fraction is automatically 2^{-8}. You cannot edit this value or reference user-defined data types that have constraints.
- User-defined data types that are associated with a block variable may become invalid if they are renamed or erased from the DD project file later on.
- There are some block variables where TargetLink does not support scaling offsets. This applies to:
  - State-space matrices of Discrete State-Space blocks
  - Coefficient vectors of Discrete Transfer Fcn blocks
  - Coefficient vectors of Discrete Filter blocks
  - Slew rates of Rate Limiter blocks
  - Coefficient and Delay Line vectors of the FIR Filter block

    In the block dialog pages of these blocks, you can select only user types whose offset is zero. They become invalid if you redefine them with a nonzero offset. In this case, code generation will not work, and TargetLink will show an error message when you reopen the dialog page.

Method

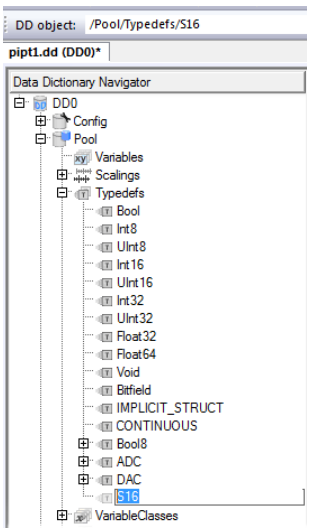
To create a user-defined data type

1 In the Data Dictionary Navigator, right-click /Pool/Typedefs.

2 From the context menu, select Create Typedef.

A new typedef object named new_Typedef is created.

3 Enter a unique name, for example, S16.



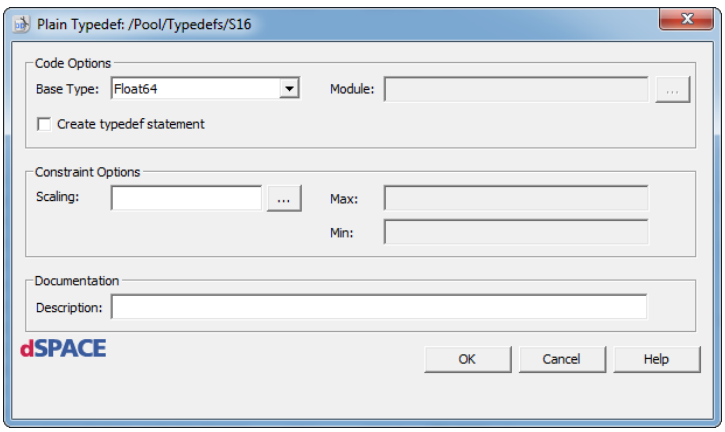
Note

The type name must be a valid C identifier, consisting of letters, digits and underscores.

You have created a new typedef object but the default base type is still Float64, which may not be suitable for your project. To set the typedef properties, carry out the following steps.

4 Double-click the typedef you created beforehand.

The Plain Typedef dialog opens.



5 In the dialog, enter all the relevant data.

It is not possible to rename a typedef object via the Plain Typedef dialog. If you reference an existing scaling object, you can enter constrained limits (**Max**, **Min**). If you enter invalid attributes, an error message is displayed.

Note

If you enter constrained limits, you guarantee that the signal does not exceed either the upper (Max) or the lower (Min) constrained limit. TargetLink exploits these values for overflow protection and optimization.

6 Click **OK** to save the settings.

The dialog is closed and a constraints object is created below the typedef object.

Result

You have now created a user-defined data type, which you can use in any TargetLink block dialog. User-defined data types are not available in the BaseTypes drop-down list in the property value list, which contains only built-in TargetLink base types. In addition, a user-defined data type must not necessarily correspond to a C typedef in the generated production code. This only applies if you set the CreateTypedef property to **on**.

Related topics

Basics

Basics on TargetLink Base Types, Typedefs, and Header Files..... 128

HowTos

How to Create Implicitly Specified Structured Variables..... 158
How to Create New Type Definitions..... 134
How to Rename Base Types..... 136
How to Specify Global Scaling and Constrained Range Limits for DD Typedef Objects (TargetLink Preparation and Simulation Guide)

Example of Controlling the Generation of the File Containing TargetLink Base Data Types

Controlling the generation of the file containing the TargetLink base data types

During code generation, the Code Generator evaluates predefined target configuration files (`TargetConfig.xml`). A separate configuration file for each supported target is available in `<TL_InstRoot>\Matlab\Tl\TargetConfiguration\<MicrocontrollerFamily>\<CompilerFamily>`. The files contain information on the base data types

(base types) of the processor/compiler combination (target platform), such as the byte order and alignment, and are used to create the `tl_basetypes.h` header file.

This file provides the type definitions originating from the selected processor/compiler combination (`typedef signed long int Int32; /* 32 bit signed integer base type */`). TargetLink basically maps base types to type definitions, called typedefs below. A typedef defines a new data type by creating a named alias for an already defined data type which can be used as a basis for renaming base types or creating your own data types (user-defined data types). A user-defined data type is a derivative of a typedef and usually defines a combination of base types and constraints. For details, refer to Customizing Data Types on page 127.

If you want to rename `tl_basetypes.h` or include your own definitions specified in a handcoded header file instead of the one generated by TargetLink, you can achieve this by using a module object in `/Pool/Modules/TLPredefinedModules/`. A module object represents the file created in the file system during production code generation. By setting module object properties, you can influence the generation of the internal file `tl_basetypes.h`, for example. To use a handcoded file instead of generating `tl_basetypes.h`, you have to perform two steps. First you have to rename the file name. Then you have to suppress the generation of `tl_basetype.h`.

Note

As of TargetLink 3.1, module templates are not applied to modules that are specified via module objects in the TargetLink Data Dictionary.
The following example is based on the PIPT1 demo model located in the `<DocumentsFolder>\dSPACE\TargetLink\<Version>\Demos\pipt1` folder.

Method

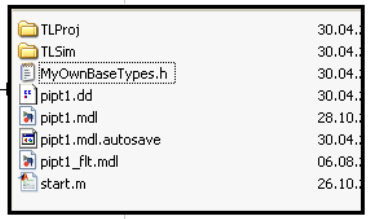
To control the generation of the file containing the TargetLink base data types

1. Create a header file containing your type definitions and save it in a suitable folder, for example, `<DocumentsFolder>\dSPACE\TargetLink\<Version>\Demos\pipt1`.
   This file must contain the definitions of all the base data types and their names you want to use for modeling and code generation.

   Note

   You have to guarantee that all data types that are defined in the handcoded file have the correct word length (e.g. U16 represents an unsigned integer and has 16-bit word length).

```
/*****************************************************************************************\
 ***
 *** Simulink model       : pipt1
 *** TargetLink subsystem : pipt1/picontroller
 *** Codefile             : MyOwnBaseTypes.h
 ***
 *** Generation date: 2012-08-31 14:33:50
 ***
 *** TargetLink version      : 4.0  from 21-Oct-2014
 *** Code generator version  : Build Id 4.0.0.23 from 2014-10-2
 *** Copyright (c) 2014 dSPACE GmbH
\*****************************************************************************************/

#ifndef MyOwnBaseTypes_H
#define MyOwnBaseTypes_H

typedef unsigned char Bool; /* boolean basetype */
typedef float Float32; /* 32 bit floating-point basetype */
typedef double Float64; /* 64 bit floating-point basetype */
typedef signed short int S16; /* 16 bit signed integer basetype */
typedef signed long int S32; /* 32 bit signed integer basetype */
typedef signed char S8; /* 8 bit signed integer basetype */
typedef unsigned short int US16; /* 16 bit unsigned integer basetype */
typedef unsigned long int US32; /* 32 bit unsigned integer basetype */
typedef unsigned char US8; /* 8 bit unsigned integer basetype */
typedef void Void; /* void basetype */

#endif/*MyOwnBaseTypes_H */
```



2 Open the PIPT1 demo model.

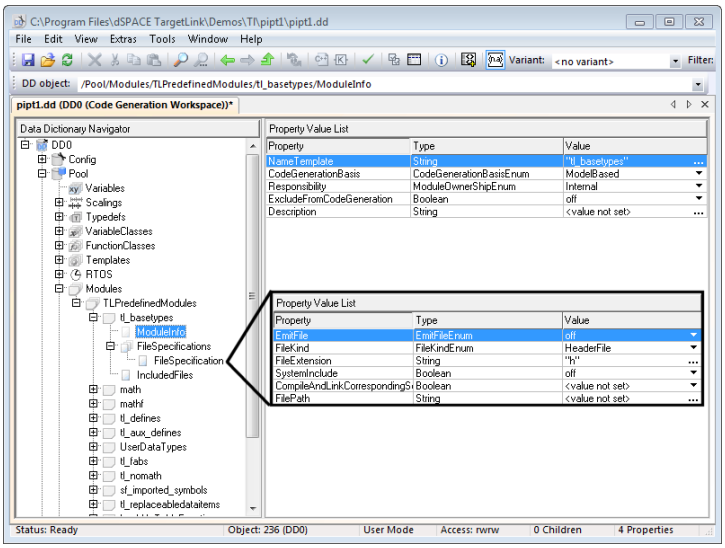
3 Append the path to the header file to the list of include paths in the IncludeFileSearchPath property of the DD `/Config/Targetlink` object.

4 In the TargetLink Data Dictionary, create all the data types in the `/Pool/Typedefs` group that are defined in your MyOwnBaseTypes header file and rename the original TargetLink base types afterwards.

   For details on renaming a type definition, refer to How to Rename Base Types on page 136.

5 In the Data Dictionary Navigator, select the `/Pool/Modules/TLPredefinedModules/tl_basetypes` DD object.

6 In the Property Value List, define the properties NameTemplate and EmitFile for ModuleInfo and FileSpecification as shown below.



As the EmitFile property is set to off, the type definitions specified in `MyOwnBaseTypes.h` are used but file regeneration is suppressed avoiding unintentional overwriting of the file.

Tip

If you just want to rename the `tl_basetypes.h` type definition file, it is sufficient to enter an alternative name for the NameTemplate property in `/Pool/Modules/TLPredefinedModules/tl_basetypes/ModuleInfo`. In this case, the content of the file is unchanged.

Result

Carrying out the above instructions disables the generation of the `tl_basetypes.h` type definition file and makes TargetLink include your own header file with your type definitions instead.

```
 *** Copyright (c) 2014 dSPACE GmbH
\*****************************************************************************

#ifndef _PICONTROLLER_H_
#define _PICONTROLLER_H_

/*----------------------------------------------------------------------------*\
  DEFINES (OPT)
\*----------------------------------------------------------------------------*/
/*----------------------------------------------------------------------------*\
  INCLUDES
\*----------------------------------------------------------------------------*/
#include "MyOwnBaseTypes.h"
/*----------------------------------------------------------------------------*\
  DEFINES
\*----------------------------------------------------------------------------*/
/*----------------------------------------------------------------------------*\
  TYPEDEFS
\*----------------------------------------------------------------------------*/
/*----------------------------------------------------------------------------*\
  ENUMS
\*----------------------------------------------------------------------------*/
/*----------------------------------------------------------------------------*\
  VARIABLES
\*----------------------------------------------------------------------------*/

/*****************************************************************************\
   SLGlobal: Default storage class for global variables | Width: 16
\*****************************************************************************/
extern UInt16 Sa1_REF /* LSB: 2^-11 OFF:  0 MIN/MAX:  0 .. 31.99951171875 */;
extern S16 Sa1_U /* LSB: 2^-6 OFF:  0 MIN/MAX:  -512 .. 511.984375 */;
extern UInt16 Sa1_Y /* LSB: 2^-11 OFF:  0 MIN/MAX:  0 .. 31.99951171875 */;
```

MyOwnBaseTypes.h | picontroller.h

Related topics

Basics

Basics on Specifying the Build Object (📖 TargetLink Interoperation and Exchange Guide)
Basics on TargetLink Base Types, Typedefs, and Header Files..... 128
Details on Providing a Target Config File (📖 TargetLink Interoperation and Exchange Guide)

Working With Structures and Pointers to Structures

Where to go from here

Information in this section

Arranging Data in Structured Variables..... 148

Structuring Production Code via Pointers to Structs..... 163

Arranging Data in Structured Variables

Introduction

You can specify structured data types in the Data Dictionary Manager and assign them to variables.

Where to go from here

Information in this section

Basics on Arranging Data in Structured Variables..... 148

How to Explicitly Specify Structured Data Types..... 152

How to Scale the Leaf Struct Components of Array-Of-Struct Variables with Non-Standard Scaling..... 154

How to Create Explicitly Specified Structured Variables..... 155

How to Create Implicitly Specified Structured Variables..... 158

Example of Collecting Gain Parameters via Dynamic Components..... 161

Basics on Arranging Data in Structured Variables

Struct variables

A structure, called a struct below, collects different types of data in one variable. A struct consists of components, which hold the different types of data, and is identified by a unique name. The allocation of the struct variable in the ECU memory depends on its variable class.

Structured data types

You can specify structured data types in the Data Dictionary Manager, and then assign them to variable objects. Once you have referenced a component of a

struct variable in the respective TargetLink block dialog, the component's DD-based specification is used for code generation.

TargetLink supports arbitrarily nested structs.

Tip

If you double-click a Typedef object, the **Plain Typedef** (📖 TargetLink Data Dictionary Manager Reference) dialog opens, which is especially useful when defining data types of the `Struct` kind.

Specifying structs

You can use different methods to specify structs. Some offer TargetLink degrees of freedom (implicit specification), others force TargetLink to create the struct variable from your explicit specification.

Implicit structs Implicit structs are called *implicit* because you do not specify the structured data type. Instead the structured data type is implicitly created by the Code Generator. Implicit structs result from DD Variable objects whose Type property is set to `IMPLICIT_STRUCT`.

Explicit structs Explicit structs are called *explicit* because the struct itself and all of its components are explicitly specified by a structured data type. The Code Generator creates the struct according to your explicit specification. Explicit structs result from DD Variable objects whose Type property references a structured data type. Structured data types are specified in the Data Dictionary by Typedef objects whose BaseType property is set to `Struct`.

Specifying components

TargetLink provides two kinds of components:

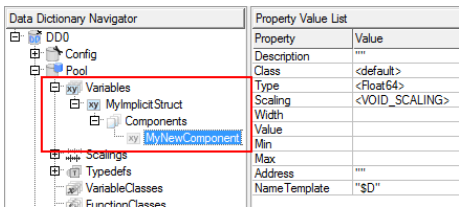
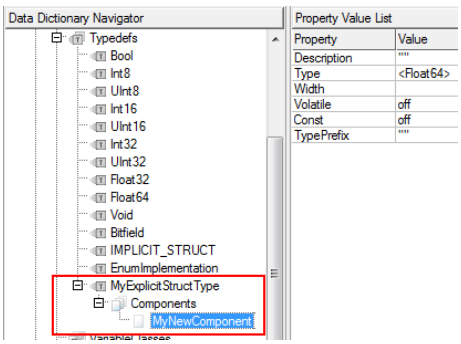
- Fixed components
- Dynamic components

Note

TargetLink expects that the VariableClass object referenced at struct components has its Scope property set to `struct_component` and its Storage property set to `default`.
For convenience, the VariableClass object's Scope property can also be set to `global`. In this case, TargetLink derives a suitable variable class to use for the struct component during code generation.

Fixed components

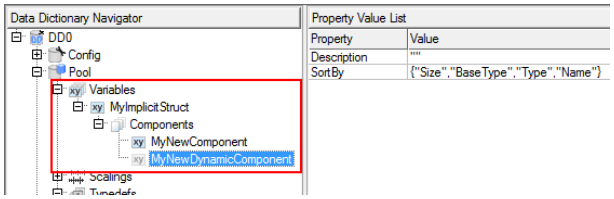
You can create fixed components for implicit or explicit structs.

Implicit Structs with Fixed Components	Explicit Structs with Fixed Components
Specify each fixed component as a Variable object in the struct variable's Components subtree: 	Specify each fixed component as a TypedefComponent object in the struct type's Components subtree: 

The settings made at the Variable/TypedefComponent of the Components subtree determine the contents of the struct in production code. Each object within the Components subtree is the specification basis for a struct component in production code.

Dynamic components

You can create dynamic components for implicit structs.

Implicit Struct with Dynamic Components
Specify each dynamic component as a DynamicComponent object in the struct variable's Components subtree: 

A DynamicComponent object is a placeholder that you can reference at different blocks.

During code generation, a struct component is implicitly created in the production code for each reference of the DynamicComponent object. This lets you implement one or more block variable(s) as components of the implicit struct.

The properties of each component depend on the block data that is relevant for the Code Generator: e.g., the block's Output page for an output variable, or a block's State page for a state variable.

The order of the struct's components depends on the criteria specified in the DynamicComponent object's SortBy property.

Note

Do not reference the same dynamic component(s) at blocks belonging to different CGUs.

Bitfields

According to C semantics, bitfields are struct components. By default, TargetLink collects plain variables of the `Bitfield` type in a global bitfield structure that it implicitly generates during code generation.

If you want to collect bitfield components in your own structures, you can specify each component via fixed or dynamic components.

Fixed Bitfield Component	Dynamic Bitfield Component
Make the following settings at the Variable/TypedefComponent within the Components subtree: ▪ Type set to `Bitfield` ▪ NumberOfBits set to `1` Reference the fixed component in the Variable edit field of one TargetLink block.	Make the following settings at the block whose block variable(s) you want to implement as a bitfield component: ▪ Reference the `Bitfield` type at the Type edit field ▪ Reference the dynamic component in the Variable edit field

Note

TargetLink also uses bitfield components for optimizing booleans and Stateflow states. For details, refer to Basics on Collecting Booleans and Stateflow States in Global Bitfield Structures on page 555.

Summary of methods

Specification	Use Case	Instructions
Explicit struct, fixed components	Creating a structured data type to use for several variables	1. Create a DD Typedef object whose BaseType property is set to `Struct`. 2. Explicitly specify the data type's components. 3. Reference the structured data type at one or more DD Variable object(s). 4. Reference one or more of the variable's component(s) at one or more TargetLink block(s).
	Creating a structured variable with an explicitly specified structured data type	
Implicit struct, fixed components	Creating a structured variable, leaving the structured data type implicit	1. Create a DD Variable object whose Type property is set to `IMPLICIT_STRUCT`. 2. Explicitly specify the variable's components. 3. Reference one or more of the variable's component(s) at one or more TargetLink block(s).

Specification	Use Case	Instructions
Implicit struct, dynamic components	Gathering block variables in structures via dynamic components	1. Create a DD **Variable** object whose **Type** property is set to `IMPLICIT_STRUCT`. 2. Create one or more dynamic component(s) of the variable. 3. Reference one or more dynamic component(s) at one or more TargetLink block(s).

Related topics

Basics

Mapping Buses and Subbuses to Implicit Structs in the Code..... 468
Mapping Entire Buses to Explicit Structs in the Code..... 479

HowTos

How to Create Explicitly Specified Structured Variables..... 155
How to Create Implicitly Specified Structured Variables..... 158
How to Create New Type Definitions..... 134
How to Explicitly Specify Structured Data Types..... 152

Examples

Example of Collecting Gain Parameters via Dynamic Components..... 161

How to Explicitly Specify Structured Data Types

Objective

To explicitly specify a structured data type.

The structured data type can be used to specify structured variables based on that type.

Explicit structs

Explicit structs are called *explicit* because the struct itself and all of its components are explicitly specified by a structured data type. The Code Generator creates the struct according to your explicit specification. Explicit structs result from DD **Variable** objects whose **Type** property references a structured data type. Structured data types are specified in the Data Dictionary by **Typedef** objects whose **BaseType** property is set to `Struct`.

Method

To explicitly specify a structured data type

1 Create a new DD Typedef object in the `/Pool/Typedefs` subtree and rename it as required.

> **Note**
>
> The name of the typedef must be a valid C identifier and must not contain a reserved prefix. Refer to Reserved prefixes on page 133.

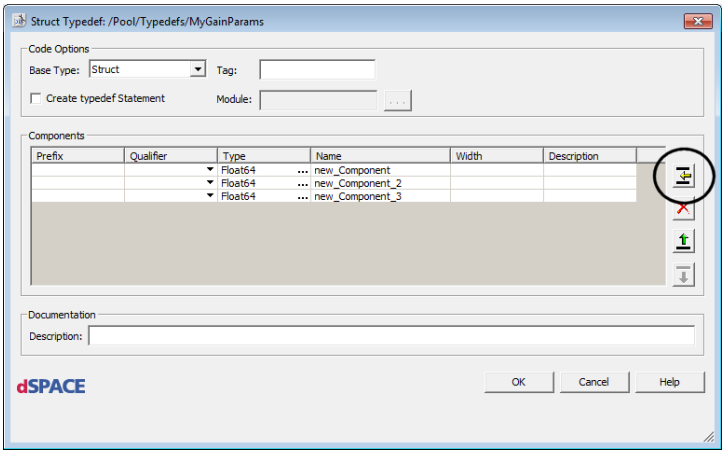
2 Double-click the new typedef to open the Plain Typedef dialog.

3 Set the Base Type to `Struct`.

The Struct Typedef dialog automatically opens.

4 If you want TargetLink to create a typedef declaration statement instead of a struct statement, select the Create typedef statement checkbox.

5 Add one or more components to the structured data type via the Add component button.



Each component represents a leaf struct component.

6 For each component, adjust the Name, Type, and Qualifier columns as required.

> **Tip**
>
> To create a nested struct, reference another struct via the Type property.

7 Click OK to close the dialog.

The struct type is created together with its components as specified.

8 If you want TargetLink to generate the struct definition in a separate module, specify a Module object and reference it at the ModuleRef property of the Typedef object.

Result

You have created a structured typedef.

Next step

You can create a variable object that references your structured typedef. For details, refer to How to Create Explicitly Specified Structured Variables on page 155.

Related topics

Basics

Basics on Arranging Data in Structured Variables..... 148

HowTos

How to Create Explicitly Specified Structured Variables..... 155
How to Create Implicitly Specified Structured Variables..... 158
How to Create New Type Definitions..... 134

Examples

Example of Collecting Gain Parameters via Dynamic Components..... 161

References

Create <DD Object> (TargetLink Data Dictionary Manager Reference)

How to Scale the Leaf Struct Components of Array-Of-Struct Variables with Non-Standard Scaling

Objective

To scale the leaf struct components of an array-of-struct variable with non-standard scaling, you must constrain the data type of each leaf struct component that you want to scale.

Restrictions

These instruction apply only to leaf struct components whose data type is a signed or unsigned integer.

No action is required for the following data types:

- Bool
- Bitfield
- Float
- Enum

Precondition

You created a constrained type for each leaf struct component of the array-of-struct variable you want to scale. Refer to How to Specify Global Scaling and

Constrained Range Limits for DD Typedef Objects (📖 TargetLink Preparation and Simulation Guide)

Method

To scale the leaf struct components of an array-of-struct variable with non-standard scaling

1. In the Data Dictionary, locate the DD **Typedef** object that is referenced by your array-of-struct variable and expand its **Components** subtree.
2. For each component that you want to scale with non-standard scaling, make the following settings:

Property	Value
Type	The constrained type you created for the component.

3. Locate the DD **Variable** object that references the DD **Typedef** object and use the **Adjust to Typedef** command to synchronize the **DD Variable** object with the referenced **DD Typedef** object.

Result

You scaled the leaf struct components of the DD **Typedef** object that is referenced at the array-of-struct variable with non-standard scaling.

Related topics

Basics

Basics on Arrays of Buses and Array-of-Struct Variables (📖 TargetLink Preparation and Simulation Guide)

HowTos

How to Explicitly Specify Structured Data Types..... 152
How to Specify Array-of-Struct Variables (📖 TargetLink Preparation and Simulation Guide)

References

Adjust to Typedef (📖 TargetLink Data Dictionary Manager Reference)

How to Create Explicitly Specified Structured Variables

Objective

To specify a structured variable based on a structured data type that is explicitly specified.

The variable's components can be referenced at TargetLink blocks.

Precondition

You created a DD Typedef object whose BaseType property is set to `Struct` and specified one or more component(s).

For details, refer to How to Explicitly Specify Structured Data Types on page 152.

Method

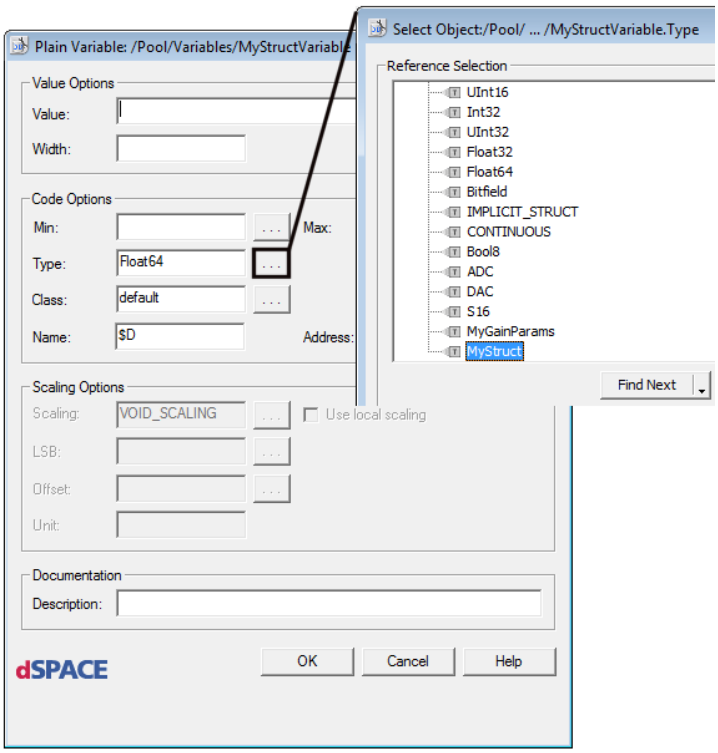
To create an explicitly specified structured variable

1 Create a new DD Variable object in the `/Pool/Variables` subtree and rename it as required.

> **Note**
>
> The variable's name must be a valid C identifier.

2 Double-click the variable object to open the Plain Variable dialog.

3 In the Type edit field, reference a DD Typedef object whose BaseType property is set to `Struct`: e.g., MyStruct.

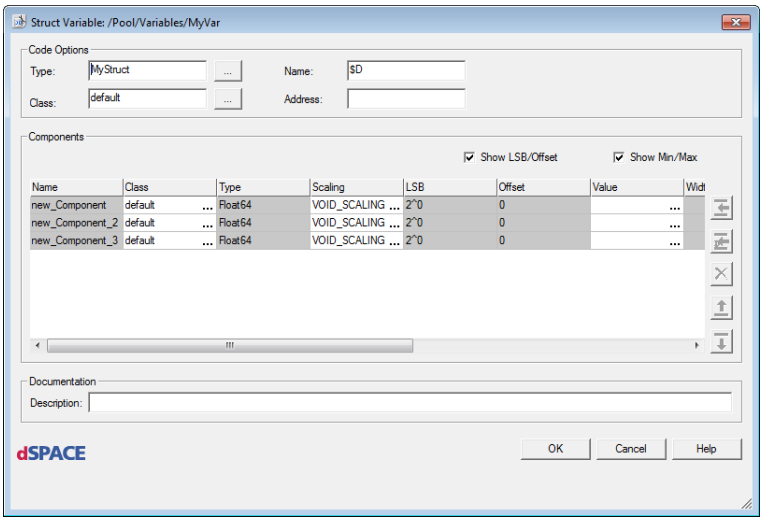


Changing the type of the variable requires transforming the underlying data accordingly.

4 Click OK. TargetLink transforms the plain variable into a structured variable.

The Plain Variable dialog automatically changes to the Struct Variable dialog.

The dialog shows the struct components as defined by the explicitly specified structured data type:



5 Specify each component's **Class**, **Scaling**, **Value**, **Min**, or **Max** as required and click **OK**.

> **Note**
>
> The **Name** and **Type** properties of the components are specified at the structured data type that you specified explicitly and referenced at your structured variable.
> These settings are propagated to the components of the structured variable, whenever you open the variable's **Struct Variable** dialog.
> Changes made to the **Name** or **Type** of the components in the `/Pool/Variables/MyVar/Components` subtree will be overwritten.

6 If you want TargetLink to generate the variable's definition into a separate module, specify a **Module** object and reference it at the **Variable** object's **ModuleRef** property.

> **Note**
>
> Do not specify the **ModuleRef** property of the structured variable's components.

Result

You created a structured variable based on an explicitly specified structured data type. You can now reference one or more of its component(s) at one or more TargetLink blocks.

TargetLink will generate definitions of the variable/typedef such as the following:

Definition of Structured Data Type	Definition of Structured Variable

`struct MyStruct_tag {`
`  Int16 FirstComponent;`
`  Int16 SecondComponent;`
`  Int16 ThirdComponent;`
`};`
	`struct MyStruct_tag MyStructVar;`

Related topics

Basics

Basics on Arranging Data in Structured Variables..... 148

HowTos

How to Create Implicitly Specified Structured Variables..... 158
How to Create New Type Definitions..... 134
How to Explicitly Specify Structured Data Types..... 152

Examples

Example of Collecting Gain Parameters via Dynamic Components..... 161

References

Create <DD Object> (📖 TargetLink Data Dictionary Manager Reference)

How to Create Implicitly Specified Structured Variables

Objective

To specify a structured variable based on a structured data type that is implicitly created by the Code Generator.

The variable's components can be referenced at TargetLink blocks.

Implicit structs

Implicit structs are called *implicit* because you do not specify the structured data type. Instead the structured data type is implicitly created by the Code Generator. Implicit structs result from DD **Variable** objects whose **Type** property is set to `IMPLICIT_STRUCT`.

Method

To create an implicitly specified structured variable

1 Create a new DD Variable object in the `/Pool/Variables` subtree and rename it as required.

> **Note**
>
> The variable's name must be a valid C identifier.

2 Double-click the variable object to open the Plain Variable dialog.

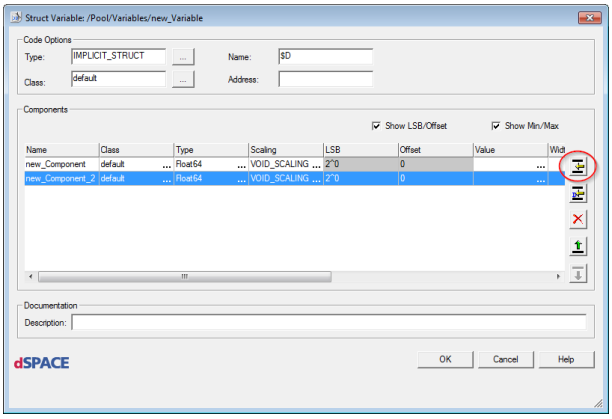
3 In the Type edit field, reference the Typedef object named `IMPLICIT_STRUCT`.

Changing the type of the variable requires transforming the underlying data accordingly.

4 Click OK. TargetLink transforms the plain variable into a structured variable.

The Plain Variable dialog automatically changes to the Struct Variable dialog.

5 In the Struct Variable dialog, add one or more component(s) to the structured variable via the Add component button.



Each component represents a leaf struct component.

6 For each component, adjust the Name, Class, and Type columns as required.

> **Tip**
>
> To create a nested struct, reference another struct via the Type property.

7 Click OK to close the dialog.

The struct variable is created as specified together with its components.

Result

You created a structured variable whose structured data type is implicitly created by the Code Generator. You can now reference on or more of its component(s) at one or more TargetLink blocks.

TargetLink will generate definitions of the variable/typedef such as the following:

Definition of Structured Data Type	Definition of Structured Variable

`struct tag_MyStructVar {`
`  Int16 FirstComponent;`
`  Int16 SecondComponent;`
`  Int16 ThirdComponent;`
`}; /* struct type created for variable: MyStructVar */`
	`struct tag_MyStructVar MyStructVar;`

Related topics

Basics

Basics on Arranging Data in Structured Variables..... 148

HowTos

How to Create Explicitly Specified Structured Variables..... 155
How to Create New Type Definitions..... 134
How to Create User-Defined Data Types..... 139
How to Explicitly Specify Structured Data Types..... 152

Examples

Example of Collecting Gain Parameters via Dynamic Components..... 161

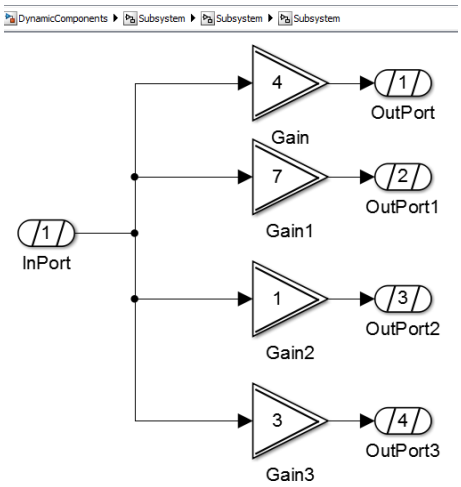
Example of Collecting Gain Parameters via Dynamic Components

Introduction

To collect gain parameters as struct components of an implicit struct by using dynamic components.

Model overview

This example makes use of a simple model containing four **Gain** blocks:



Dynamic components

A **DynamicComponent** object is a placeholder that you can reference at different blocks.

During code generation, a struct component is implicitly created in the production code for each reference of the **DynamicComponent** object. This lets you implement one or more block variable(s) as components of the implicit struct.

The properties of each component depend on the block data that is relevant for the Code Generator: e.g., the block's **Output** page for an output variable, or a block's **State** page for a state variable.

The order of the struct's components depends on the criteria specified in the **DynamicComponent** object's **SortBy** property.

Precondition

You created a DD **Variable** object whose **Type** property is set to `IMPLICIT_STRUCT`.

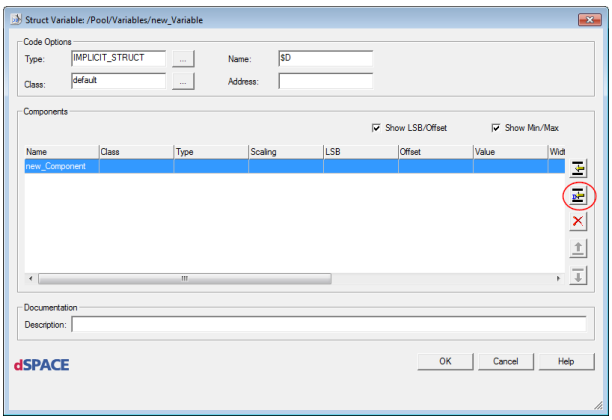
For details, refer to How to Create Implicitly Specified Structured Variables on page 158.

Method

To collect gain parameters via a dynamic component

1 Double-click a DD Variable object whose Type property is set to `IMPLICIT_STRUCT` to open the Struct Variable dialog.

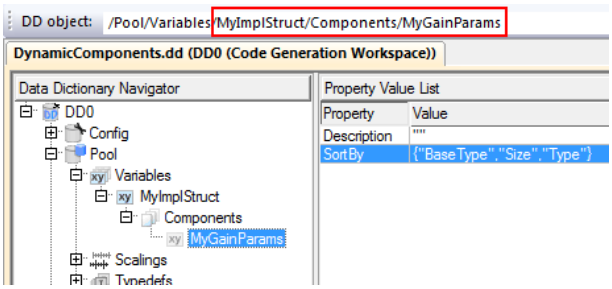
2 In the Struct Variable dialog, add a dynamic component to the structured variable via the Add dynamic component button.



3 Adjust the Name and SortBy columns as required.

4 Click OK to close the dialog.

The struct variable is modified and now contains your dynamic component:



5 Reference the dynamic component at all the Gain blocks of your model via the API:

```
hgain = tl_get_blocks('DynamicComponents','TL_Gain');
tl_set(hGain, 'gain.variable', 'MyImplStruct/Components/MyGainPara
ms');
```

6 Generate code.

Result

You added a dynamic component to an implicit struct to collect all the gain parameters of all the Gain blocks of your model.

During code generation, the Code Generator creates a struct definition such as the example below and uses it to implement the gain parameters in the generated code:

```
struct tag_MyImplStruct {
   Int16 Gain;
   Int16 Gain1;
   Int16 Gain2;
   Int16 Gain3;
}; /* struct type created for variable: MyImplStruct */
```

Related topics

HowTos

How to Create Explicitly Specified Structured Variables..... 155
How to Create Implicitly Specified Structured Variables..... 158
How to Explicitly Specify Structured Data Types..... 152

References

Edit StringList (📖 TargetLink Data Dictionary Manager Reference)
tl_get_blocks (📖 TargetLink API Reference)
tl_set (📖 TargetLink API Reference)

Structuring Production Code via Pointers to Structs

Introduction

You can specify pointers to structured variables (structs) as formal parameters in function signatures.

Where to go from here

Information in this section

Basics on Specifying Pointers to Structs..... 164

How to Create Pointers to Predefined Structs..... 167

How to Create Pointers to Predefined Structs Using Bus Port Blocks..... 170

How to Add Pointers to Structs to Function Signatures Manually..... 171

How to Create Pointers to Implicit Structs Using Bus Port Blocks..... 174

How to Specify Names for Structured Implicit Types..... 176

Basics on Specifying Pointers to Structs

Benefits of pointers to structured variables

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.

Struct components which are passed to functions using pointers to structs can be used for function input and output, internal block variables, and also for parameters, state variables, and Stateflow variables.

Note

For pointers, you can specify the variable classes of the formal and the actual parameter separately. The variable class of the formal parameter is specified using the `Class` attribute. The implementation of the actual parameter of a function is controlled by the `ArgClass` variable class attribute. Refer to Example of Specifying Block Parameters as Input Arguments of Functions on page 114.

Specifying pointers to structs

You can use the following methods to have TargetLink generate pointers to structs in function signatures:

- You can create the struct in the DD using a variable class with the *ref_param* scope, for example `FCN_REF_ARG`, and then reference its components from the block dialogs of your model.
- You can have TargetLink generate pointers to implicit structs using TargetLink's **Bus Inport** and **Bus Outport** block (called bus port blocks below).

Referencing a predefined struct in the DD

Suppose you want to interface to an external function which has a pointer to a struct as a formal parameter, or you want to group, for example, inputs, outputs, and parameters of a block in one struct. You can perform the following process steps to have TargetLink create a pointer to the struct in the production code:

1. Create the struct in the DD.
2. Assign a variable class with the ref_param scope to the struct.
3. Reference the struct components from the relevant block signals of your model.

During code generation, TargetLink adds a pointer to the struct as a formal parameter to the signature of each function generated from a subsystem where at least one struct component is used. For details, refer to How to Create Pointers to Predefined Structs on page 167.

Referencing a predefined struct using bus port blocks

You can map buses to predefined structs in the DD and have TargetLink create a pointer to each struct using bus port blocks. Refer to How to Create Pointers to Predefined Structs Using Bus Port Blocks on page 170.

Adding pointers to structs to function signatures manually

Basically, TargetLink adds pointers to structs only to function signatures which are generated from subsystems in which one or more struct components are used. However, you can have TargetLink add a struct pointer to function signatures whose associated subsystems do not reference any struct components. You have to specify additional parameters in the TargetLink **Function** block for this. Refer to How to Add Pointers to Structs to Function Signatures Manually on page 171.

Note

If you work with pointers to structs which are predefined in the DD, you have to note the following points:

- If two or more block signals reference the same struct component, the `MERGEABLE` optimization attribute of the struct component's variable class must be set.
- TargetLink does not support arrays of structs.
- You must not specify dynamic struct components. For details, refer to Arranging Data in Structured Variables on page 148.
- You can assign the actual parameter of the struct to a code file. If you do not, the struct is generated in the code file in which it is used for the first time. You can specify the code file on the Options page of the **Function** block. The **Function** block also lets you reference a Data Dictionary module object (`/Pool/Modules`). The reference overtrumps and disables the **C code file name** option. Refer to the **Options** page of the **Function Block**.

Creating pointers to implicit structs using bus port blocks

Suppose a bus signal is connected to a subsystem which is mapped to a separate function. TargetLink provides a method to map the bus signal to an implicit struct and generate a pointer to that struct in the relevant function signature. Refer to How to Create Pointers to Implicit Structs Using Bus Port Blocks on page 174.

Note

If you work with pointers to implicit structs using bus port blocks, you have to note the following points:

- You can specify a variable class with the ref_param scope only for the root bus signal.
- TargetLink usually creates one actual struct variable for each bus port block. In the case of a one-to-one connection between bus port blocks, TargetLink tries to use the struct generated for the preceding bus port block without generating a new struct variable.

Specifying names

TargetLink provides a method to specify the names of structured types and of pointers to structs.

Names of structured types TargetLink usually assigns names for the structured types during production code generation, but it also provides a method for you to specify a name for a structured type. Refer to How to Specify Names for Structured Implicit Types on page 176.

Names of pointers to structs You can specify the name via the Name edit field in the block dialogs, or the Name property in the Data Dictionary Manager. The name is used for the function's formal parameter. The name of the actual parameter, is derived from the specification of the formal parameter by default. If required, you can specify the name of the actual parameter in the Name field after the name of the formal parameter, separated by a colon. You can use TargetLink's name macros to configure the names. Refer to Generating Unique Names via Name Macros on page 78.

Demo model

TargetLink comprises a ready-to-use example model for pointers to structs. To open it, type **`tl_demos`** in the MATLAB Command Window, and double-click the `STRUCT_INTERFACES` model.

Related topics

Basics

Arranging Data in Structured Variables..... 148

References

Function Block (📖 TargetLink Model Element Reference)

How to Create Pointers to Predefined Structs

Objective

To structure production code concisely, you can, for example, group block inputs and outputs, and block parameters, in a struct and have TargetLink generate a pointer to that struct in the signature of the relevant functions.

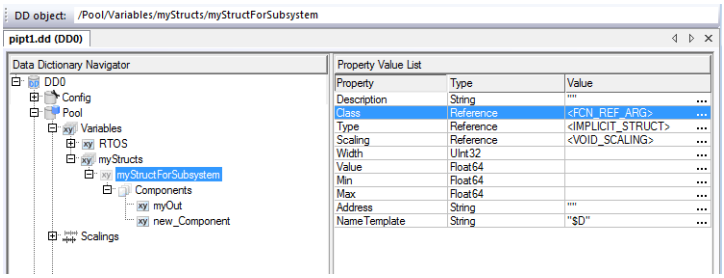
Preconditions

- A TargetLink model containing a subsystem with a **Function** block is open.
- The Data Dictionary Manager is open.

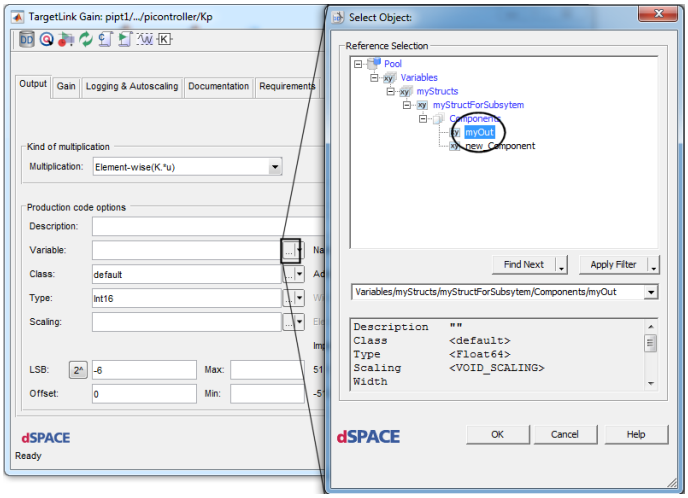
Method

To create pointers to predefined structs

1. In the Data Dictionary Manager under `/Pool/Variables`, create the struct and all its components (for details, refer to Arranging Data in Structured Variables on page 148).
2. In the property value list of the Data Dictionary Manager, click the *Class* browse button and select a variable class with the **ref_param** scope, for example, select **FCN_REF_ARG**.



3. In your model, open the subsystem that is to contain a signature with a pointer to a struct. The subsystem must contain a **Function** block.
4. Open the block dialogs containing the variables you want to use as components in the structure that is passed, and click the *Variable* browse buttons to reference the struct components created in the DD.



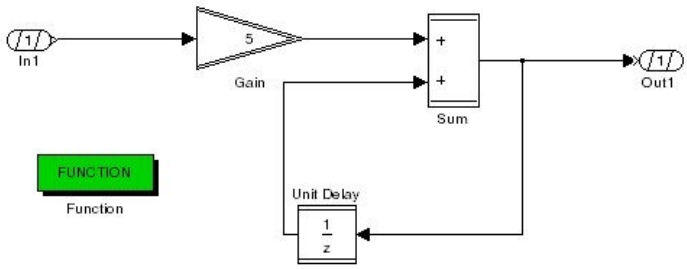
5 Generate production code.

Result

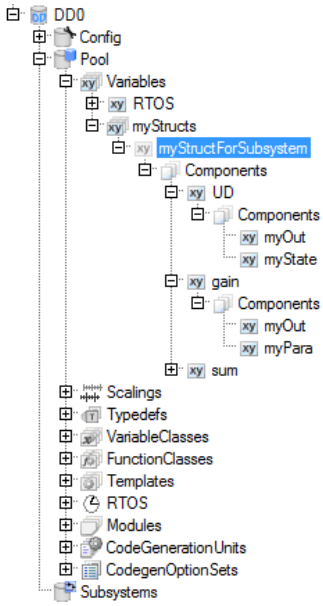
TargetLink creates the struct variable in the relevant header file. It also adds a pointer to the struct as a formal parameter to the signature of the function created from the subsystem.

Example

Consider the following example model:



The blocks in this example reference the `myStructForSubsystem` struct, which is created in the DD with the `FCN_REF_ARG` variable class.



The struct components are referenced by the blocks in the model. Since `Subsystem` contains a Function block, TargetLink converts it into a separate C function during code generation. TargetLink adds a pointer to the `myStructForSubsystem` struct as a formal parameter to the signature of the function:

```
…
static void Sa2_Subsystem (MY_STRUCT_TYPE * myStructForSubsystem)
{
   /* Gain: Subsystem/Subsystem/Gain */
   myStructForSubsystem->gain.myOut = ((Int32) Sa2_InPort)
    * ((Int32)myStructForSubsystem->gain.myPara);
   /* Unit delay: Subsystem/Subsystem/Unit Delay */
   myStructForSubsystem->UD.myOut = myStructForSubsystem->UD.myState;
   /* Sum: Subsystem/Subsystem/Sum */
   myStructForSubsystem->sum =
    (Int16) ((UInt16) (Int16) (myStructForSubsystem->gain.myOut  >> 13))
    + ((UInt16) myStructForSubsystem->UD.myState));
   /* Unit delay: Subsystem/Subsystem/Unit Delay */
   myStructForSubsystem->UD.myState = myStructForSubsystem->sum;
}
#endif/*_SUBSYSTEM_C_ */
…
```

Related topics

Basics

Arranging Data in Structured Variables..... 148

How to Create Pointers to Predefined Structs Using Bus Port Blocks

Objective

To create pointers to predefined structs from using bus port blocks, you have to perform the following steps:

1. Create the structured variable with a variable class with the **ref_param** scope in the DD.
2. Map the elements of a bus signal to the struct components.

Preconditions

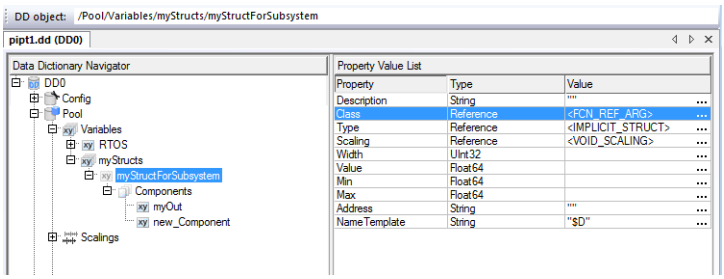
- A TargetLink model containing a bus signal connected to a subsystem via a bus port block is open.
- The Data Dictionary Manager is open.

Method

To create pointers to predefined structs using bus port blocks

1 In the Data Dictionary Manager under `/Pool/Variables`, create the struct and all its components (for details, refer to Arranging Data in Structured Variables on page 148).

2 Select the struct variable you created, click the **Class** browse button in the **Property Value List** of the Data Dictionary Manager and select a variable class with the **ref_param** scope, for example, select **FCN_REF_ARG**.



3 In your model, double-click the bus port block to open the **Output** page of the block dialog.

4 Expand the bus hierarchy by double-clicking the buses in the **bus hierarchy**.

5 Select **Apply explicit struct**.

6 Select a predefined variable for the bus root.

7 Generate production code.

Result

You have assigned the bus elements to the components of a predefined struct. During production code generation, TargetLink generates a separate C function for the subsystem. Additionally, it adds a pointer to the predefined struct variable to the function signature.

Example

The following code excerpt results from the example shown in the above steps:

```
…
typedef struct StructTag {
   Float64 UD;
   Float64 gain;
   Float64 sum;
} MY_STRUCT_TYPE;
…

void FcnBusInterfacePreselected(MY_STRUCT_TYPE * MyStructForSubsystem)
{
   /* SLLocal: Default storage class for local variables | Width: 64 */
   Float64 Sa1_Sum;

   /*SLStaticLocalInit: Default storage class for static local variables
    with initvalue | Width: 16 */
   static Int16 X_Sa1_Unit_Delay1 = 0;
   static Int16 X_Sa1_Unit_Delay2 = 0;

   /* Sum: Subsystem/Sum1 */
   Sa1_Sum1 = (MyStructForSubsystem->UD * 2.) + ((Float64) X_Sa1_Unit_Delay1);

   /* TargetLink outport: Subsystem/OutPort */
   Sa1_OutPort = ((Float64) X_Sa1_Unit_Delay2) + (MyStructForSubsystem->sum
    * 5.5);

   /* Unit delay: Subsystem/Unit Delay1 */
   X_Sa1_Unit_Delay1 = (Int16) (MyStructForSubsystem->gain * 3.5);

   /* Unit delay: Subsystem/Unit Delay2  */
   X_Sa1_Unit_Delay2 = (Int16) Sa1_Sum1;
}
```

Related topics

Basics

Arranging Data in Structured Variables..... 148

HowTos

How to Create New Data Dictionary Objects (📖 TargetLink Data Dictionary Basic Concepts Guide)

How to Add Pointers to Structs to Function Signatures Manually

Objective

Basically, TargetLink adds pointers to structs only to function signatures which are generated from subsystems in which one or more struct components are referenced. However, you can have TargetLink generate a pointer to a struct to function signatures whose associated subsystems do not reference any struct

components. This is useful, for example, if you want to pass a pointer to a struct in nested subsystems.

Preconditions

- A TargetLink model containing a subsystem with a Function block is open.
- The struct is created in the DD. It has a variable class with the ref_param scope, for example, FCN_REF_ARG.

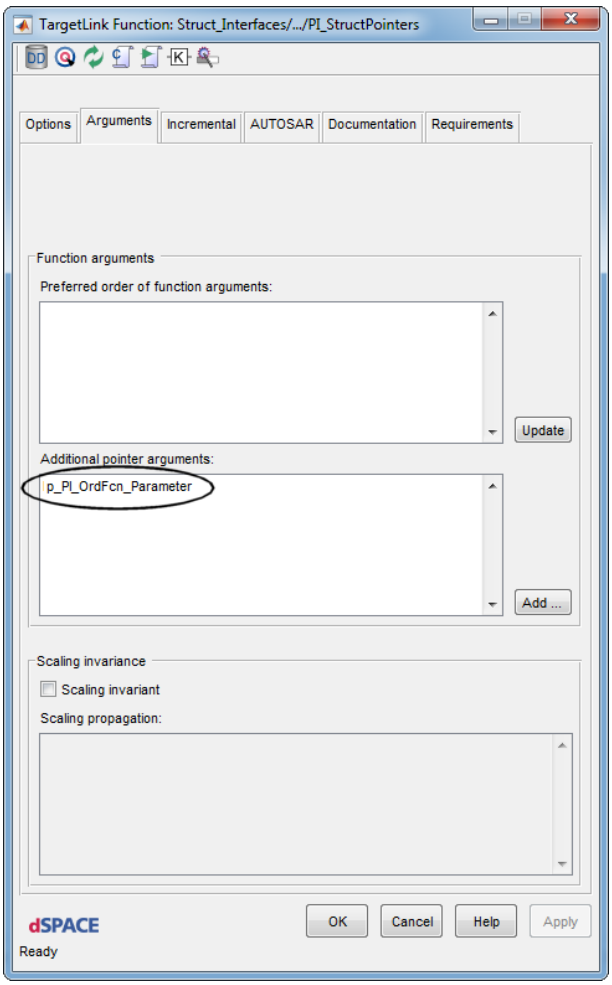
Method

To add pointers to structs to function signatures manually

1 Double-click the Function block to open the Function block dialog.

2 Change to the Arguments page and click Add. The Select Object dialog opens.

3 In the Select Object dialog, select the struct variable to add it as an additional pointer argument and click OK. The selected struct appears in the Additional pointer arguments list.



Tip

You can specify the sort sequence of the additional pointer arguments by modifying the order of the function arguments in the *Preferred order of function arguments* field. For details, refer to the Arguments page of the Function Block block.

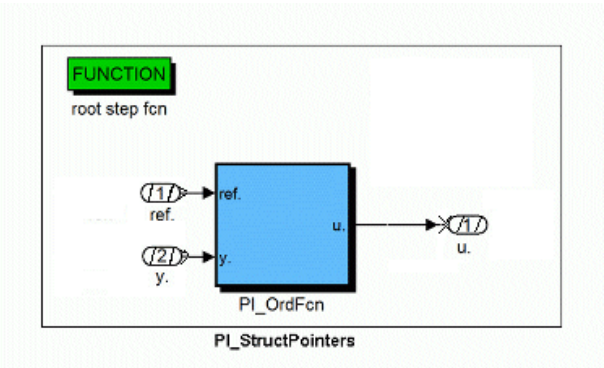
4 Generate production code.

Result

During production code generation, TargetLink adds a pointer to the selected struct variable as a formal parameter to the function signature of the subsystem.

Example

Consider the following example model.



In the Function block, the `p_PI_OrdFcn_Parameter` struct is specified as an additional pointer argument for the `PI_StructPointers` subsystem. During production code generation, TargetLink converts the subsystem into a separate C function and adds a pointer to the specified struct to the function signature:

```
…
typedef struct PI_OrdFcn_Param_Type_tag {
   Float64 PP /* Description: Proportional Parameter */;
   Float64 PI /* Description: Integral Parameter */;
} PI_OrdFcn_Param_Type; /* Struct containing the parameters PP and PI of the PI
  controller */
…

void PI_StructPointers(PI_Interface_Type * StructPointer_Interface,
PI_OrdFCN_Param_Type *p_PI_OrdFcn_Parameter)
{
   /* call of function: PI_StructPointers/PI_OrdFcn */
   StructPointer_Interface->Output = PI_Step_Interior(StructPointer_Interface,
    p_PI_OrdFcn_Parameter);
}
…
```

Related topics

References

Function Block (📖 TargetLink Model Element Reference)

How to Create Pointers to Implicit Structs Using Bus Port Blocks

Objective

If you want TargetLink to create an implicit struct from a bus signal pending at a subsystem and to create a pointer to that struct in the function signature of the subsystem, you have to configure the bus signal in the bus port block dialogs accordingly.

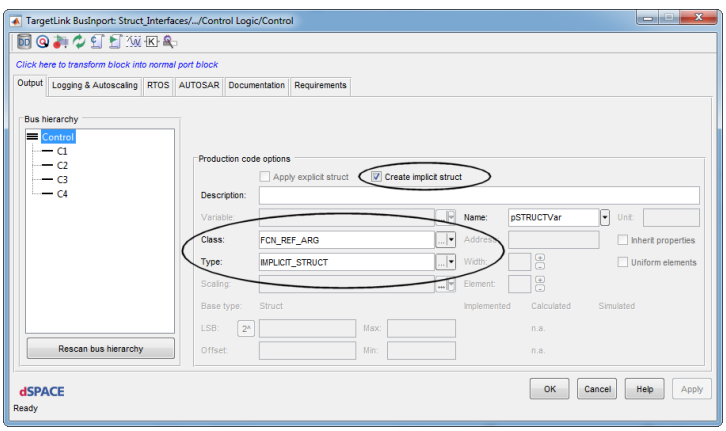
Precondition

A TargetLink model containing a subsystem with a Function block and a bus signal at its inport is open.

Method

To create pointers to implicit structs using bus port blocks

1. Open the bus port block dialog.
2. Under Bus hierarchy, select the root bus signal.
3. Select the Create implicit struct checkbox.
4. Under Production code options, click the Class browse button and select a variable class with the ref_param scope, for example, select `FCN_REF_ARG`.
5. Click the Type browse button and select `IMPLICIT_STRUCT`.



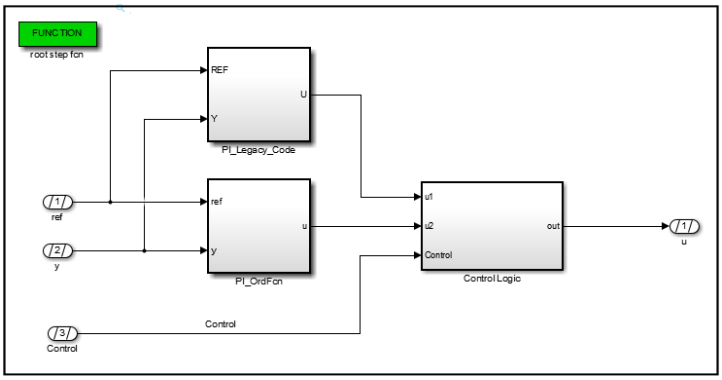
6. Select Create implicit struct and IMPLICIT_STRUCT for all subbuses you want TargetLink to create substructures for.
7. Click Apply.

Result

During production code generation, TargetLink generates an implicit struct for the root bus in the header file. Additionally, TargetLink generates a pointer to the struct in the function signature resulting from the subsystem.

Example

Consider the following example model:



Below is an excerpt of the code TargetLink generates if you specify the settings shown in step 5 for the **Bus Inport** block dialog:

- Refer to `<DocumentsFolder>\dSPACE\TargetLink\<Version>\Demos\struct_interfaces\TLProj\PI_StructPointers\udt_f.h`:

```
struct BS_IP_Sf1_Control {
   Float64 C1;
   Float64 C2;
   Float64 C3;
   Float64 C4;
}; /* bus inport struct */
```

- Refer to `<DocumentsFolder>\dSPACE\TargetLink\<Version>\Demos\struct_interfaces\TLProj\PI_StructPointers\PI_StructPointers.c`:

```
/*****************************************************************
************\
   STATIC_FCN: static function(s) (invisible for other module)
\*****************************************************************
************/
static Float64 ControlFcn(Float64 Sf2_u1, Float64
Sf2_u2, struct BS_IP_Sf1_Control * Control);
```

Related topics

Basics

Customizing the Representation of Buses in Production Code..... 467

How to Specify Names for Structured Implicit Types

Objective

Usually, TargetLink generates names for structured types using name macros. However, you can also specify a name for a structured type in the DD.

Preconditions

- A TargetLink model is open.
- The Data Dictionary Manager is open.
- The relevant DD project file (`<name>.dd`) is open.

Method

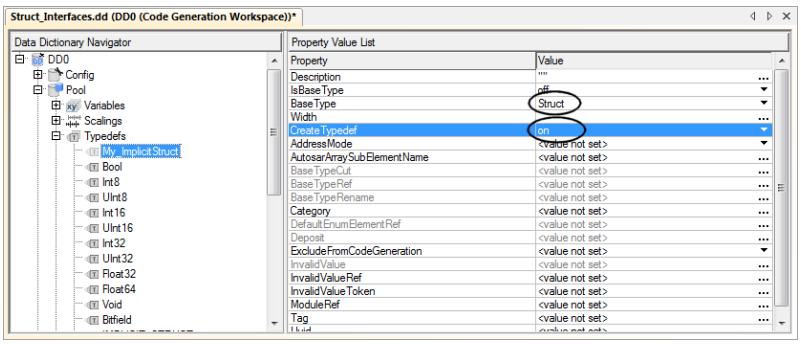
To specify a name for structured types

1 In the Data Dictionary Manager, change to `/Pool/Typedefs` and create a new typedef (refer to How to Create New Data Dictionary Objects (📖 TargetLink Data Dictionary Basic Concepts Guide)).

> **Note**
>
> The typedef must be empty. You must not specify any structure components for it.

2 Double-click the new typedef and enter a name, for example, `MY_IMPLICIT_STRUCT`.

3 In the Property Value List, specify the settings for the new typedef as shown in the illustration below:



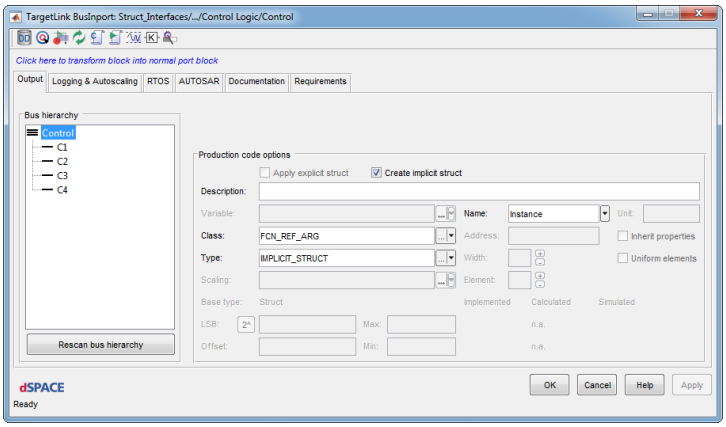
4 In your model, open the Output page of a bus port block dialog.

5 On the Output page, select the Create implicit struct checkbox and the FCN_REF_ARG variable class.

6 Click the Type browse button and select the MY_IMPLICIT_STRUCT typedef in the Select Object dialog.

7 Enter a name for the structured variable in the Name edit field, for example, `Instance` (optional).

The following illustration shows the settings specified in the bus port block dialog:



8 Generate production code.

Result

During production code generation, TargetLink generates a structured type with the name specified in the DD and referenced in the bus port block dialog.

Example

If you specify the settings according to the above steps, TargetLink generates the following production code:

```
typedef struct MY_IMPLICIT_STRUCT_tag {
Int32 element_k;
Int32 element_l;
Int32 element_m;
Int32 element_n;
} MY_IMPLICIT_STRUCT;
…

/*****************************************************************************\
   GLOBAL_FCN: global function(s) (exported to other modules)
\*****************************************************************************/
extern void PI_StructPointers(MY_IMPLICIT_STRUCT *Instance);
```

Customizing Production Code Generation via Hook Scripts

Introduction

TargetLink allows you to intervene at different steps of the code generation process using hook scripts.

Basics on Using Hook Scripts

Hook scripts

TargetLink provides several hook script templates that you can use to derive M-script hook scripts. These are used as a callback mechanism for user-specific customization of certain TargetLink processes.

Hook scripts are called at specific points during the process, for example, to add your own commands before and after the code generation, compilation or download process. In addition, there are hook scripts you can use to customize system preparation and clearance, and hook scripts to customize file export and tailor the document generation process. Hence, hook scripts are a powerful mechanism that allows you to adapt TargetLink in a tool chain for your company's development process.

Deriving hook scripts from templates

Use TargetLink's `tlCustomizationFiles('Create',...)` API function or the Create Customization Files dialog to derive customization files from their templates. You can open the dialog by executing the `tlCustomizationFiles` command without arguments. Refer to Deriving Customization Files From Their Templates on page 33.

Tip

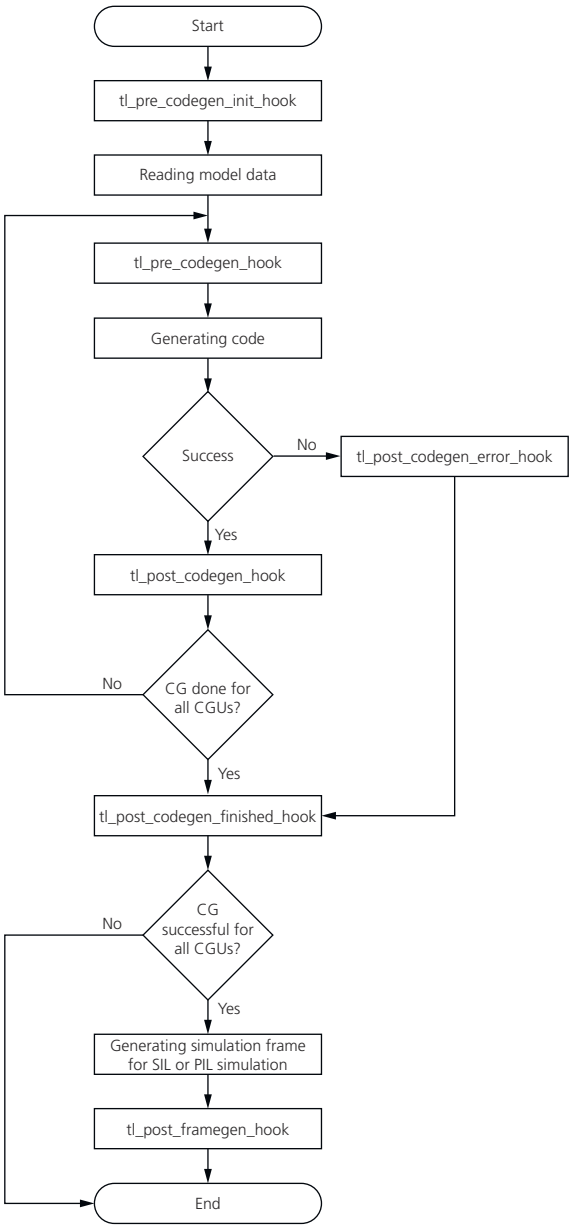
The **`tlFindHook`** hook script lets you find all hook scripts of a certain kind, e.g., *pre_codegen_hook*, located on TargetLink's search path.

Code generation

The code generation process can be customized by using the following hook scripts:

- `tl_pre_codegen_init_hook`
- `tl_pre_codegen_hook`
- `tl_post_codegen_hook`
- `tl_post_codegen_finished_hook`
- `tl_post_codegen_error_hook`
- `tl_post_framegen_hook`

These hook scripts are evaluated by the `tl_generate_code` API function (associated with the **Generate Code** button in the **TargetLink Main Dialog** block) as shown in the following flow chart:.



Note

If you use API functions in the tl_pre_codegen_hook hook script, keep in mind that this hook script is called just before the start of the Code Generator, but after the model was initialized by Simulink. For this reason, using API functions that change the Simulink model or block data can lead to unexpected or even incorrect results when generating production code.

Do not use the `tl_pre_codegen_hook` hook script to make changes to the model, such as:

- Changes to the Simulink structure and to Simulink model properties (e.g., adding or deleting blocks)
- Changes to the signal widths
- Changes to the signal labels

To modify the model before code generation, you can derive the `tl_pre_codegen_init_hook` hook script from its template.

Compilation

To customize the compilation process for the simulation application in SIL simulation mode, you can use the following hook scripts. These hook scripts are called immediately before or after the simulation application is compiled via an API function like `tl_compile_host` or an associated button such as **Compile SIL**:

- `tl_pre_compile_host_hook`
- `tl_pre_compile_target_hook`

To customize the compilation process for the simulation application in PIL simulation mode, you can use the following hook scripts. These hook scripts are called immediately before or after the simulation application is compiled via an API function like `tl_compile_target` or an associated button such as **Compile PIL**:

- `tl_pre_compile_target_hook`
- `tl_post_compile_target_hook`

Download

To customize the download process for the simulation application in PIL simulation mode, you can use the following hook scripts. These hook scripts are called immediately before or after the simulation application is downloaded to the evaluation board(physical or virtual) by an API function like `tl_download` or an associated button such as **Download**:

- `tl_pre_download_hook`
- `tl_post_download_hook`

Model preparation

To customize the system preparation/clearance, you can use the following hook scripts. These scripts are called before the first block or after the last block is prepared or cleared.

- `tl_pre_preparation_hook`
- `tl_post_preparation_hook`

- `tl_pre_clear_hook`
- `tl_post_clear_hook`

For detailed information on all the available hook script templates, refer to Hook Scripts (🕮 TargetLink File Reference).

Example

Suppose you want to insert a company-specific header at the beginning of the source code file generated for a TargetLink subsystem. Include the following commands in the `tl_post_codegen_hook`:

```
%read generated source file
[fid,msg] = fopen('.\TLProj\picontroller\picontroller.c','rt');
fileContent = fscanf(fid,'%c',inf);
fclose(fid);


% open file for writing and insert header text
disp(['Inserting header text into ' '.\TLProj\\picontroller\picontroller.c']);
[fid,msg] = fopen('.\TLProj\picontroller\picontroller.c','wt');
fprintf(fid,[...
  '/*****************************************************************\n', ...
 '**  Proprietary header text entered via TL_POST_CODEGEN_HOOK  **\n', ...
 '*****************************************************************/\n\n']);
fprintf(fid,'%s',fileContent);
fclose(fid);
```

Related topics

Basics

Basics on the Download Process (📖 TargetLink Preparation and Simulation Guide)
Basics on Using Hook Scripts for Customization (AUTOSAR) (📖 TargetLink Interoperation and Exchange Guide)
Deriving Customization Files From Their Templates..... 33

HowTos

How to Create Customization Files via the Create Customization Files Dialog..... 38
How to Define TargetLink's Search Path for M-API-Related Customization Files..... 36

References

Create Customization Files Dialog Description (📖 TargetLink Tool and Utility Reference)
Hook Scripts (📖 TargetLink File Reference)
tl_compile_host (📖 TargetLink API Reference)
tl_compile_target (📖 TargetLink API Reference)
tl_download (📖 TargetLink API Reference)
tl_generate_code (📖 TargetLink API Reference)
tl_post_clear_hook (📖 TargetLink File Reference)
tl_post_codegen_error_hook (📖 TargetLink File Reference)
tl_post_codegen_finished_hook (📖 TargetLink File Reference)
tl_post_codegen_hook (📖 TargetLink File Reference)
tl_post_compile_target_hook (📖 TargetLink File Reference)
tl_post_download_hook (📖 TargetLink File Reference)
tl_post_framegen_hook (📖 TargetLink File Reference)
tl_post_preparation_hook (📖 TargetLink File Reference)
tl_pre_clear_hook (📖 TargetLink File Reference)
tl_pre_codegen_hook (📖 TargetLink File Reference)
tl_pre_codegen_init_hook (📖 TargetLink File Reference)
tl_pre_compile_host_hook (📖 TargetLink File Reference)
tl_pre_compile_target_hook (📖 TargetLink File Reference)
tl_pre_download_hook (📖 TargetLink File Reference)
tl_pre_preparation_hook (📖 TargetLink File Reference)
tlCustomizationFiles (📖 TargetLink API Reference)

Decorating Production Code

Where to go from here

Information in this section

Basics on Code Decorations..... 183

How to Specify Code Decorations in the Data Dictionary..... 186

Example of Mapping Variables to Memory Sections via Code Decorations..... 187

Example of Assigning Variables to Memory Sections When Using a Target Optimization Module..... 193

Basics on Code Decorations

Introduction

TargetLink lets you decorate variables and functions in production code with user-specified strings such as pragmas. You can use decorations to place pragmas or compiler instructions in the generated production code, for example, to perform memory mapping.

Controlling the generation of code decorations

You can control whether TargetLink creates code decorations as follows:

- Via the EnableSections Code Generator option
- Via the corresponding Enable sections/pragmas/inline/ISR/user attributes checkbox on the Code Generation page of the TargetLink Main Dialog Block

If you use code decorations to specify special statements like non-pragma section statements of a cross compiler, you can disable the generation of code decoration to facilitate SIL simulation if the SIL compiler does not support these decorations. You have to repeat the code generation after changing the option's value.

Objects in the Data Dictionary

You specify code decorations in the Data Dictionary in the `/Pool/CodeDecorationSets` subtree via CodeDecorationSet and CodeDecoration objects.

CodeDecorationSet objects You reference DD CodeDecorationSet at DD VariableClass and DD FunctionClass objects. This lets you specify decorations for variables and functions that belong to the same class.

Each CodeDecorationSet can contain one ore more CodeDecoration objects.

CodeDecoration objects Each DD CodeDecoration object specifies a specific code decoration. This decoration is applied to code elements that match the filter of the CodeDecoration object.

Specifying the filter of the object

You specify the filter of each DD CodeDecoration object via the properties in its Filter subtree.

Via the IntendedFor property, you can select which code elements to decorate, functions or variables.

Additional filter criteria You can specify additional filter criteria, as shown in following table:

Property	Description
Const	Filters by the constness of the variable.
DataTypeSpec	Filters by the base data type of the variable.
DimensionSpec	Filters by the dimension of the variable.
InitializationSpec	Filters by the initialization of the variable.
Macro	Filters by the macro property of the function or variable.
ScopeSpec	Filters by the scope and storage of the variable.
WidthSpec	Filters by the bit width of the data type of the variable.

Specifying the decoration

You specify the decoration applied by each DD CodeDecoration object via the properties in its Settings subtree.

The following table shows possible use cases for the key properties:

Property	Use Case	Examples
DeclarationStatements	Specifying one of the following: ▪ A decoration inserted before or after a code elements declaration or definition. ▪ A decoration before or after a code block collecting the declarations or definitions of familiar code elements.	Example of Mapping Variables to Memory Sections via Code Decorations on page 187
SectionName	Specifying a memory section for use with one of TargetLink's Target Optimization Modules (TOM).	Example of Assigning Variables to Memory Sections When Using a Target Optimization Module on page 193
	Specifying a memory section in AUTOSAR code generation mode.	Example of Adapting the AR_POSCONTROL Demo Model to use Memory Sections (Classic AUTOSAR) (📖 TargetLink Classic AUTOSAR Modeling Guide)

Decorating TargetLink's internal default classes

You can decorate TargetLink's internal default classes by referencing a CodeDecorationSet object at the DefaultDecorationSetRef property of the `/Config/TargetLink` object.

If you additionally set its **AlwaysUseDefaultDecorationSet** property to **on**, TargetLink also uses this **CodeDecorationSet** object for the following variables and functions:

- Implicitly generated functions and variables with classes derived from internal default classes via DD **FunctionClassTemplate** or **VariableClassTemplate** objects
- Functions and variables associated with classes explicitly defined in the Data Dictionary that do not reference a **CodeDecorationSet** themselves.

If you want to distinguish the decoration of internal default classes, you can combine the use of DD **CodeDecorationSet**, **FunctionClassTemplate**, and **VariableClassTemplate** objects. Refer to Controlling Objects and Properties Created by the Code Generator (Templates) on page 203.

Example

The following code is taken from the POSCONTROL demo model with a code decoration with the `PreDefinitionStatement` part of the DeclarationStatements property set to **`#pragma section = "CODE_$F"`** and the `PostDefinitionStatement` part set to **`#pragma section restore`**:

```
/*****************************************************************************\
 ***  FUNCTION:
 ***      Controller
 ***
 ***  DESCRIPTION:
 ***      controller function
 ***
 ***  PARAMETERS:
 ***      Type                Name                Description
 ***      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 ***
 ***  RETURNS:
 ***      void
 ***
 ***  SETTINGS:
 ***
\*****************************************************************************/
#pragma section = "CODE_CONTROLLER"
void Controller(void)
{
   /* SLLocal: Default storage class for local variables | Width: 16 */
   Int16 SCtrl1_Ki; /* LSB: 2^-9 OFF: 0 MIN/MAX: -64 .. 63.998046875 */
   Int16 SCtrl1_Kp; /* LSB: 2^-9 OFF: 0 MIN/MAX: -64 .. 63.998046875 */
   Int16 SCtrl1_e; /* LSB: 2^-10 OFF: 0 MIN/MAX: -32 .. 31.9990234375 */
   Int16 SCtrl1_si; /* LSB: 2^-9 OFF: 0 MIN/MAX: -64 .. 63.998046875 */
   /* call of function: controller/Linearization */
   Linearization();
   /* Sum: controller/e */
   SCtrl1_e = (Int16) (Reference - LinPosition);
   /* Gain: controller/Kp */
   SCtrl1_Kp = SCtrl1_e;
   /* Sum: controller/si */
   SCtrl1_si = (Int16) (((Int16) (SCtrl1_e >> 1)) + X_SCtrl1_xi);
   /* Gain: controller/Ki */
   SCtrl1_Ki = (Int16) ((((Int32) SCtrl1_si) * 13107) >> 16);
   /* Sum: controller/sPI */
   SPI = (Int16) (SCtrl1_Kp + SCtrl1_Ki);
```

```
    /* TargetLink outport: controller/UPI */
    UPI = SPI;
    LOG_VAR(Ctrl, _UPI, UPI);
    /* Unit delay: controller/xi */
    X_SCtrl1_xi = SCtrl1_si;
}
#pragma section restore
```

Related topics

Basics

AR_MEMORY_MAPPING (📖 TargetLink Demo Models)
Basics on Code Decorations..... 183
POSCONTROL (📖 TargetLink Demo Models)

HowTos

How to Specify Code Decorations in the Data Dictionary..... 186

Examples

Example of Assigning Variables to Memory Sections When Using a Target Optimization Module..... 193
Example of Mapping Variables to Memory Sections via Code Decorations..... 187

References

EnableSections (📖 TargetLink Model Element Reference)
TargetLink Main Dialog Block (📖 TargetLink Model Element Reference)

How to Specify Code Decorations in the Data Dictionary

Objective

To specify code decorations, such as declaration statements, pragmas or memory sections, for code elements.

Method

To specify code decorations in the data dictionary

1. In the Data Dictionary Manager, select the Pool subtree.
2. From its context menu, select Create CodeDecorationSets.
3. From the context menu of the CodeDecorationSets object, select Create CodeDecorationSet.

   TargetLink creates a DD CodeDecorationSet.
4. Rename the DD CodeDecorationSet object as required, e.g., `MyCodeDecorationSet`.

5 Reference MyCodeDecorationSet at each DD VariableClass or FunctionClass object whose associated variables or functions you want to decorate via their CodeDecorationSetRef property.

6 From MyCodeDecorationSet's context menu, select Create CodeDecoration.

   TargetLink creates a DD CodeDecoration object.

7 Rename the DD CodeDecoration object as required, e.g., `CodeDecoration`.

8 Select its Filter object and specify its properties as required.

9 Select its Settings object and specify its properties as required.

Result

You specified a code decoration, such as declaration statements, pragmas or memory sections.

Related topics

Basics

Basics on Code Decorations..... 183

Examples

Example of Assigning Variables to Memory Sections When Using a Target Optimization Module..... 193
Example of Mapping Variables to Memory Sections via Code Decorations..... 187

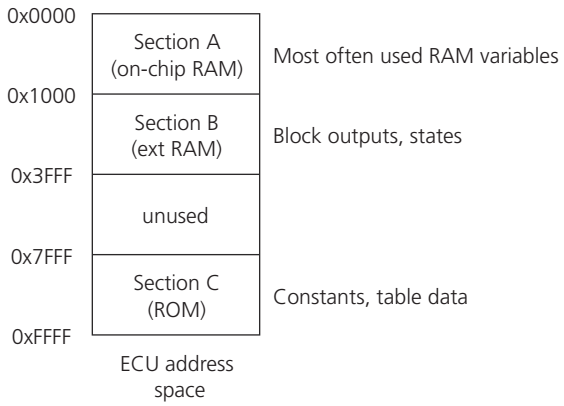
Example of Mapping Variables to Memory Sections via Code Decorations

Using different memory sections

Suppose you want to define the memory sections to which your application variables are allocated, because the compiler's or linker's default variable separation is not sufficient. To do so, you have to make the compiler use different memory sections for the variables.

Basics

The allocation of memory sections is essential for a production-quality ECU program. Certain memory sections in an ECU are used by specific types of memory modules, for example, RAM, ROM, EEPROM, and flash. Changeable variables that are often used are usually allocated to the on-chip RAM and table values to the flash memory.



Overview of the example

This example shows the following:

- Mapping all non-calibratable variables with global scope and static or default storage to width-specific memory sections via pragmas.
- Mapping calibratable variables with global scope and static or default storage to separate width-specific memory sections via pragmas.

Workflow

This example consists of the following parts:

- Creating a **CodeDecorationSet** object that contains a **CodeDecoration** object, refer to Part 1 on page 188.
- Specifying the filter of the **CodeDecoration** object, refer to Part 2 on page 189.
- Adjusting the **CodeDecoration** object to different widths, refer to Part 3 on page 189.
- Setting your **CodeDecoration** object as default, refer to Part 4 on page 190.
- Mapping calibratable variables to a special memory section, refer to Part 5 on page 191.

Tip

If you want to perform memory mapping for code that you generate using one of TargetLink's Target Optimization Modules (TOM), you can also follow these instructions: Example of Assigning Variables to Memory Sections When Using a Target Optimization Module on page 193.

Part 1

To create a CodeDecorationSet object that contains a CodeDecoration object

1. In the Data Dictionary Manager, select the **Pool** subtree.
2. From its context menu, select **Create CodeDecorationSets**.

3 From the context menu of the CodeDecorationSets object, select Create CodeDecorationSet.

TargetLink creates a DD CodeDecorationSet.

4 Rename the DD CodeDecorationSet object as required, e.g., Data.

5 From Data's context menu, select Create CodeDecoration.

TargetLink creates a DD CodeDecoration object.

Interim result

You created a DD CodeDecorationSet object that contains one DD CodeDecoration object.

Next you specify the Filter object of the CodeDecoration object.

Part 2

To specify the CodeDecoration object's filter

1 Select the DD CodeDecoration object's Filter object.

2 Set the IntendedFor property to `Variables`.

3 From the ScopeSpec property's context menu, select Open Editor.

4 In the Edit Bitfield dialog, select `APPLY_TO_GLOBAL` and `APPLY_TO_STATIC_GLOBAL`.

5 Close the dialog by clicking OK.

Interim result

You specified the Filter object of the DD CodeDecoration object. It now filters for variables of global scope whose storage is default or static.

Next you adjust the CodeDecoration object to different widths by duplicating it and refining its filter and settings.

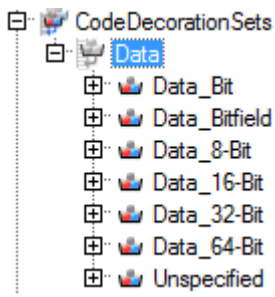
Part 3

To adjust the CodeDecoration object to different widths

1 Select the DD CodeDecoration object and duplicate (`Ctrl-D`) it six times.

2 Rename each DD CodeDecoration object to `Data_<Width>`. Rename the remaining DD CodeDecoration object to `Unspecified`.

Your DD CodeDecorationSet object named `Data` now contains seven DD CodeDecoration objects:



3 At each DD CodeDecoration object, set the Filter.WidthSpec property as shown in the following table:

CodeDecoration Object	Value
Data_Bit	`APPLY_TO_BIT`
Data_Bitfield	`APPLY_TO_BITFIELD`
Data_8-Bit	`APPLY_TO_8BIT`
Data_16-Bit	`APPLY_TO_16BIT`
Data_32-Bit	`APPLY_TO_32BIT`
Data_64-Bit	`APPLY_TO_64BIT`
Unspecified	`<value not set>`

4 At each DD CodeDecoration object, set the Settings.DeclarationStatements property as required by your compiler, e.g., as shown in the following table:

CodeDecoration Object	PreDefinitionBlockStatement	PostDefinitionBlockStatement
Data_Bit	`#pragma section = "DATA_BIT"`	`#pragma section restore`
Data_Bitfield	`#pragma section = "DATA_BITFIELD"`	
Data_8-Bit	`#pragma section = "DATA_8BIT"`	
Data_16-Bit	`#pragma section = "DATA_16BIT"`	
Data_32-Bit	`#pragma section = "DATA_32BIT"`	
Data_64-Bit	`#pragma section = "DATA_64BIT"`	
Unspecified	`#pragma section = "DATA_UNSPECIFIED"`	

Interim result

You adjusted the CodeDecoration object to different widths by duplicating it and refining its filter and settings.

Next you set the CodeDecorationSet object named `Data` as default to be used for all variable definitions generated by TargetLink.

Part 4

To set your CodeDecorationSet as default

1 Select the `/Config/TargetLink` object.

2 In its Property Value List, make the following settings:

Property	Value
DefaultDecorationSetRef	`CodeDecorationSets/Data`
AlwaysUseDefaultDecorationSet	`on`

Interim result

You set your CodeDecorationSet as the default and configured TargetLink to use it for the following variables:

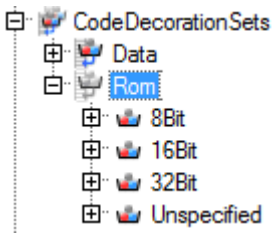
- Variables associated with classes explicitly defined in the Data Dictionary that do not reference a CodeDecorationSet themselves (explicit object)
- Implicitly generated variables (implicit object)
- Implicitly generated variables with classes derived from internal default classes via DD VariableClassTemplate objects (implicit property)

Next you can specify more CodeDecorationSet objects to reference at DD VariableClass objects. This lets you map frequently used variables, calibration parameters, or measurable variables to special memory sections.

Part 5

To map calibratable variables to special memory sections

1. Create a DD CodeDecorationSet that contains as many DD CodeDecoration objects as required.
2. Rename this object and its child objects as required:



3. Specify each DD CodeDecoration object's filter and settings as required.
4. Reference the DD CodeDecorationSet named `Rom` at all the DD VariableClass objects that you used for your calibratable variables via the CodeDecorationSetRef property of each class.

   For example, at the predefined variable classes CAL and STATIC_CAL.

Result

TargetLink generates the decoration into the production code according to your specifications. The following code example stems from the POSCONTROL demo model:

```
#pragma section = "DATA_8BIT"
/*****************************************************************************\
   SLGlobalConst: Default storage class for global const variables | Width: 8
\*****************************************************************************/
const UInt8 SCtrl2_Look_Up_Table_Nx = 25; /*
   LSB: 2^0 OFF: 0 MIN/MAX: 25 .. 25
   Description: number of axis points */
const UInt8 SCtrl2_Look_Up_Table_diffx = 3; /*
   LSB: 2^0 OFF: 0 MIN/MAX: 3 .. 3
   Description: step size axis vector, arbitrary or as a shift factor */
const Int8 SCtrl2_Look_Up_Table_startx = -96 /* -12. */; /*
   LSB: 2^-3 OFF: 0 MIN/MAX: -12 .. -12
   Description: axis vector starting point */
#pragma section restore

#pragma section = "ROM_8BIT"
/*****************************************************************************\
   CAL: global calibratable parameters (ROM) | Width: 8
\*****************************************************************************/
const volatile Int8 LinTable[25] =
{
   /* [0..9] */ -12, -12, -11, -11, -10, -9, -8, -7, -6, -4,
   /* [10..19] */ -3, -1, 0, 1, 3, 4, 6, 7, 8, 9,
   /* [20..24] */ 10, 11, 11, 12, 12
   /* -12., -11.8002765350741, -11.416413701523, -10.8544015504781, -10.1230100935865, -9.2336524499
   1286, -8.20020674803336, -7.03879956248094, -5.76755426394346, -4.4063082101256,
   -2.97630319041666, -1.49985395487165, 0., 1.49985395487165, 2.97630319041666, 4.4063082101256, 5.
   76755426394346, 7.03879956248094, 8.20020674803336, 9.23365244991286,
   10.1230100935865, 10.8544015504781, 11.416413701523, 11.8002765350741, 12. */
}; /*
   LSB: 2^0 OFF: 0 MIN/MAX: -12 .. 12
   Description: linearized output values */

/*****************************************************************************\
   STATIC_CAL: static calibratable parameters (ROM) | Width: 8
\*****************************************************************************/
static const volatile Bool LinEnableLUT = 1; /*
   LSB: 2^0 OFF: 0 MIN/MAX: 0 .. 1
   Description: enable table lookup */
static const volatile UInt8 LinGain = 85 /* 0.666666666666667 */; /*
   LSB: 2^-7 OFF: 0 MIN/MAX: 0.3 .. 1.3
   Description: linearization factor for small scale signals */
#pragma section restore
```

Related topics

Basics

Basics on Code Decorations..... 183
Basics on Using Name Macros..... 78
Basics on Variable Classes..... 88
POSCONTROL (📖 TargetLink Demo Models)

HowTos

How to Specify Code Decorations in the Data Dictionary..... 186

Examples

Example of Assigning Variables to Memory Sections When Using a Target Optimization Module..... 193

References

Declaration Statement (📖 TargetLink Data Dictionary Manager Reference)

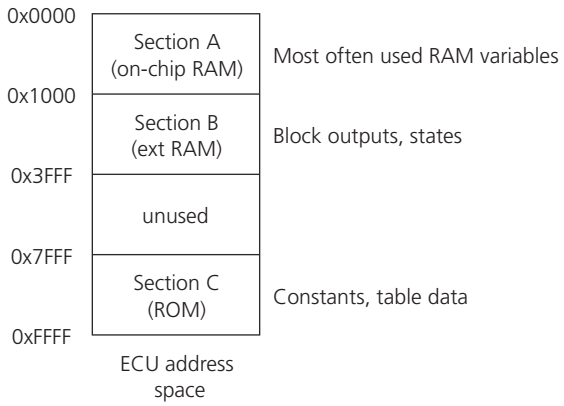
Example of Assigning Variables to Memory Sections When Using a Target Optimization Module

Using different memory sections in target-specific code

Suppose you are using one of TargetLink's Target Optimization Modules and want to define the memory sections to which your application variables are allocated, because the default variable separation of the compiler or linker is not sufficient. To do so, you have to make the compiler use different memory sections for the variables.

Basics

The allocation of memory sections is essential for a production-quality ECU program. Certain memory sections in an ECU are used by specific types of memory modules, for example, RAM, ROM, EEPROM, and flash. Changeable variables that are often used are usually allocated to the on-chip RAM and table values to the flash memory.



Restriction

This method to allocate variables to memory sections is restricted to target-specific code generated for one of TargetLink's Target Optimization Modules.

Note

When you generate code for one of TargetLink's Target Optimization Modules (TOM), using the **SectionName** property usually leads to non-portable code. This code is executable only on a specific hardware platform. The generated code might not be executable on the host PC or on a target evaluation board (physical or virtual) in SIL/PIL simulation mode.

Preconditions

- The **Enable sections/pragmas/inline/ISR/user attributes** checkbox on the TargetLink Main Dialog is selected.
- You created a DD **CodeDecorationSet** called `Rom` as described in How to Specify Code Decorations in the Data Dictionary on page 186.

Method

To assign variables to memory sections when using a target optimization module

1. Open the **POSCONTROL** demo model.
2. On the **Code Generation** page of the **TargetLink Main Dialog Block**, select your TOM from the **Code generation target settings** list, e.g., `TOM TriCore1767/Task32` and close the dialog via the **OK** button.
3. Reference the DD **CodeDecorationSet** named `ROM` at all the DD **VariableClass** objects that you used for your calibratable variables via the **CodeDecorationSetRef** property of each class.

   For example, at the predefined variable classes **CAL** and **STATIC_CAL**.
4. Set the DD **CodeDecoration** object's **Settings.SectionName** property to `Rom_8Bit`.
5. Generate code.

Result

TargetLink generates the target-specific code for the TOM named `TOM TriCore1767/Task32`. It uses the value of the SectionName to generate the pragmas as required by the `Tasking 32` compiler. Additionally, the definitions of the variables associated with the predefined variable classes CAL and STATIC_CAL are grouped between the pragmas:

```
/* new name for section 'rom' */
#pragma section nearrom = "near_Rom_8Bit"
#pragma section farrom = "far_Rom_8Bit"

/*****************************************************************************\
   CAL: global calibratable parameters (ROM) | Width: 8
\*****************************************************************************/
const volatile Int8 LinTable[25] =
{
   /* [0..9] */ -12, -12, -11, -11, -10, -9, -8, -7, -6, -4,
   /* [10..19] */ -3, -1, 0, 1, 3, 4, 6, 7, 8, 9,
   /* [20..24] */ 10, 11, 11, 12, 12
   /* -12., -11.8002765350741, -11.416413701523, -10.8544015504781, -10.1230100935865, -9.2336524499
   1286, -8.20020674803336, -7.03879956248094, -5.76755426394346, -4.4063082101256,
   -2.97630319041666, -1.49985395487165, 0., 1.49985395487165, 2.97630319041666, 4.4063082101256, 5.
   76755426394346, 7.03879956248094, 8.20020674803336, 9.23365244991286,
   10.1230100935865, 10.8544015504781, 11.416413701523, 11.8002765350741, 12. */
}; /*
   LSB: 2^0 OFF: 0 MIN/MAX: -12 .. 12
   Description: linearized output values */

/*****************************************************************************\
   STATIC_CAL: static calibratable parameters (ROM) | Width: 8
\*****************************************************************************/
static const volatile UInt8 LinGain = 85 /* 0.666666666666667 */; /*
   LSB: 2^-7 OFF: 0 MIN/MAX: 0.3 .. 1.3
   Description: linearization factor for small scale signals */

/*****************************************************************************\
   STATIC_CAL: static calibratable parameters (ROM) | Width: 1
\*****************************************************************************/
static const volatile Bool LinEnableLUT = 1; /*
   LSB: 2^0 OFF: 0 MIN/MAX: 0 .. 1
   Description: enable table lookup */
/* set default section names */
#pragma section nearrom
#pragma section farrom
```

Non-ANSI C statements

For other non-ANSI C statements in conjunction with variable declarations, refer to Example of Mapping Variables to Memory Sections via Code Decorations on page 187.

Related topics

Basics

Basics on Function Reuse..... 298
Basics on the Download Process (📖 TargetLink Preparation and Simulation Guide)
Basics on Variable Classes..... 88
POSCONTROL (📖 TargetLink Demo Models)

Examples

Example of Mapping Variables to Memory Sections via Code Decorations..... 187
Example of Specifying Type Qualifiers via Type Prefixes..... 122

References

TargetLink Main Dialog Block (📖 TargetLink Model Element Reference)

Customizing TargetLink's Code Formatting

Introduction

TargetLink lets you customize the style and layout of the generated code output using an XML style definition file and a predefined root style sheet (XSL).

Where to go from here

Information in this section

Basics on Code Formatting..... 197

How to Customize the Code Formatting via Code Output Style Definition File..... 198

How to Select a Root Style Sheet..... 200

Basics on Code Formatting

Code formatting in TargetLink

Via code formatting, you can define the style and layout of your code output. TargetLink lets you customize the code formatting as required. For this purpose, you can customize the code output style definition file and code output style sheets. Both have to be derived from templates. Refer to Defining Search Paths for Customization Files, Demo Models,TSM Extension Packages, and Instruction Set Simulators on page 28.

Code output style definition file By customizing the code output style definition file (XML file), you can modify the representation of comments and statements in your code output.

Code output style sheets By customizing code output style sheets (XSL files), you can modify your code output. A root style sheet is used to organize several style sheets defining code formatting. To select a root style sheet, refer to How to Select a Root Style Sheet on page 200.

Updating version-specific code output style definition file and code output style sheets

The code output style definition file and code output style sheets are version-specific and subject to change from one TargetLink version to another. Therefore, modified code output style definition file and code output style sheets of older TargetLink versions have to be checked and updated, e.g., with the help of a diff and merge tool.

During code generation TargetLink displays an error if the used code output style definition file or code output style sheets do not match the current TargetLink version.

Related topics

Basics

Defining Search Paths for Customization Files, Demo Models,TSM Extension Packages, and Instruction Set Simulators..... 28

HowTos

How to Customize the Code Formatting via Code Output Style Definition File..... 198
How to Select a Root Style Sheet..... 200

References

Code Formatting Customization Files (TargetLink File Reference)
tlCustomizationFiles (TargetLink API Reference)

How to Customize the Code Formatting via Code Output Style Definition File

Objective

To customize the code formatting, you can modify a code output style definition file by setting the attributes as required. Detailed information and examples on changing values of attributes are commented in the code output style definition file.

Precondition

You derived a new code output style definition file from its template and saved it with a suitable file name, for example, `my_cconfig.xml`. Refer to Defining Search Paths for Customization Files, Demo Models,TSM Extension Packages, and Instruction Set Simulators on page 28.

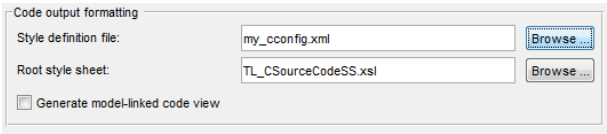
Method

To customize the code formatting via code output style definition file

1. In the code output style definition file, set the attributes as required. For example:
   - `show="true"` or `"false"`
   - `format="tl"` or `"user"`
   - `type="tl"` or `"user"`

   The format and type settings represent different layouts, for example, for the `if`, `while`, and `switch` commands.
2. Save your code output style definition file.
3. Open the TargetLink Main Dialog.
4. Switch to the Advanced page.

5 Click the **Browse** button next to the **Style definition file** edit field.



6 Select your code output style definition file, for example, `my_cconfig.xml` and click **Open**.

Result

You customized the code formatting via code output style definition file. The changes are applied during the next code generation.

Example

By default, a greet comment from TargetLink is displayed in a code header of code output files.

```
/*****************************************************************************
 ***
 *** Simulink model       : vcfp
 *** TargetLink subsystem : vcfp/pidcontroller
 *** Codefile             : pidcontroller.c
 ***
 *** Generated by TargetLink, the dSPACE production quality code generator
 *** Generation date: 2014-10-27 16:56:00
 ***
 *** CODE GENERATOR OPTIONS:
```

You can disable this comment by setting the `tl-greet-comment` attribute to **false**. Your next generated code output will no longer contain the comment.

```
<TL:comments delimiter="***" show="true">
<!--
    The 'delimiter' property is used for multiline comments in "prefixed" mode
    (see below).
    For example:
    delimiter="***" will produce /***
                                  ***   comment
                                  ***/

    Use 'show="true/false"' on a comment element to specify, if it shall appear or not.
    Use 'show-in-non-root-files="true/false"' to specify that a comment element shall (not)
    appear in files that do not contain model code like e.g. tl_basetypes.h .
-->
    <TL:file-section-comment show="true"/>
    <TL:header-comment show="true" show-in-non-root-files="true">
        <TL:model-file-comment show="true"/>
        <TL:active-variant show="true"/>
        <TL:tl-greet-comment show="false"/>
        <TL:generation-date-comment show="true"/>
        <TL:subsystem-list-comment show="true"/>
        <TL:stateflow-nodes-list-comment show="true"/>
        <TL:modelref-block-list-comment show="true"/>
        <TL:option-list-comment show="true"/>
        <TL:version-comment show="true"/>
        <!-- enter your copyright notice here and set show="true" -->
        <TL:copyright-comment text="enter your copyright notice here" show="false"/>
    </TL:header-comment>
    <TL:include-comment show="true"/>
    <TL:code-section-comment>
```

Related topics

Basics

Basics on Code Formatting..... 197
Defining Search Paths for Customization Files, Demo Models,TSM Extension Packages, and Instruction Set Simulators..... 28

HowTos

How to Create Customization Files via the Create Customization Files Dialog..... 38
How to Select a Root Style Sheet..... 200

References

Code Formatting Customization Files (TargetLink File Reference)
tlCustomizationFiles (TargetLink API Reference)

How to Select a Root Style Sheet

Objective

To select a root style sheet.

Preconditions

You changed style sheets in addition to a root style sheet to customize your code formatting as required.

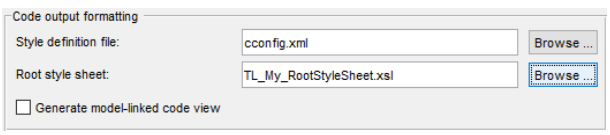
Note

If you are not familiar with XSL style sheets, it is recommended not to edit them. Incorrect entries in style sheets will interrupt code generation.

Method

To select a root style sheet

1. Open the TargetLink Main Dialog.
2. Switch to the Advanced page.
3. Click the Browse button next to the Root style sheet edit field.



4. Select your root style sheet, for example, `TL_My_RootStyleSheet.xsl` and click Open.

Result

You selected a root style sheet. The changes are applied during the next code generation.

Related topics

Basics

Basics on Code Formatting..... 197

HowTos

How to Customize the Code Formatting via Code Output Style Definition File..... 198

Controlling Objects and Properties Created by the Code Generator (Templates)

Introduction

You can use templates to change implicit objects (e.g., internal variables or functions), or implicit properties (e.g., variable classes), which are automatically generated by the Code Generator.**TargetLink Data Dictionary**. For example, you can adapt the naming of the implicit objects to company coding styles or change implicit properties like variable classes or function classes chosen by the Code Generator.

Where to go from here

Information in this section

Basics on Using Templates.....203
Details on Predefined DD Template Objects.....209
How to Create DD Template Objects.....213
Details on Variable Class Templates (VCTs).....214
Details on Scope Reduction and VCTs.....217
Details on Function Class Templates.....220
How to Specify VCTs Preserving Code Optimization Capabilities.....222

Basics on Using Templates

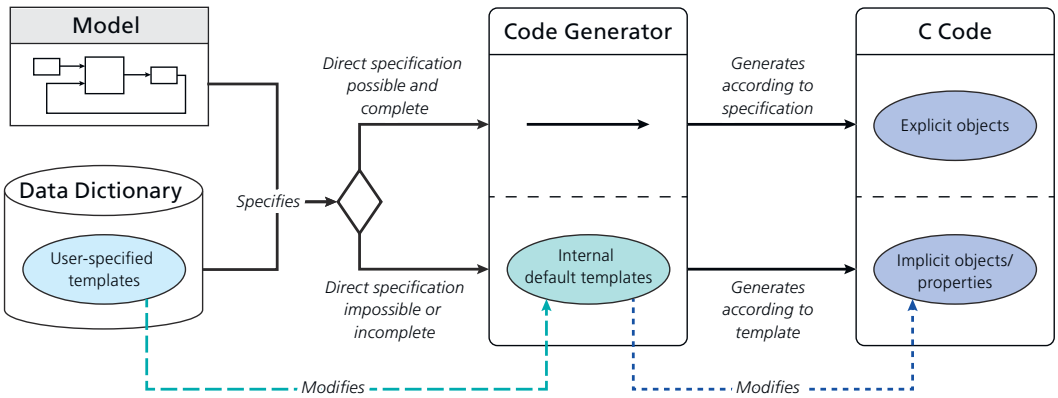
Differences between explicit objects and implicit objects/properties

TargetLink lets you specify objects, such as variables, type definitions, or functions. In most cases, you make your specifications in TargetLink block

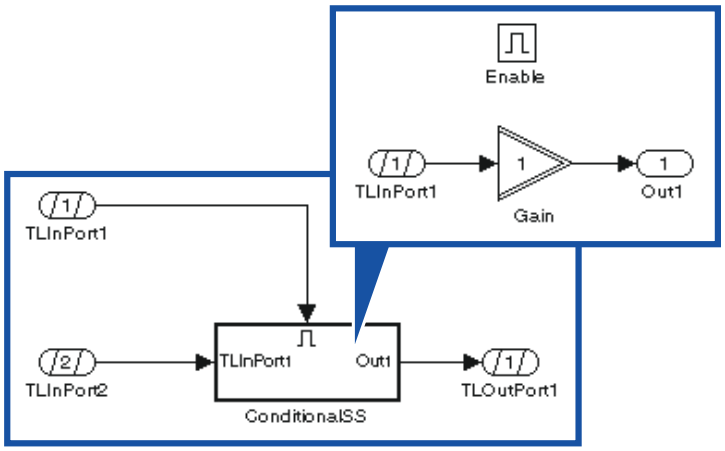
dialogs, the Stateflow Explorer, or the **TargetLink Data Dictionary**. This is called *direct specification*.

The Code Generator creates various entities. Three types are of particular interest:

- Explicit objects
- Implicit objects
- Implicit properties



Example of an implicit object Consider a conditionally executed subsystem that requires a persistent output variable, which you have not specified explicitly. This means you used a Simulink Outport block instead of a TargetLink **OutPort** block to model the subsystem output.



If the block output variable of the **Gain** block in the **ConditionalSS** subsystem is of `local` scope and therefore not persistent, it cannot be used as the output variable of the conditionally executed system. In this case, the Code Generator automatically creates an implicit interface variable `IF_<$S_$L>` on the basis of the internal default templates.

Example of an implicit property Consider a block you intentionally specified incompletely to give the Code Generator as many optimization options as possible. Suppose you specified scaling parameters but not a variable class. Instead, you left `default` as the value of the **Class** edit field in a block dialog.

This allows the Code Generator to choose an appropriate variable class and possibly further optimize the generated code. If the block variable is implemented in the generated code, some of its properties are set according to the internal default templates. This makes the variable class property an implicit property.

Note

The following template descriptions do not apply to access function templates. Refer to Encapsulating Access to Data Signals and Parameters via Access Functions on page 268.

Influencing implicit objects/properties via templates

You can influence the Code Generator default settings for implicit objects/properties via templates. In addition, templates let you influence groups of implicit objects/properties all at once, within defined limits.

Specifying templates

Templates are specified in the in the `/Pool/Templates` subtree of the TargetLink Data Dictionary. Every DD **Template** object has an object tree consisting of a **Filter** and a **Settings** object. The properties of each **Filter** object let you define a filter rule, and the properties of each **Settings** object let you specify what to do if the filter rule applies.

Key property of the Filter object Depending on the kind of template, the **Filter** object has different properties that let you specify complex filter criteria. You must specify at least the **Filter** object's key property by entering an appropriate string. The key properties for the templates are:

DD Template Object	Key Property	Documentation
DD FunctionTemplate object	FunctionName	Filter Criteria of the FunctionTemplate Object
DD FunctionClassTemplate object	FunctionClassSpec	Filter Criteria of the FunctionClassTemplate Object
DD LookupFunctionTemplate object	LookupFunctionName	Filter Criteria of the LookupFunctionTemplate Object
DD StructTemplate object	StructSpec	Filter Criteria of the StructTemplate Object
DD SubStructTemplate object	StructSpec	Filter Criteria of the SubStructTemplate Object
DD TypedefTemplate object	TypeDefKind	Filter Criteria of the TypedefTemplate Object
DD VariableTemplate object	VariableKind	Filter Criteria of the VariableTemplate Object
DD VariableClassTemplate object	VariableClassSpec	Filter Criteria of the VariableClassTemplate Object

Note

To influence RTOS root functions via templates, refer to:

- How to Add a Root Function Template and Specify Its Properties (TargetLink Multirate Modeling Guide)
- Filter Criteria of the RootFunctionTemplate Object

Specifying the Filter object You can define the filter rule by choosing from predefined strings for the value of the template filter's key property. Each predefined string forms a keyword that the Code Generator uses for filtering. The concise naming of the keywords helps you choose the correct one.

There are different ways to influence functions and variables via DD **Template** objects:

DD Template Object Kind	Use Case
FunctionTemplate	Influence how functions that match a specified filter condition are generated in the production code.
FunctionClassTemplate	Influence which DD **FunctionClass** object is used when no DD **FunctionClass** object is specified.
VariableTemplate	Influence how implicit variables that match a specified filter condition are generated in the production code.
VariableClassTemplate	Influence which DD **VariableClass** object is used when no DD **VariableClass** object is specified.

Specifying the Settings object The template's **Settings** object defines how TargetLink treats implicit objects/properties that match the filter rule.

For variable and function classes, the settings you specify in the **Settings** object must not contradict the filter condition. For example, if a filter applies to extern global variables, you must not reference a variable class for module-local or function-local variables at the **Settings** object. Conflicting values cause the Code Generator to display an error message.

Code generation and templates

Matching the filter rule If the implicit objects/properties generated by the Code Generator can be controlled via a template, TargetLink searches the `/Pool/Templates` subtree for the first template that matches the filter rule.

The Code Generator uses the kind of the implicit object/property to decide which kind of template to search for and the context of the implicit object/property to decide whether the filter settings apply. The first matching template that is found is applied, i.e., the internal default values of the Code Generator are overwritten by the settings from that template. All other templates are ignored. You can change the order of DD template objects or rename them via their context menu. You can use any number of templates.

Example Consider the ConditionalSS (refer to Example of an implicit object on page 204) subsystem again.

Because the block output variable of the **Gain** block in the **ConditionalSS** subsystem is of `local` scope and therefore not persistent, it cannot be used as the output variable of the conditionally executed system. The Code Generator automatically creates an implicit interface variable, which you can influence via a DD **VariableTemplate** object with `GlobalInterfaceVar` as the value of the **Filter** object's **VariableKind** property.

If you want to change the properties of the internal default variables TargetLink uses for global interface variables, you have to set the **VariableClassSpec** property to either `SLGlobal` or `SLGlobalInit`, and reference a user-specified variable class at the variable templates **Settings** object. Which **VariableClassSpec** string you choose depends on whether the interface variable is initialized, as shown in Filter Criteria of the VariableTemplate Object.

Note

Changing the default behavior of an entire variable class, affects all variables that implicitly use this variable class. This might not be your intention. However, it is the intended purpose of templates.

Furthermore, different templates can interfere with each other: To avoid conflicts between DD **Template** objects sharing the value of their DD **VariableClassSpec** property, you can set the DD **FilterCondition** property to `NEVER`.

Predefined templates

If your DD workspace is based on the **dsdd_master_advanced** template, it contains several predefined DD **Template** objects for a broad range of use cases. Refer to Details on Predefined DD Template Objects on page 209.

Related topics

Basics

Details on Function Class Templates..... 220
Details on Predefined DD Template Objects..... 209
Details on Variable Class Templates (VCTs)..... 214
Encapsulating Access to Data Signals and Parameters via Access Functions..... 268

HowTos

How to Add a Root Function Template and Specify Its Properties (📖 TargetLink Multirate Modeling Guide)
How to Assign Function Classes to Implicit Functions..... 234
How to Create DD Template Objects..... 213
How to Create DD Workspaces (📖 TargetLink Data Dictionary Basic Concepts Guide)
How to Specify VCTs Preserving Code Optimization Capabilities..... 222

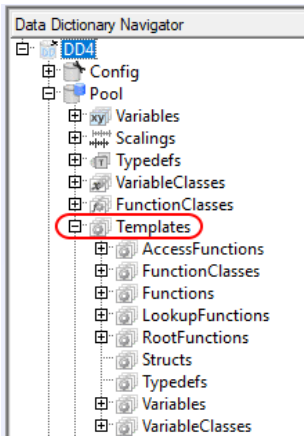
References

Filter Criteria of the FunctionClassTemplate Object (📖 TargetLink Data Dictionary Reference)
Filter Criteria of the FunctionTemplate Object (📖 TargetLink Data Dictionary Reference)
Filter Criteria of the LookupFunctionTemplate Object (📖 TargetLink Data Dictionary Reference)
Filter Criteria of the RootFunctionTemplate Object (📖 TargetLink Data Dictionary Reference)
Filter Criteria of the StructTemplate Object (📖 TargetLink Data Dictionary Reference)
Filter Criteria of the SubStructTemplate Object (📖 TargetLink Data Dictionary Reference)
Filter Criteria of the TypedefTemplate Object (📖 TargetLink Data Dictionary Reference)
Filter Criteria of the VariableClassTemplate Object (📖 TargetLink Data Dictionary Reference)
Filter Criteria of the VariableTemplate Object (📖 TargetLink Data Dictionary Reference)

Details on Predefined DD Template Objects

Templates in the DD object tree

Templates are defined in the TargetLink Data Dictionary. The following illustration shows the DD object tree containing several templates.



The templates are stored as child objects of the DD Templates object in the Pool area. The DD Templates object is the parent object of two different kinds of objects:

- Templates for special objects, for example, FunctionClassTemplate or VariableTemplate.
- DD TemplatesGroup objects. DD TemplatesGoup objects are not required for code generation and are used to structure the Data Dictionary.

The templates have the following child objects:

- DD Filter object.

  Defines identification criteria for the objects to which the template applies.
- DD Settings object.

  Defines how the objects are changed by the Code Generator.

Predefined settings for the DD project file

When you create a new DD project file, you can choose from predefined system templates.

Different predefined objects are available depending on which template you select.

dsdd_master_basic.dd [System] Provides the new DD workspace with a basic set of predefined DD objects, such as DD Typedef, DD VariableClass, and DD FunctionClass objects.

dsdd_master_advanced.dd [System] Provides the new DD workspace with additional predefined objects, such as DD Template objects and DD RTOS objects required for multirate code generation.

dsdd_master_autosar4.dd [System] Provides the new DD workspace with additional predefined objects, such as DD Typedef and DD Variable objects to

be used in AUTOSAR SWCs, and a DD object for specifying AUTOSAR software components and interfaces. Use this template to work with the AUTOSAR 4.x standard.

Using templates

When you use templates, you can influence the following kinds of implicit objects.

Template Types	Description	Documentation
Access function	The **AccessFunctionTemplate** lets you specify an access function, for example, to encapsulate the access to global variables such as interface signals or parameters.	▪ Encapsulating Access to Data Signals and Parameters via Access Functions on page 268
Function class	The **FunctionClassTemplate** lets you specify a function class to be used instead of a default function class that TargetLink defines for implicit functions.	▪ How to Assign Function Classes to Implicit Functions on page 234 ▪ Filter Criteria of the FunctionClassTemplate Object
Function	The **FunctionTemplate** lets you customize implicitly created functions, i.e., you can change the function names and classes. For certain functions, you can also specify a module in which to place them.	▪ Filter Criteria of the FunctionTemplate Object
Look-up function	The **LookupFunctionTemplate** lets you change the function name, code file, and record layout.	▪ Basics on Using Custom Look-Up Functions (📖 TargetLink Preparation and Simulation Guide) ▪ Filter Criteria of the LookupFunctionTemplate Object
Root function	The **RootFunctionTemplate** lets you modify the properties of Restart, Init, Terminate, and Step functions if no **Function** block is placed on the TargetLink root system. The templates for the Restart, Init, and Terminate (not Step) root functions must fulfill the following conditions: ▪ Only the **FunctionKind** and **TLSubsystemID** properties are specified in the **Filter** of the **RootFunctionTemplate**. The other properties must not be set. If you specify the other properties, the Code Generator cannot find the template. ▪ A **Task** must not be specified as a child DD object in the **Settings** of the **RootFunctionTemplate**. You can use the **RootFunctionTemplate** to specify the properties of the Step functions in the RTOS code generation mode. A step function is used to execute a code part with a fixed sample time.	▪ How to Add a Root Function Template and Specify Its Properties (📖 TargetLink Multirate Modeling Guide) ▪ How to Embed TargetLink Code into Your Own C Modules (📖 TargetLink Interoperation and Exchange Guide) ▪ Filter Criteria of the RootFunctionTemplate Object

Template Types	Description	Documentation
Struct	The **StructTemplate** lets you change the properties of implicit structs, i.e., the name for the struct instance, the variable class, the name of the instance struct type, or the order of the elements. Implicit structs are created by the Code Generator, for example, if functions are reused, global bitfields are enabled, or variant coding is used.	▪ Filter Criteria of the StructTemplate Object
Substruct	The **SubStructTemplate** is used only for function reuse. It lets you change the name for the embedded struct instance, the variable class, the name of the instance struct type, or the order of the components. For function reuse, substructures are embedded in a structure variable which contains the instance-specific reuse data. A single substructure contains instance-specific variables with compatible properties. If the substruct contains structs, you have to delete/unset the **WidthSpec** property.	▪ Filter Criteria of the SubStructTemplate Object
Enum	The **EnumTemplate** object is used to specify which DD **Typedef** object to use in the generated production code for model elements with a Simulink enumeration data type if the predefined `IMPLICIT_ENUM` DD **Typedef** object is used for model elements or if no data type can be specified in TargetLink.	▪ Basics on Making Simulink Enumerations TargetLink-Compliant (🕮 TargetLink Preparation and Simulation Guide) ▪ How to Assign Explicit TargetLink Enumeration Data Types via Templates (🕮 TargetLink Preparation and Simulation Guide) ▪ EnumTemplate
Typedef	The **TypedefTemplate** lets you specify naming rules for implicitly created and therefore anonymous types, such as the data type tag, data type name, and pointer to data type.	▪ Filter Criteria of the TypedefTemplate Object
Variable	The **VariableTemplate** lets you specify naming rules for the variable, the variable class, and the data type. Specifying the data type is allowed only in some cases, for example, for an iterator output.	▪ Filter Criteria of the VariableTemplate Object
Variable class	The **VariableClassTemplate** lets you change a default variable class and thus variables implicitly generated by the Code Generator.	▪ Filter Criteria of the VariableClassTemplate Object

Related topics

Basics

Basics on Making Simulink Enumerations TargetLink-Compliant (📖 TargetLink Preparation and Simulation Guide)
Basics on Using Custom Look-Up Functions (📖 TargetLink Preparation and Simulation Guide)
Encapsulating Access to Data Signals and Parameters via Access Functions..... 268

HowTos

How to Add a Root Function Template and Specify Its Properties (📖 TargetLink Multirate Modeling Guide)
How to Assign Explicit TargetLink Enumeration Data Types via Templates (📖 TargetLink Preparation and Simulation Guide)
How to Assign Function Classes to Implicit Functions..... 234
How to Create DD Template Objects..... 213
How to Create DD Workspaces (📖 TargetLink Data Dictionary Basic Concepts Guide)
How to Embed TargetLink Code into Your Own C Modules (📖 TargetLink Interoperation and Exchange Guide)

References

Filter Criteria of the FunctionClassTemplate Object (📖 TargetLink Data Dictionary Reference)
Filter Criteria of the FunctionTemplate Object (📖 TargetLink Data Dictionary Reference)
Filter Criteria of the LookupFunctionTemplate Object (📖 TargetLink Data Dictionary Reference)
Filter Criteria of the RootFunctionTemplate Object (📖 TargetLink Data Dictionary Reference)
Filter Criteria of the StructTemplate Object (📖 TargetLink Data Dictionary Reference)
Filter Criteria of the SubStructTemplate Object (📖 TargetLink Data Dictionary Reference)
Filter Criteria of the TypedefTemplate Object (📖 TargetLink Data Dictionary Reference)
Filter Criteria of the VariableClassTemplate Object (📖 TargetLink Data Dictionary Reference)
Filter Criteria of the VariableTemplate Object (📖 TargetLink Data Dictionary Reference)

How to Create DD Template Objects

Objective

To tailor production code generation to your needs, you can create templates in the TargetLink Data Dictionary.

Templates

A template consists of a filter rule and settings for code generation. If the filter conditions of a template are fulfilled, the settings define how the production code for the implicit object is generated. For information on the templates, refer to Basics on Using Templates on page 203.

Managing templates

The templates are managed in the TargetLink Data Dictionary. Refer to Details on Predefined DD Template Objects on page 209.

Method

To create a DD template object

1. Start the Data Dictionary Manager and load the DD project file that belongs to your model.
2. In the Data Dictionary Navigator, open the `/Pool/Templates` DD object.

   The `Templates` DD object is the parent DD object for all the kinds of templates in a model.
3. In the context menu of the `Templates` DD object, choose Create - <Object>Template to create a template.

   > **Tip**
   >
   > You can use `TemplateGroup` DD objects to structure the Data Dictionary.

   The TargetLink Data Dictionary creates the new template and its `Filter` and `Settings` child DD objects.
4. Define the properties for the `Filter` DD object. The number and kinds of properties depend on the type of the object. For information on the filter criteria, refer to Data Dictionary-Related Name Macros.

   > **Tip**
   >
   > To display all the available properties, make sure that you have activated the Show unset properties button in the toolbar.

5. Define the properties for the `Settings` DD object. The number and kinds of properties depend on the type of the object.

Result

In the next code generation, the objects to which the filter rules apply are changed according to the settings defined.

Related topics

Basics

Basics on Using Templates..... 203
Details on Predefined DD Template Objects..... 209

HowTos

How to Add a Root Function Template and Specify Its Properties (📖 TargetLink Multirate Modeling Guide)
How to Create an Access Function Specification..... 275
How to Create DD Workspaces (📖 TargetLink Data Dictionary Basic Concepts Guide)

Details on Variable Class Templates (VCTs)

Kinds of variable class templates

TargetLink provides variable class templates (VCTs) for different purposes. There are VCTs whose defining characteristics are the scope and storage properties of the variable class in question, VCTs pertaining to function interfaces and VCTs pertaining to TargetLink blocks or specific purposes.

Defining proper filter criteria, i.e., providing valid values for the **VariableClassSpec** property of each VCT is key to specifying VCTs.

User-specified variable class and internal default variable class

VCTs are used to change implicit variable classes or sets of internal default variables created by the Code Generator. That is, you instruct the Code Generator to change properties of implicit variable classes by deriving them from user-specified variable classes.

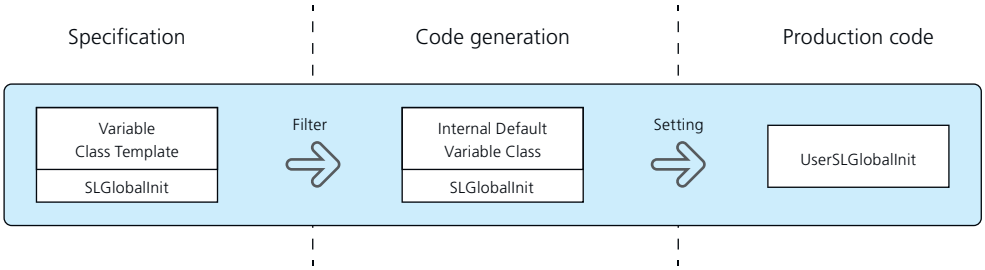
You must make at least the following two settings at each variable class template:

1. Specify a filter criterion by specifying the DD **Filter** object's **VariableClassSpec** property.

   This instructs the Code Generator to change one of its internal default variable classes according to your specifications. The possible filter criteria are the names of TargetLink's internal default variable classes. For a list, refer to Filter Criteria of the VariableClassTemplate Object.
2. Reference a user-specified variable class at the **VariableClass** property of the DD **Settings** object.

   This instructs the Code Generator to change certain properties of implicit variable classes by deriving them from user-specified variable classes.

The following illustration shows the basic template mechanism. Note that it abstracts from optimizations that can occur during code generation.



Constraints

VCTs are subject to the following constraints:

General constraints The user-specified variable class referenced at a VCT must fulfill its preconditions. For example:

- Function outputs cannot have their Scope property set to `value_param`, which is reserved for function inputs.
- Initialized variable classes with global scope must have their Scope property set to `global` and their InitAtDefinition property set to `on`. Alternatively, you can specify their RestartFunctionName property.

  This is because the Code Generator expects an initialization value for the variable, and initialization can occur at definition *or* in a restart function.

If the variable class you referenced at the Settings object of the DD VariableClassTemplate object does not fulfill the constraints associated with the VariableClassSpec property of the Filter object, the Code Generator displays an error message.

Scope Constraints You cannot reduce the scope of a default variable class by means of a template. For example, if you define a global variable class, you cannot give it local scope via a template.

Besides this basic rule, there are constraints derived from C semantics and special constraints.

C Semantics Constraints

- If a default variable class does not have `Const` as part of its VariableClassSpec string, the Code Generator assumes that write access can be performed. This means:
  - The Const property must not be set via a template.
  - The Macro property must not be set via a template without the Storage property being set to `extern` at the same time.
- Static initial values and nonzero simulation start values require an explicit initialization or value assignment. This corresponds partially to the general initialization constraints above.

- Function outputs in the parameter list require pointers. This corresponds to setting the DD Scope property to `ref_param`. Note that from a C language point of view, setting the DD Scope property to `value_param` leads to the same code and semantics as setting it to `ref_param`. See also the general constraints above.

Special Constraints There are more special constraints for the internal default variable classes of specific purposes, such as look-up table functions or filters. These constraints are shown in the following table:

Variable Class	Permitted Scope of Variable Class
SLLutLocal SLFilterLocal	`local`
SLLutFcnInput SLFilterFcnInput SLULutFcnInput	`value_param`
SLLutFcnReturn SLFilterFcnReturn	`fcn_return`
SLULutFcnOutputConst	`ref_param`
SLSystemTimerLocal SFLocalReentrant	`local`[1)]

[1)] With DD Storage property set to `default`.

Properties not changeable by VCTs

The following variable class properties cannot be influenced by using templates, because TargetLink *always* overwrites them when deriving an internal default variable class from the user-specified variable class:

- The name and description, which depend on the VariableClassSpec string.
- The UseName property, which is always set to `off`.
- The Optimization property values:
  - `ERASABLE`, which is always set.
  - `MOVABLE`, which is always set.
  - `MERGEABLE`, which is never set.

Application note

Additional information for expert users is available as an application note. For details, refer to Using TargetLink Variable Class Templates.

Related topics

Basics

Basics on the Scope Reduction of Variables..... 513
Basics on Using Templates..... 203
Details on Predefined DD Template Objects..... 209

HowTos

How to Create DD Template Objects..... 213
How to Specify VCTs Preserving Code Optimization Capabilities..... 222

Details on Scope Reduction and VCTs

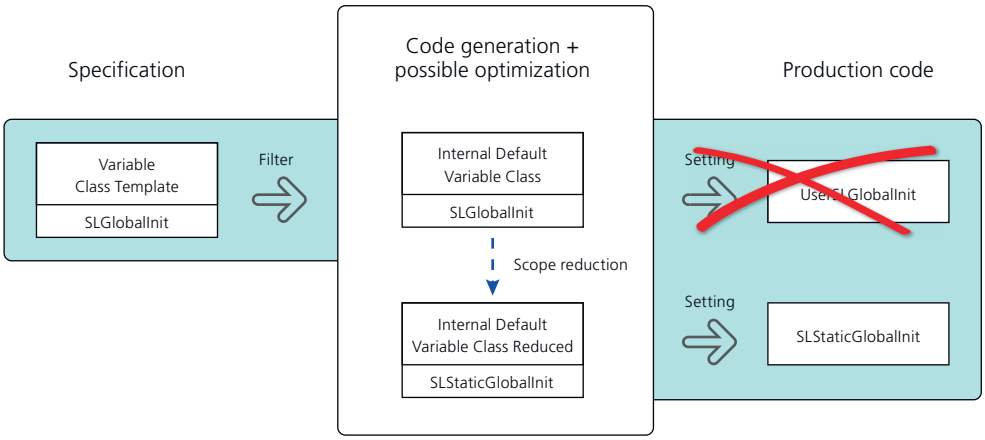
Changing variable class properties for specific scopes

TargetLink allows you to change properties of implicit variable classes via templates. Suppose you want to change a property of an implicit variable class that has a specific scope. That is, you want to change a property of all variables with that scope, be they implicit objects or directly specified variables with the variable class as an implicit (default) property. When specifying variable class templates for the present use case, note the difference between initially generated code and optimized code.

Initially generated versus optimized code

TargetLink performs optimizations, including scope reduction, on *initially generated code*: This is essentially the generated production code that you get if you set the Code Generator options Optimization to `off` and InliningThreshold to `0`. If you wish to affect variables that have a default variable class intended for a certain scope, then you have to take into account that the generated code that you finally see is not necessarily the code that TargetLink initially started with. If you do not take this into account, you might modify variable class settings in a way that leads to TargetLink not using the modified variables class. For code generation, this can mean not reducing to the same scope as prior to your specification or creation of a different variable class of that scope.

Thus, for your VCTs to work, you have to take TargetLink's optimizations into consideration, especially scope reduction:



Suppose you defined a DD **VariableClass** object to change all the variables with the implicit variable class `SLGlobalInit`. During optimization, however, the scope of one or more variables changes and they get the implicit variable class `SLStaticGlobalInit`. Your DD **Template** object is therefore not used. In the production code you do not find variables with the *derived* `UserSLGlobalInit` implicit variable class, as you might have expected, but variables with the implicit *default* variable class `SLStaticGlobalInit`.

Specifying VCTs along the scope reduction chain

TargetLink tries to reduce the scope of variables as much as possible. The different scope reduction steps form a chain called the *scope reduction chain* (SRC), which looks like this:

`Global ⇒ StaticGlobal ⇒ StaticLocal ⇒ Local`

To ensure that your VCTs work correctly, you have to specify one VCT for each scope belonging to the chain.

Specifying the VCTs You have to specify each VCT such that the user-specified variable classes referenced at the VCT's **Settings** object satisfy the constraints mentioned in Constraints on page 215 and Properties not changeable by VCTs on page 216.

For instructions, refer to How to Specify VCTs Preserving Code Optimization Capabilities on page 222.

Implicit variable classes for specific scopes

TargetLink provides several implicit variable classes whose defining characteristic is their scope and storage.

Naming schema If an implicit variable class is defined by its scope and storage, its **VariableClassSpec** string consists of different naming elements, as shown in the following schema:

Naming Schema
<Source> + <Scope> [+ <SubSpec>]

To specify VCTs along the scope reduction chain, you have to create a VCT for each pair of `<Source>` and `<SubSpec>` VariableClassSpec string parts, of a given `<Scope>`.

Naming elements of the schema The naming elements of the schema are shown in the following table:

Naming Element	Replaced By	Internal Variable Class For
<Source>	SL	Simulink variables
	SF	Stateflow variables
<Scope>	ExternGlobal	External global variables
	Global	Global variables
	StaticGlobal	Static global variables
	StaticLocal	Static local variables
	Local	Local variables
<SubSpec>	Init	Initialized variables
	Const	Constant variables

Frequent use case for specifying VCTs along the SRC

One frequent use case for specifying VCTs along the SRC is manipulating implicit variable classes whose `<SubSpec>` naming part is empty or `Init`.

Usually, you also want variables to behave the same regardless of whether they stem from Simulink blocks or Stateflow charts. You therefore need one user-specified variable class for the pair of SL/SF variable classes in each scope of the SRC. The following table shows the variable classes and VCTs you have to specify:

Variable Class Object	Variable Class Templates
`VC_Global<SubSpec>`	`VCT_SLGlobal<SubSpec>` `VCT_SFGlobal<SubSpec>`
`VC_StaticGlobal<SubSpec>`	`VCT_SLStaticGlobal<SubSpec>` `VCT_SFStaticGlobal<SubSpec>`
`VC_StaticLocal<SubSpec>`	`VCT_SLStaticLocal<SubSpec>` `VCT_SFStaticLocal<SubSpec>`
`VC_Local<SubSpec>`	`VCT_SLLocal<SubSpec>` `VCT_SFLocal<SubSpec>`

Note

Ensure to set the user-specified variable class' Optimization property to `SCOPE_REDUCIBLE`.

Tip

TargetLink does not display warnings for default variable class property differences that prohibit scope reduction.
To test whether scope reduction works, duplicate the variable classes for all scopes, remove `ERASABLE` from their **Optimization** property, and string the new classes together using the **ScopeReducedClass** property. If you assign the copy of the variable class with global scope to a block output variable followed by a **Rescaler** block with the same class, you can observe whether the block output variable is reduced to the new local scope variable class.
If this is not possible due to incompatible variable class properties, TargetLink displays a warning. This way of testing the variable classes also helps you to find out whether you forgot to set the **Optimization** property to `SCOPE_REDUCIBLE`.

Application note

Additional information for expert users is available as an application note. For details, refer to Using TargetLink Variable Class Templates.

Related topics

Basics

Basics on the Scope Reduction of Variables..... 513

HowTos

How to Specify VCTs Preserving Code Optimization Capabilities..... 222

References

InliningThreshold (📖 TargetLink Model Element Reference)
Optimization (📖 TargetLink Model Element Reference)

Details on Function Class Templates

Function class templates

TargetLink provides function class templates (FCTs) for different purposes: There are FCTs for auxiliary functions, Stateflow-specific FCTs, subsystem-specific FCTs, and FCTs for TargetLink blocks or special purposes.

Defining proper filter criteria, i.e., providing valid values for each FCT's **FunctionClassSpec** property is key to specifying FCTs.

User-specified function class and internal default function class

You can use FCTs to change implicit function classes (function class set to `default`) or sets of implicit functions created by the Code Generator. That is, you instruct the Code Generator to change properties of implicit function classes by deriving them from user-specified function classes.

You must make at least the following two settings at each function class template:

1. Specify a filter criterion by specifying the DD **Filter** object's **FunctionClassSpec** property.

   This instructs the Code Generator to change one of its internal default function classes according to your specifications. The possible filter criteria are the names of TargetLink's internal default function classes. For a list, refer to Filter Criteria of the FunctionClassTemplate Object.
2. Reference a user-specified function class at the DD **Settings** object's **FunctionClass** property.

   This instructs the Code Generator to change certain properties of implicit function classes by deriving them from user-specified function classes.

Constraints

General constraint The following rule applies to all default function classes:

If the **Macro** property is set to `on`, the **Storage** property must also be set to `extern`.

Visibility Constraint You cannot limit the visibility of internal default function classes via FCTs.

<Storage> FunctionClassSpec String Part	**Valid Storage of the User-Specified Function Class**		
	static	**default**	**extern**
Static	✓	✓	✓
Global	–	✓	✓
Extern	–	–	✓

Special Constraints The following function classes require that the **Optimization** property has `CG_NO_INLINE` selected:

- GlobalLookup1D
- GlobalLookup2D
- GlobalFIR

Related topics

Basics

Details on Predefined DD Template Objects..... 209
Details on Variable Class Templates (VCTs)..... 214

HowTos

How to Assign Function Classes to Implicit Functions..... 234
How to Create DD Template Objects..... 213

How to Specify VCTs Preserving Code Optimization Capabilities

Objective

One of the most important uses for variable class templates (VCTs) is changing implicit variable classes that belong to a specific scope. For example, you can use VCTs to initialize all the global initialized variables in a restart function or have them enclosed by pragmas. To make the VCTs work, you have to note TargetLink's optimization procedures and specify the VCTs along the relevant links of the scope reduction chain (SRC).

Restrictions

VCTs are subject to several constraints. For details, refer to Constraints on page 215 and Properties not changeable by VCTs on page 216.

Precondition

The following preconditions have to be fulfilled:

- You read Basics on the Scope Reduction of Variables on page 513 and Details on Scope Reduction and VCTs on page 217.
- The Data Dictionary Manager is open.

Method

Specifying VCTs while preserving code optimization capabilities

1. Create a DD **VariableClass** object for each scope belonging to TargetLink's scope reduction chain and rename it `VC_<Scope><SubSpec>`. This helps you when referencing your variable classes at the VCTs in step 5.
2. For all the DD **VariableClass** objects created in step 1, set the following properties:

Property	Value
Scope	<Scope> of the implicit variable class to change
Optimization	`SCOPE_REDUCIBLE`
<PropertyToChangeAtImplicitVariableClass>	<ValueOfPropertyToChangeAtImplicitVariableClass>

Ensure not to violate any of the constraints VCTs are subject to.

3. Create a DD **VariableClassTemplate** object for each scope belonging to TargetLink's scope reduction chain and rename it VCT_<Source><Scope><SubSpec>.
4. At the DD **Filter** object of each DD **VariableClassTemplate** object created in step 3, specify the implicit variable class to change as the value of the DD **VariableClassSpec** property.
5. Reference the DD **VariableClass** objects created in step 1 at the appropriate DD **VariableClassTemplate** object's **Settings** object.

   For example, you can reference `VC_<Scope>Init` at the `VCT_SF<Scope>Init` and `VCT_SL<Scope>Init` variable class objects to treat initialized variables with implicit variable classes the same way, whether they stem from Stateflow charts or Simulink blocks.
6. Generate code.

Result

You specified VCTs along the scope reduction chain.

Tip

Instead of creating DD **VariableClassTemplate** objects as described in step 3, you can copy them from a DD workspace based on the **dsdd_master_advanced** template.

Related topics

Basics

Basics on Optimizing Variables..... 504
Basics on the Scope Reduction of Variables..... 513
Basics on Using Templates..... 203
Details on Scope Reduction and VCTs..... 217

Mapping of Subsystems and Functions

Introduction

To understand and modify TargetLink's behavior with regard to the mapping of subsystems and functions, you need to know the following points.

Where to go from here

Information in this section

Structuring Large Models via Subsystems 226

Customizing the Appearance of Functions via Subsystems and Function Classes 230

Defining the Function Interface 241

Centrally Specifying and Synchronizing Function System Signatures 245

Encapsulating Access to Data Signals and Parameters via Access Functions 268

Reusing C Functions and Variables for Identical Model Parts (Function Reuse) 297

Implementing Scaling-Invariant Functions 320

Controlling Subsystem Execution 334

Controlling Function Inlining 350

Embedding External Code 358

Structuring Large Models via Subsystems

Virtual and atomic subsystems

Like Simulink, TargetLink lets you implement virtual and atomic subsystems. You can control how code is generated for each type of subsystem, that is, how the different subsystem types are treated during code generation and finally represented in the production code.

Where to go from here

Information in this section

Basics on Using Subsystems.....226

How to Implement Subsystems as Functions.....228

Basics on Using Subsystems

Mapping of subsystems

TargetLink allows you to map subsystems below the TargetLink subsystem to C functions or code blocks. It is recommended to structure the hierarchy of large Simulink models to make them easier to maintain. The use of hierarchies offers a more conceptual look at a model and hides details of C functions.

Note

Do not use TargetLink port blocks in subsystems except for one of the following reasons:

- TargetLink requires a port block in a subsystem.
- You need interface variables in the generated code.
- You need to access a signal value for a purpose such as logging.

During code generation, TargetLink treats each Simulink port block as signal routing. For each TargetLink port block, however, it generates a variable. This can result in overhead, such as greater memory consumption. Consequently, you should use Simulink port blocks at the interfaces of subsystems whenever possible.

Triggered or enabled model parts

For triggered or enabled model parts, the use of subsystems is even mandatory. The corresponding Simulink blocks are controlled by signal conditions and/or trigger events and have to be placed in subsystems that contain a Simulink Enable and/or Trigger block. The control signals are additional inputs of a subsystem and get their values from the surrounding parent subsystem.

Stateflow model parts

For model parts specified with Stateflow, hierarchies are available for states and form a fundamental part of the underlying formalism (refer to the *Stateflow User's Guide* provided by MathWorks®).

Virtual subsystems

Virtual subsystems are mainly used to structure a model. In contrast to atomic subsystems, the hierarchical structure of a virtual subsystem is flattened out by the Code Generator. As a result, it makes no difference to the generated code whether a block is located in a virtual subsystem or in the parenting subsystem. The code that is generated for a virtual subsystem is embedded in the code of its parenting subsystem. This is also true for nested virtual subsystems. The code generated for blocks inside a virtual subsystem can interleave with code generated for blocks outside it.

Atomic subsystems (code block)

Atomic subsystems are a powerful feature of MATLAB/Simulink. Unlike virtual subsystems, they influence the block execution order. The code generated for atomic subsystems does not interleave with code generated for blocks outside them. By default, the resulting code resembles an unconditionally inlined function, which is called a code block.

Note

- An atomic subsystem is always implemented as a code block.
- For the TargetLink Code Generator, atomic subsystems always have direct feed-through from the input to the output.
- Simulink automatically specifies enabled and/or triggered subsystems as atomic and disables the *Treat as atomic unit* checkbox for them, so the specification cannot be changed.

User-defined subsystem (C function)

If you add a TargetLink **Function** block to an atomic subsystem, TargetLink treats it as a user-defined subsystem. Usually, a user-defined subsystem is represented by a separate C function, called a user-defined function below. However, user-defined functions can be inlined if the selected **Optimization** function class property in the Data Dictionary Manager allows this. If you allow function inlining, TargetLink generates a code block instead of a C function.

Related topics

Basics

Basics on Optimizing Functions via Property Values..... 524
Basics on the TargetLink Data Dictionary (TargetLink Data Dictionary Basic Concepts Guide)
Defining the Function Interface..... 241

HowTos

How to Implement Subsystems as Functions..... 228

References

Subsystem Block (TargetLink Model Element Reference)

How to Implement Subsystems as Functions

Objective

To prevent a subsystem from being inlined in the generated production code, you must add a Function block to the desired subsystem.

Preserving the block diagram's hierarchy

The Function block can also be added to triggered and enabled subsystems. Thus, you can use the Function block to preserve the block diagram's hierarchy in the generated code.

Note

User-defined subsystems have priority over some optimization strategies and are basically represented as C functions. However, they can be inlined if the selected Optimization function class property in the Data Dictionary Manager allows it. If you allow function inlining, TargetLink generates a code block instead of a C function.

Precondition

If you include a Function block in a subsystem, you should declare the subsystem as atomic. Otherwise the direct feed-through behavior in MIL simulations could be different to that in SIL simulation and PIL simulation mode.

Method

To implement a subsystem as a function

1 Add a TargetLink Function block to the desired subsystem.

Result

The TargetLink **Function** block instructs the Code Generator to implement the subsystem as a C function.

Related topics

Basics

Basics on Function Interfaces..... 241
Basics on Optimizing Functions via Property Values..... 524
Basics on Using Subsystems..... 226

HowTos

How to Select Function Classes..... 233
How to Specify Function Interfaces via Function Parameters..... 242

References

Function Block (📖 TargetLink Model Element Reference)
Subsystem Block (📖 TargetLink Model Element Reference)

Customizing the Appearance of Functions via Subsystems and Function Classes

Introduction

You can use subsystems and function classes to customize the appearance of functions in the generated production code.

Where to go from here

Information in this section

Basics on TargetLink Subsystems and Function Classes..... 230
How to Select Function Classes..... 233
How to Assign Function Classes to Implicit Functions..... 234
How to Specify Subsystem Interfaces..... 236

Basics on TargetLink Subsystems and Function Classes

Defining the appearance of a function

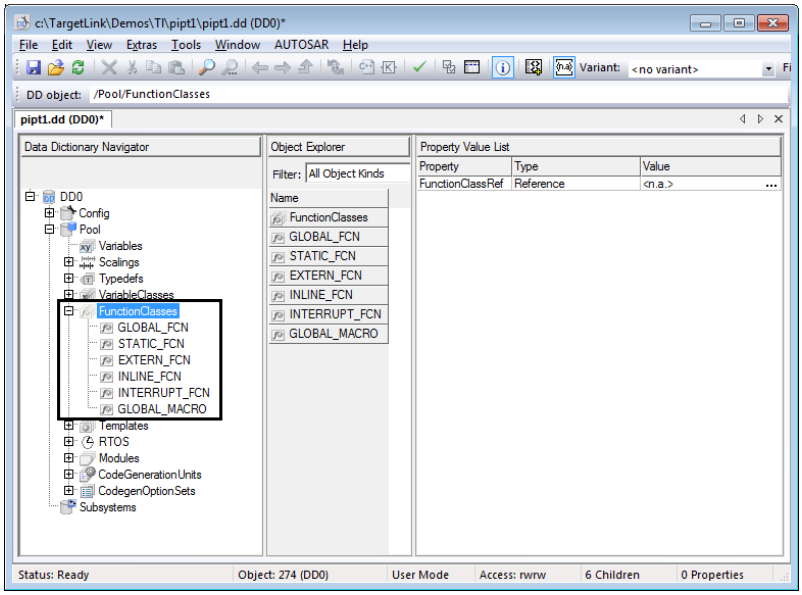
A function class is a set of properties that define the appearance of a function associated with the class in the generated production code.

You can modify a generated function by means of the function class to

- Define whether the function is visible in other modules
- Export or import the function to/from other modules
- Make the target/host compiler inline the function code in the calling function(s)
- Control how the Code Generator treats the function during optimization or inlining

Accessing function classes

Function classes are defined in the TargetLink Data Dictionary, where each function class object corresponds to one function class definition. You can define function classes via the Data Dictionary Manager or using API functions.



Categorizing function classes

When you open the `/Pool/FunctionClasses` node, you find several predefined function class objects. Each function class object has a fixed set of properties that describe how the Code Generator generates a function associated with the class.

Note

All the predefined DD FunctionClass objects are only examples that might not match your use cases.
We strongly advise to create your own DD FunctionClass objects that fit your actual use cases. Duplicate (**CTRL + D**) any of the predefined objects as a starting point. Do not change the predefined objects themselves.

Function classes differ with regard to their property values (property/value pairs) and can roughly be categorized according to implementation aspects.

Function Class Object	Description
`GLOBAL_FCN`	Global function(s) (exported to other modules)
`STATIC_FCN`	Static function(s) (invisible to other modules)
`EXTERN_FCN`	External function(s) (imported from another module)
`INLINE_FCN`	Function(s) that are inlined; a compiler directive is added to the function's definition
`INTERRUPT_FCN`	Interrupt service function(s)
`GLOBAL_MACRO`	Functions implemented as macros

Tip

Function classes have the same structure as variable classes. For details on variable classes, refer to Details on Preconfigured VariableClass Objects in the Data Dictionary on page 90.

Implicit function classes

Implicit functions are functions TargetLink creates internally, for example, to initialize the model, to evaluate look-up tables or as subfunctions of a Stateflow machine. By default, all functions are implicit functions, except for the model step functions (task functions), and cannot be accessed directly. For certain applications, however, this is mandatory anyway, which is why TargetLink provides predefined function class templates that are used for implicit functions automatically. For details, refer to How to Assign Function Classes to Implicit Functions on page 234.

Tip

Implicit functions generated for enabled, triggered or function-call subsystems can become user-defined functions if a Function block is inserted into the subsystem.

Related topics

Basics

Basics on Using Subsystems..... 226
Basics on Using Templates..... 203
Details on Preconfigured VariableClass Objects in the Data Dictionary..... 90

HowTos

How to Assign Function Classes to Implicit Functions..... 234
How to Embed External Functions in a TargetLink Subsystem..... 360
How to Select Function Classes..... 233
How to Specify Subsystem Interfaces..... 236

References

Function Block (📖 TargetLink Model Element Reference)

How to Select Function Classes

Objective

You must select a function class to define how a function is treated by TargetLink. Function classes can be set via the corresponding block dialog, with the TargetLink API or with the Property Manager.

Preconditions

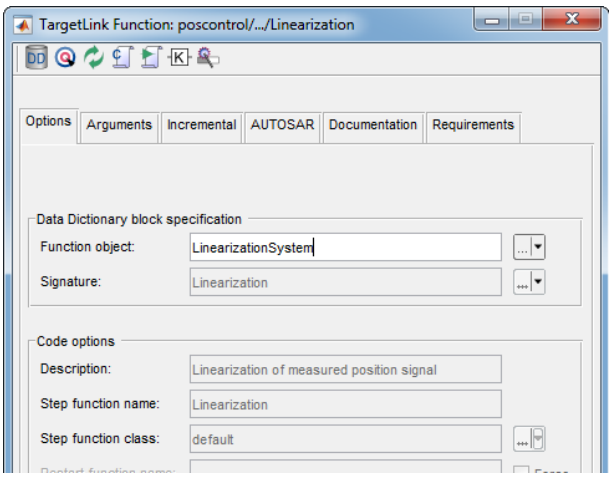
- The controller subsystem in the POSCONTROL demo model located in `<DocumentsFolder>\dSPACE\TargetLink\<Version>\Demos\poscontrol` is open.
- The DD project file belonging to the model is open.

Method

To select a function class

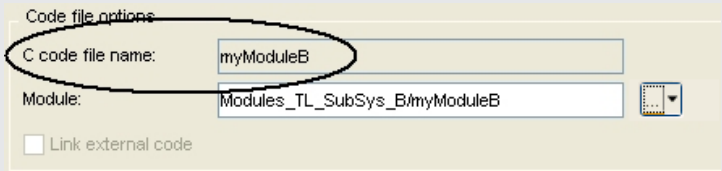
1. Double-click the Linearization subsystem to open it.
2. Double-click the Function block to open the block dialog.

   The Options page is displayed. If a DD Function object from the Data Dictionary is referenced, all Code options are greyed. If the step function class is set to default, the function is not inlined by TargetLink.



Tip

If you want the code for the function to be created in a separate C code file, you have to define its name. You can do this via the **ModuleRef** property in `/Pool/ModelDesign/Blocks/<Block>` or, if no DD **Function object** is referenced, as shown below.



3 Click the **OK** button to confirm your selection.

Result

The function class is selected.

Related topics

Basics

Basics on TargetLink Subsystems and Function Classes..... 230

HowTos

How to Assign Function Classes to Implicit Functions..... 234
How to Implement Subsystems as Functions..... 228
How to Specify Subsystem Interfaces..... 236

References

Function Block (📖 TargetLink Model Element Reference)
Subsystem Block (📖 TargetLink Model Element Reference)

How to Assign Function Classes to Implicit Functions

Objective

For certain applications the Code Generator defaults for implicit functions might generate code that does not meet your requirements. If so, you can use templates to modify the defaults within certain limits.

Usage of implicit functions

Besides functions that are defined explicitly, for example, by a **Function** block, TargetLink also generates other functions internally, for example, to initialize the model, to evaluate look-up tables or as subfunctions of a Stateflow machine. The properties of these implicit functions are defined by default function classes.

Note

Default function classes also apply to explicitly defined functions if their function class entry is left to **default**.

Adapting the default settings

If necessary you can easily adapt the function class templates to your specific project or company style by referencing another function class. Function classes, either predefined or ones you have created, are available in `/Pool/FunctionClasses` in the Data Dictionary Manager.

Note

- A predefined function class must not necessarily have been predefined by TargetLink.
- A **Function** block has priority over default function class settings.

Method

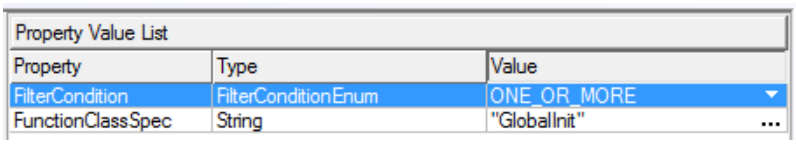
To assign function classes for implicit functions

1. In the `/Pool` area of the Data Dictionary Navigator, expand the `Template` subtree and then click the `FunctionClasses` object.
2. Click the function class template object that best describes the implicit function you want to assign a function class to.
3. Click the Filter object to display the filter properties in the property value list.

   Tip

   To display all the available properties, make sure that you have activated the **Show unset properties** button in the toolbar.

4. In the property value list, open the **FilterCondition** drop-down list and select the condition that must apply.
5. Click the **FunctionClassSpec** browse button and enter the name of the filter criteria.



Property Value List		
Property	Type	Value
FilterCondition	FilterConditionEnum	ONE_OR_MORE
FunctionClassSpec	String	"GlobalInit" ...

Note

You need to know the name of the filter criteria to be applied in advance. For details, refer to Filter Criteria of the FunctionClassTemplate Object. A default DD project file contains preset filter criteria for the templates that are available in the **TargetLink Data Dictionary**.

6 In the Data Dictionary Navigator, click the Settings object.

7 In the property value list, specify the function class you want to use for the specified type of implicit function.

By setting an already defined function class as the FunctionClass property, you define the most important properties for the implicit function. Initially the **default** class is assigned to all default function classes. This instructs the Code Generator to select the most suitable implementation.

Result

You have now specified the filter rule and defined which function class is to be used for a specific type of implicit function.

Related topics

Basics

Basics on Using Templates..... 203

HowTos

How to Define New Variable Classes..... 93
How to Select Function Classes..... 233
How to Specify Subsystem Interfaces..... 236

References

Function Block (📖 TargetLink Model Element Reference)
Subsystem Block (📖 TargetLink Model Element Reference)

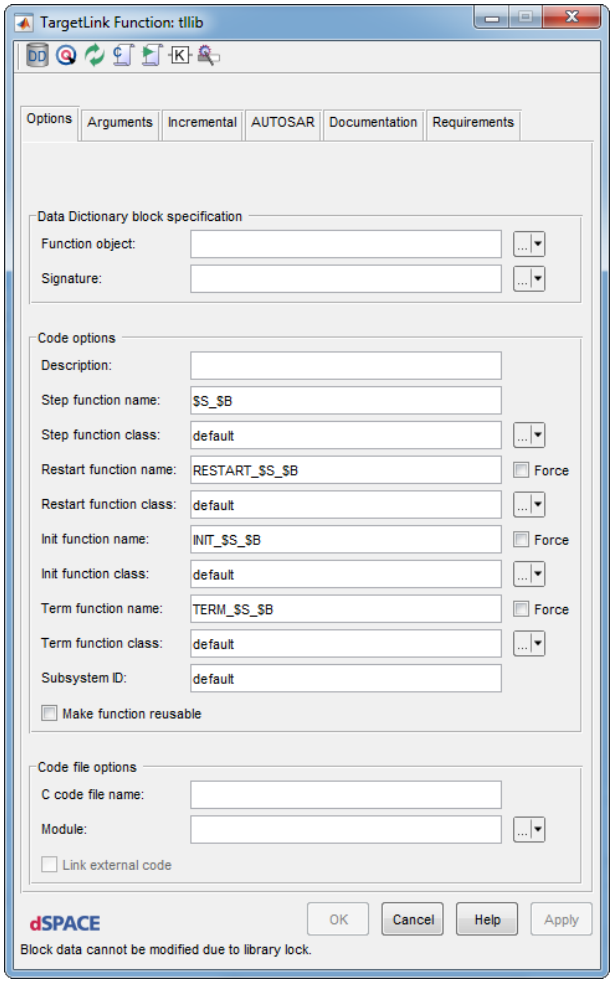
How to Specify Subsystem Interfaces

Objective

Once you have instructed the Code Generator to implement a subsystem by C functions, you have to specify which functions are to be generated for the subsystem, i.e., you have to specify the C function subsystem interface precisely. This is important for incremental code generation or integration of external subsystems.

Defining names and function classes

In TargetLink, you can specify the subsystem interface, that is, the names and function classes used for the step function or the initialization function, to name just two examples. This is done on the Options page in the **Function** block dialog. After you have finished the specification, the function is a user-defined function.



For details on the particular function block controls, refer to the **Options** page of the **Function Block**.

If you specify a separate function class for one of the functions shown above, note the following points:

- Restart function

  TargetLink uses the restart function to initialize global and static global variables if the **RestartFunctionName** property of their variable class is set. The property value can be `default` or any other fixed entry. If you leave it empty, TargetLink does not create a restart function for variables of the class. Note that the **RestartFunction** is active only, if the subsystem is a TargetLink root system or an external system (step function class set to extern function), or if it is configured for incremental code generation.

- Initialization function

  The initialization function is generated only if it is required for the C code of the model. TargetLink uses it to reset the states of enabled, triggered, and If-Action subsystems if required. This is the case if you set the **States when enabling** property to **reset** in the Simulink **Enable**, **Trigger**, or **Action Port** block of the called subsystem. The initialization function in the TargetLink

production code simulates the resetting of states to their initial conditions in Simulink.

Note

- For enabled subsystem whose States when enabling property is set to reset and that contains a TargetLink subsystem without trigger or enable ports, but with a Unit Delay block, Simulink resets the state of the Unit Delay block in MIL simulation mode as soon as the enabled-subsystem becomes active again outside the TargetLink subsystem. However, TargetLink neither generates an init function for the TargetLink subsystem nor does it reset the state in SIL simulation mode. Therefore, MIL and SIL simulation results might differ.

  To avoid simulation differences, place a Function block on the top level of the TargetLink subsystem and select the Force checkbox for the initialization function.
- The Force checkbox is also evaluated when you perform incremental code generation. For details on this setting, refer to the Options page of the Function Block. By selecting the Force checkbox, you instruct the Code Generator to generate an initialization function, even if it is empty.

Precondition

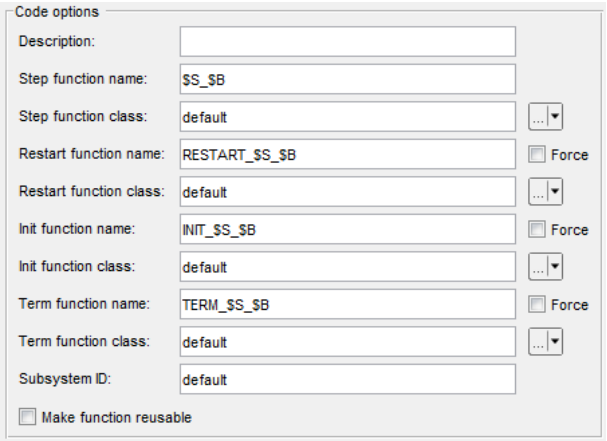
The subsystem to be represented as a C function must contain a TargetLink Function block.

Method

To specify the subsystem interface

1 Double-click the Function block to open its dialog.

2 In the Code options group box on the Options page, specify the functions according to your needs.



Tip

You can also set Function Classes with TargetLink's API or the Property Manager.

3 In the **code file options** group box, enter a file name if the function code is to reside in a separate file.

The file name must not have an extension, since a C file (`*.c`) and an associated header file (`*.h`) are generated.

Note

When you specify a user-defined function, keep the following function dependencies in mind:

- STATIC_FCN <-> code file
  A function declared as a STATIC_FCN is accessible only in the code file it is defined in. It is not visible to other modules or functions. If you try to call it from within another module, an error message will result.
- INLINE_FCN <-> code file
  With regard to the function body, which must be in the code file where it is called, inline functions are similar to static functions. Depending on whether the optimization option of the target/host compiler is enabled, the function code is directly merged into the calling function, avoiding the function-call overhead.

Note

The **Function** block lets you reference a Data Dictionary module object (`/Pool/Modules`), which the function code is to reside in. The reference overtrumps and disables the **C code file name** option.

Result

You have defined the names and function classes for all functions to be generated, that is, you have specified user-defined functions.

Tip

You can also define names and function classes for Stateflow Charts. For details, refer to Basics on Controlling the Function and Module Structure for Charts (TargetLink Preparation and Simulation Guide).

Related topics

Basics

Basics on Controlling the Function and Module Structure for Charts (📖 TargetLink Preparation and Simulation Guide)
Details on Controlling the Generation of Initialization, Termination and Restart Functions for External Step Functions by Using Function Classes..... 368

HowTos

How to Embed External Functions in a TargetLink Subsystem..... 360

References

Function Block (📖 TargetLink Model Element Reference)

Defining the Function Interface

Introduction

You must define the function interface to determine how signals and parameters are exchanged between different functions of the generated code.

Where to go from here

Information in this section

Basics on Function Interfaces..... 241

How to Specify Function Interfaces via Function Parameters..... 242

Basics on Function Interfaces

Defining the function interface

When code is generated for a subsystem, the interface of the function has to be defined. This interface describes how to pass the inputs and outputs to the generated code. Input signals can be passed to the function, and the values calculated for the output signals to the embedding subsystem can be returned as follows:

- Via global variables
- Via parameter passing

Tip

In TargetLink, the function interface, i.e., the inputs and outputs of a TargetLink subsystem, which is specified as a function, is generated via global variables if the **default** variable class is selected. Variable classes for subsystem inputs and outputs let you create a specific function interface style (for further information refer to Basics on Variable Classes on page 88). This helps you overcome the increased execution time with parameter passing and lets you meet company coding style guidelines.

For reused functions, however, the default is implemented using function parameters.

Function Interfaces via Parameters

TargetLink supports both value parameters (call-by-value) and reference parameters (call-by-reference) for input signals. Only reference parameters can be used for output signals. A value parameter is implemented by passing a copy of the value to the function code. A reference parameter is a pointer to a defined variable in the function code.

Parameter passing

Both the value parameter and the reference parameter require that the value or address of the parameter is passed to the function, which is done by stack or register copy operations prior to the call instruction. The advantage is that the signal data flow from the graphical model is directly reflected by the function implementation. Each input and output has a corresponding function parameter and thus provides a clear interface. For subsystems with numerous inputs or subsystems that have a high degree of nesting, this approach is not the best because the additional overhead for parameter passing significantly increases the execution time, code size and stack size of the generated code.

Related topics

Basics

Basics on Variable Classes..... 88

How to Specify Function Interfaces via Function Parameters

Objective

You must specify the function interface via function parameters to implement a subsystem by a function that communicates with the embedding subsystem via parameters.

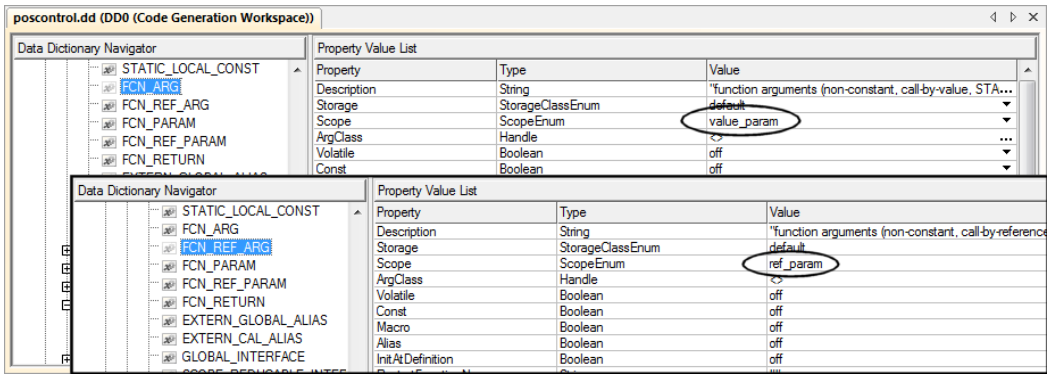
You can specify function interfaces of any level of nested subsystems.

Method

To specify the function interface via function parameters

1. Insert a TargetLink **InPort**/**OutPort** block or replace or enhance any Simulink Inport and Outport blocks with/to TargetLink **InPort**and **OutPort** blocks.
2. Double-click the **InPort** and **OutPort** block(s) to open the corresponding block dialog(s) and change to the **Output** page.
3. Click the Class browse button and select the **FCN_ARG** variable class for all TargetLink **InPort** block(s), and **FCN_REF_ARG** for all TargetLink **OutPort** block(s), that you want TargetLink to create a parameter for in the function call.

For these variable classes the Scope is set to value_param or ref_param in the Data Dictionary project file (`*.dd`).



Tip

In TargetLink, you can also implement block parameters as additional function parameters. You can use the call-by-value and the *call-by-reference* parameter passing mechanisms for block parameters.

4 Start the code generation process.

5 In the Name field (in the Production code scaling options group box) of the corresponding TargetLink interface block, enter the name for the function parameter.

Result

The identifier specified in the Name field is used for the function's formal parameter, i.e., the parameter appearing in the function definition or declaration.

The name of the actual parameter, i.e., the variable appearing in the function call statement is derived from the specification of the formal parameter by default. If required, the name of the actual parameter can be specified in the Name field after the name of the formal parameter separated by a colon (formal_name_spec:actual_name_spec).

Note that the implementation of the actual parameter is controlled by the ArgClass variable class attribute. The name of the actual parameter is not applicable for function input signals, because it is determined by the input signal. The result may read as follows:

```
void controller(Int16 Sa1_Ref, Int16 * Sa1_UPI)
{
...
```

Related topics

Basics

Defining the Function Interface..... 241

HowTos

How to Embed External Functions in a TargetLink Subsystem..... 360

Examples

Example of Specifying Block Parameters as Input Arguments of Functions..... 114

References

InPort Block (📖 TargetLink Model Element Reference)
OutPort Block (📖 TargetLink Model Element Reference)

Centrally Specifying and Synchronizing Function System Signatures

Where to go from here

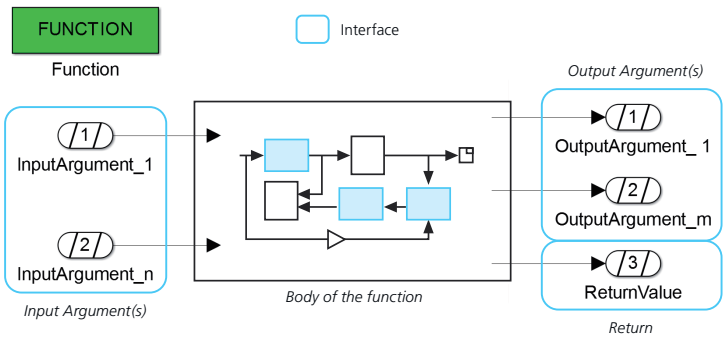
Information in this section

Basics on Centrally Specifying Function System Signatures..... 245
Basics on Synchronizing Function System Signatures..... 252
How to Specify Function Block Data..... 258
How to Specify System Signature Data..... 260
How to Synchronize Function System Signatures..... 263

Basics on Centrally Specifying Function System Signatures

Function subsystem

TargetLink lets you model function systems that serve as specification basis for the Code Generator. The characteristics of the function generated from the function system result from the system's key components. A function system looks like this:



Characteristics of a function subsystem

The signature of a function subsystem describes the characteristics of a function subsystem. These characteristics are relevant both on the model level and the code level:

Characteristics on the Model Level	Characteristics on the Code Level
The function subsystem's InPort/Bus Inport blocks	The C function's input arguments
The function subsystem's OutPort/Bus Outport blocks	The C function's output arguments and return value
The function subsystem's Function block	Characteristics of the C function, such as its name and variable class

Note

Keep in mind that variables of global scope belong to the signature of a function subsystem. However, they will not be visible in the signature of the generated C function according to C semantics.

Describing function subsystem signatures in the Data Dictionary

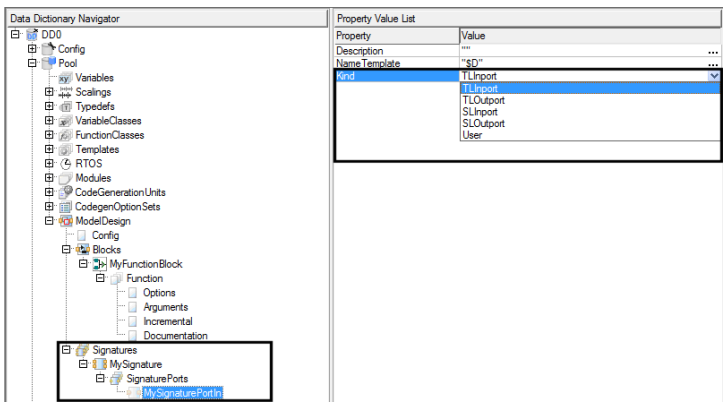
Specifying signatures You can specify the signature of a function subsystem in the Data Dictionary via DD Signature objects in the `Pool/ModelDesign/Signatures` object tree. These objects let you specify the function subsystem's name, description and ports.

Note

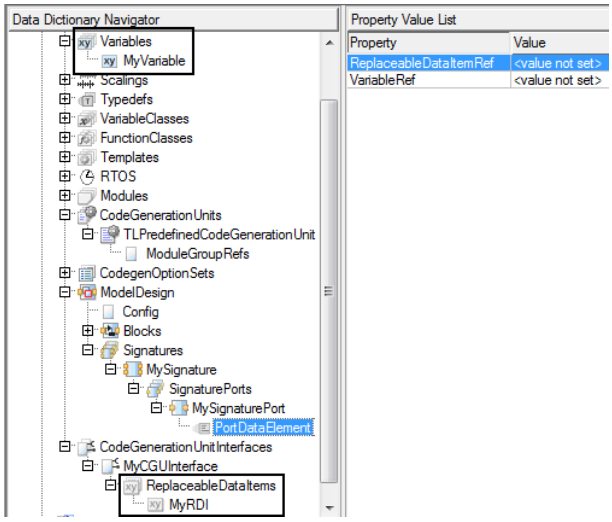
If the ModelDesign object tree does not exist, you must create it first. To do so, select Create ModelDesign from the context menu of the DD Pool object.

Specifying ports You can specify the function subsystem's ports via DD SignaturePort child objects of the DD Signature objects.

These objects let you specify the kind of the port block, its name and its description:



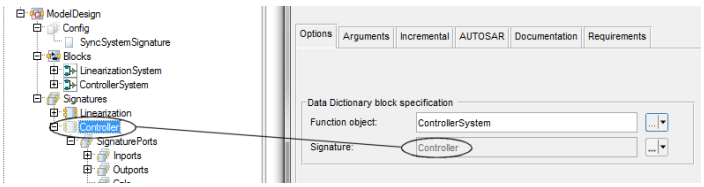
Specifying the port's data Optionally, you can further specify the port's data via its PortDataElement object. This object lets you reference *either* a Variable object *or* an ReplaceableDataItem object:



Bus ports You can specify TargetLink bus ports as follows:

- Referencing a **Variable** object, whose **BaseType** is set to `Struct`.
- Create component **PortDataElement** objects for each bus signal and reference a plain **Variable** object or **ReplaceableDataItem** object.

You can reference a DD **Signature** object from within a Function block in order to generate and synchronize the signature data from within the model.

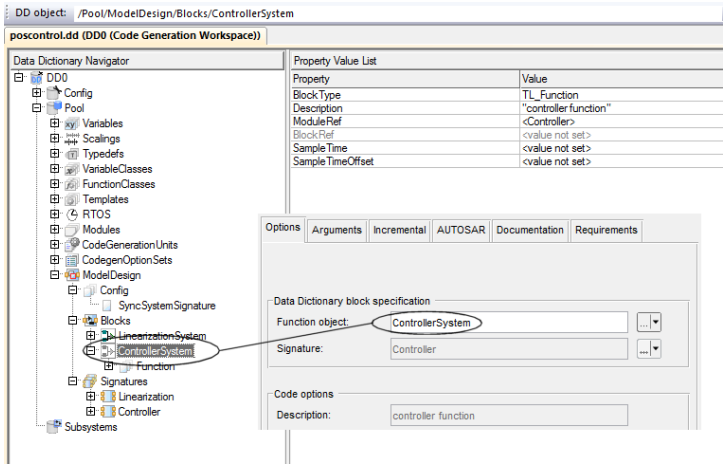


Refer to Basics on Synchronizing Function System Signatures on page 252.

Centrally specifying function block data

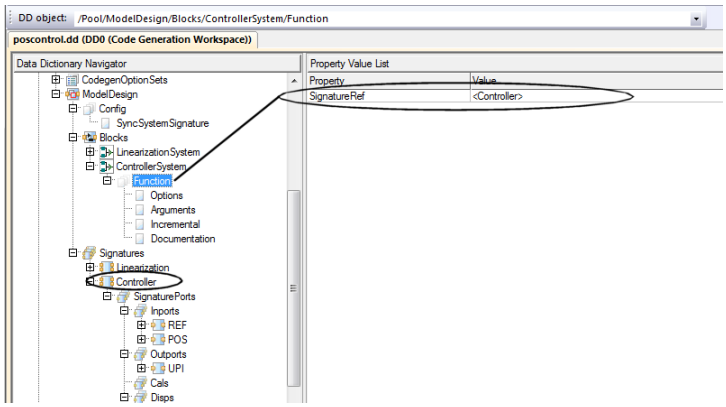
You can specify DD **Function Block** objects to centrally store a **Function** block's data in the Data Dictionary.

The DD **Function Block** object can be referenced from within a **Function** block in the model in order to synchronize the block data automatically. All settings you normally perform in the **Function** block are then controlled in the DD **Function Block** object's properties. This also applies for **Signature** objects that can be referenced in the DD **Function Block** object.

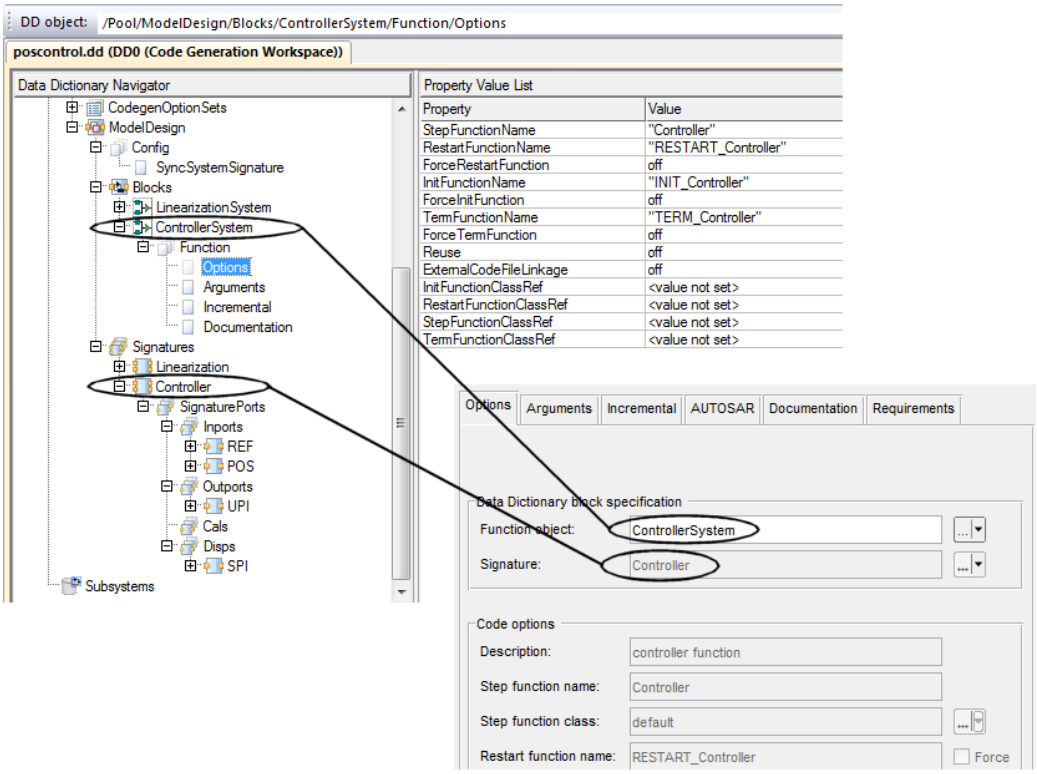


Linking DD Function Block and DD Signature objects

Each DD Function Block object can reference *exactly one* Signature object.



You can connect both DD objects to a **Function** block on the model level as shown in the following illustration:



For further information, refer to Basics on Synchronizing Function System Signatures on page 252.

Generating a function subsystem from the DD

You can generate a function subsystem from a DD **Signature** object.

Note

The parent model the function subsystem is generated in must be open.

The name of the function subsystem depends on the value of the **Signature** object's **NameTemplate** property.

Function block data The function block data of the function subsystem's Function block depends on the specification you made in the Data Dictionary:

Specification in the Data Dictionary	Result in the Generated Function Subsystem
Function Block Object References Signature Object	
Yes	The Function Block object that references the Signature object is also referenced at the Function block of the generated function subsystem. The function block data is synchronized with the referenced Function Block object.
No	The Function block of the generated function subsystem does not reference a Function Block object but has the block's default settings.

Port blocks The characteristics and number of the function subsystem's port blocks depends on the specification you made at the SignaturePort object(s) in the Data Dictionary.

TargetLink generates one port block for each SignaturePort object according to the following rules:

Specification in the Data Dictionary			Result in the Generated Function Subsystem
Signature Port Kind	**Port Data Element**		
	Exists	**Port Data Element Specification**	
`TLInport/TLOutport`	Yes	References a Variable object with BaseType ≠ `Struct`	TargetLink InPort/OutPort block that references the DD Variable object
		References a ReplaceableDataItem object	TargetLink InPort/OutPort block that references the DD ReplaceableDataItem object
		References a Variable object withBaseType = `Struct`	TargetLink Bus Inport/Bus Outport that references the DD Variable object
		Has components, each of which references a Variable object withBaseType ≠ `Struct`	TargetLink Bus Inport/Bus Outport block whose individual leaf bus elements each reference their corresponding DD Variable object
		Has components, each of which references a ReplaceableDataItem object	TargetLink Bus Inport/Bus Outport block whose individual leaf bus elements each reference their corresponding DD ReplaceableDataItem object
	No	-	TargetLink InPort/OutPort
`SLInport/SLOutport`	-	-	Simulink Inport/Outport block

Generating the function subsystem from a Function Block object You can also generate a function subsystem from a DD Function Block object.

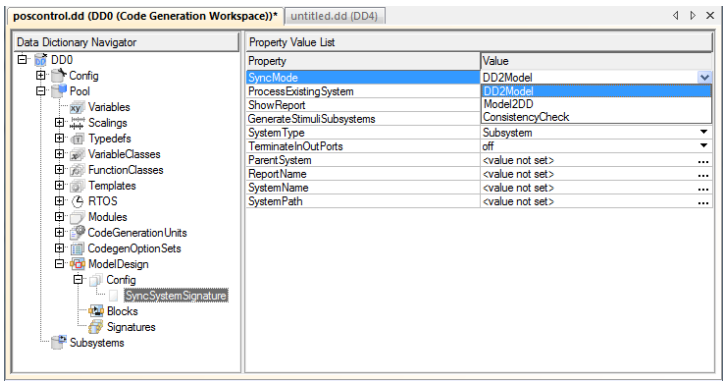
The function subsystem is generated according to the following rules:

Specification in the Data Dictionary Function Block Object References Signature Object	Result in the Generated Function Subsystem
Yes	A Function block referencing the DD Function Block object. Further function subsystem characteristics (Inports and Outports) as described by the referenced DD Signature object.
No	A Function block referencing the DD Function Block object. No Inports and Outports.

Configuring the generation process

You can configure the generation of the function subsystem via the properties of the following DD object:

`Pool/ModelDesign/Config/SyncSystemSignature`



For further information on the settings of the SyncSystemSignature object, refer to Basics on Synchronizing Function System Signatures on page 252.

For full reference information refer to the property value pairs of the **`tlSyncSystemSignature`** (📖 TargetLink API Reference) API function.

Related topics

Basics

Basics on Synchronizing Function System Signatures..... 252
Customizing the Representation of Buses in Production Code..... 467
Defining Abstract Interfaces for Model-Components..... 447
Defining the Function Interface..... 241

HowTos

How to Specify Function Block Data..... 258
How to Specify System Signature Data..... 260
How to Synchronize Function System Signatures..... 263

References

Synchronize System Signature (📖 TargetLink Data Dictionary Manager Reference)
tlSyncSystemSignature (📖 TargetLink API Reference)

Basics on Synchronizing Function System Signatures

Use cases for synchronizing function system signatures

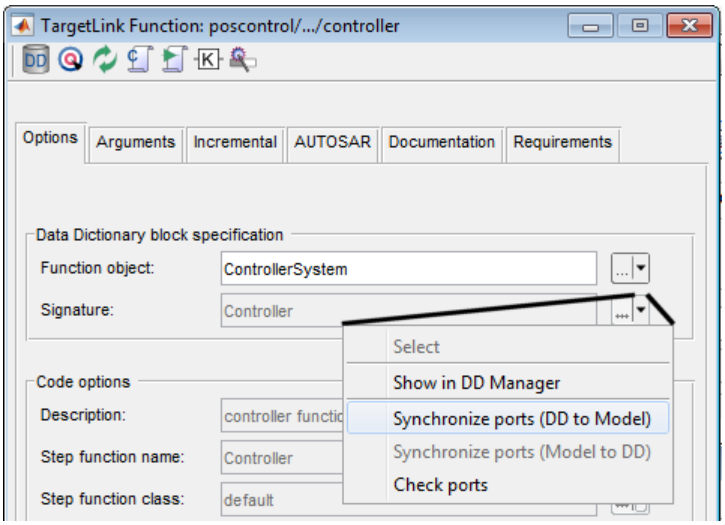
Synchronizing function system signatures means keeping the data of the interface (port data) specified in the DD Signature object and of the modeled interface of the function system in step. Even if no data exists on the model side or in the Data Dictionary, you can generate it automatically with the synchronization functionality. The following use cases apply:

- You want to centrally manage and deliver signature data that is already specified in the Data Dictionary. You can synchronize the data with the model (**DD to model**). If the signature data does not exist in the model yet, it is modeled automatically. If the model data or the DD signature data changed, you can keep them in step by synchronizing the signature data. Refer to How to Synchronize Function System Signatures on page 263.
- If you already specified signature data in the model, you can synchronize it with the Data Dictionary (**Model to DD**). This is possible only if *no* DD Signature objects exist in the Data Dictionary. Refer to How to Specify System Signature Data on page 260.
- 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**). Refer to Performing a consistency check before synchronizing.

Synchronizing data via the API, Data Dictionary Manager or model user interface

There are three ways to generate or synchronize function system signatures:

- Via a Function block in the model:

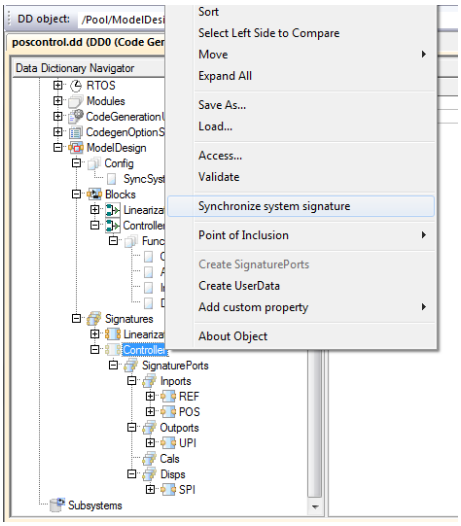


Note

Unlike the **DD to model** functionality that provides a true synchronization, the **Model to DD** functionality can create the signature port objects only once. The **Model to DD** functionality is unavailable if DD **Signature** objects already exist.

For details, refer to How to Synchronize Function System Signatures on page 263.

- Via the **Synchronize system signature** context menu command in the Data Dictionary Manager. The command is available for **Function block** objects and **Signature** objects.



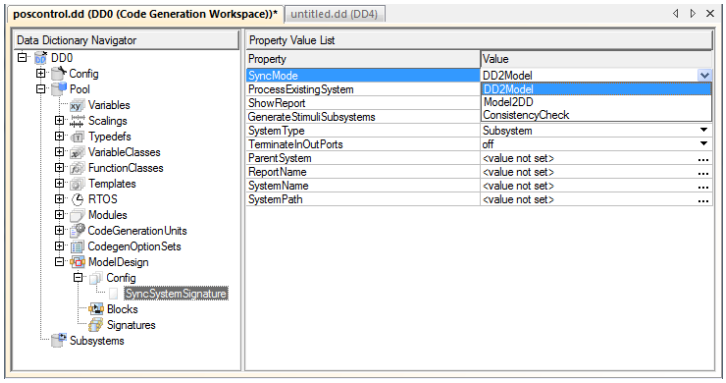
- Via the **tlSyncSystemSignature** API function. For further information on using this API function, refer to the detailed command description.

```
% Create a new model MyModel with TLSubsystem MyTLSubsystem from the DD Signature object
% /Pool/ModelDesign/Signatures/MySignature.
tlSyncSystemSignature('SyncMode','DD2Model',...
   'DDSignature', '/Pool/ModelDesign/Signatures/MySignature',...
   'ParentSystem','MyModel',...
   'SystemName','MyTLSubsystem',...
   'SystemType','TLSubsystem');
```

Synchronization settings

The `/Pool/ModelDesign/Config/SyncSystemSignature` object's properties let you adjust the settings for the synchronization. This applies to all the methods described above. You can select the synchronization mode, such as, **DD2Model** or show or hide the report. The settings you make are the default settings for the related **tlSyncSystemSignature** API function, unless defined otherwise.

For a complete description of the settings, refer to tlSyncSystemSignature (📖 TargetLink API Reference).



Note

If you start a synchronization from within a **Function** block, the settings of the **SyncSystemSignature** object's `SyncMode` and `ShowReport` properties are overwritten:

- `SyncMode` is controlled by the Function block's **Synchronize Ports (Model to DD)**, **Synchronize Ports (DD to Model)** or **Check ports** options.
- `ShowReport` is always set to **on**.

Performing a consistency check before synchronizing

You can perform a consistency check (**Check ports**) in order to compare the port data in the Data Dictionary and the model. No changes are performed. However, the consistency check helps you keep signature data consistent in both worlds, the model and the Data Dictionary. An HTML report is generated and is displayed in the MATLAB Web Browser. The report contains hyperlinks to the blocks and the DD objects. Differences are indicated with different colors.

System Signature Synchronization Report
Synchronization date: [illegible]
Synchronization mode: ConsistencyCheck
Model name: poscontrol
Data Dictionary: poscontrol.dd
DD Signature object: //DD0/Pool/ModelDesign/Signatures/MySignature

System: poscontrol/controller/Subsystem/controller/Linearization

DD SignaturePort Object	Port Block	To be Synchronized Parameter	Current Value	Expected Value	Annotation
MyInport	_				To be added.
_	POSITION				To be deleted.
MyOutport	_				To be added.
_	LIN.POS				To be deleted.

Best practice for synchronizing signature data

You are recommended to perform a consistency check and analyze the results before synchronizing data.

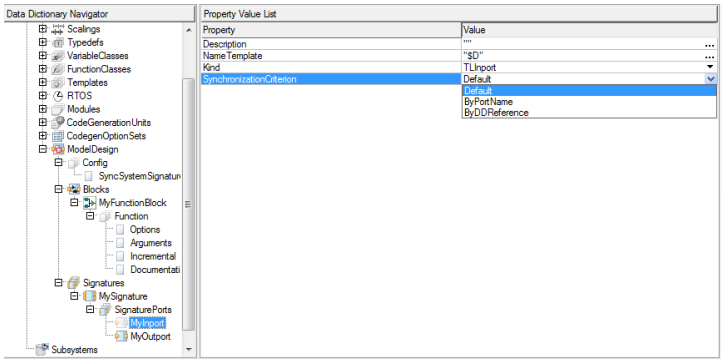
1. Start a consistency check using the **Check ports** context command.
2. Analyze the report and resolve inconsistencies.
3. Synchronize the ports.

Demo model

You can experiment with a demo model to observe TargetLink's synchronization mechanics. You can add, change or delete port data according to your needs. Refer to POSCONTROL (📖 TargetLink Demo Models).

Synchronization criteria

The following rules apply when you synchronize signature data. Keep in mind that TargetLink never deletes any blocks. Instead, it displays a warning in the report.



The **Default** setting uses the following rules:

- If a DD Reference exists, the **ByDDReference** rules apply.
- Otherwise, the **ByPortName** rules apply.

ByDDReference rules		
SignaturePort exists DD	**Block exists in Model**	**Action**
Yes	Yes	If the names of the SignaturePort and block differ, the block name is changed to the name of the **SignaturePort**.

ByDDReference rules		
SignaturePort exists DD	**Block exists in Model**	**Action**
Yes	No	New Block is added
No	Yes	TargetLink writes a warning to the report. You have to delete the block.
No	No	none

ByPortName rules		
SignaturePort exists in DD	**Block exists in model**	**Action**
Kind = SL(InOut)port	BlockType = TL(InOut)port	Block is enhanced
Kind = TL(InOut)port	BlockType = SL(InOut)port	Block is enhanced
VariableRef set	Variable not set , RDI not set	Block's Variable is set as specified in the SignaturePort object.
VariableRef set	Variable not set , RDI set	Block's Variable is set as specified in the SignaturePort object. Block's RDI reference is deleted.
VariableRef set	Different Variable set	Block's Variable is set as specified in the SignaturePort object.
VariableRef set	Identical Variable set	none
RDIRef set	RDI not set, Variable not set	Block's RDI reference will be set as specified in the SignaturePort object
RDIRef not set	RDI not set, Variable set	Block's RDI reference will be set as specified in the SignaturePort object. Block's Variable reference will be deleted
RDIRef set	Different RDI set	Block's RDI reference will be set as specified in the SignaturePort object
RDIRef set	Identical RDI set	none
VariableRef not set, RDIRef not set	Variable not set , RDI not set	none
VariableRef not set, RDIRef not set	Variable set	Block's Variable reference is deleted
VariableRef not set, RDIRef not set	RDI set	Block's RDI reference is deleted
No	Yes	TargetLink writes a warning to the report. You have to delete the block.
No	No	none

Customizing the system generation/synchronization process via hook scripts

TargetLink provides several hook script templates that you can use to derive M-script hook scripts. These are used as a callback mechanism for user-specific customization of certain TargetLink processes. You can customize the system generation/synchronization process with the following hook scripts:

Hook script name	Description
Stimulus creation	
tl_pre_create_stimuli_hook	Called before a stimuli subsystem is created
tl_post_create_stimuli_hook	Called after a stimuli subsystem has been created
Model to Data Dictionary	
tl_pre_add_ddsignatureport_hook	Called before adding a new DD SignaturePort object to the specified DD **Signature** object
tl_post_add_ddsignatureport_hook	Called after adding a new DD SignaturePort object to the specified DD **Signature** object
Data Dictionary to model	
tl_pre_add_systemsignatureport_hook	Called before an **InPort** or **OutPort** block is added to the function subsystem
tl_post_add_systemsignatureport_hook	Called after an **InPort** or **OutPort** block is added to the function subsystem
tl_pre_syncsystemsignature_hook	Called before the function subsystem generation/synchronization starts
tl_post_syncsystemsignature_hook	Called after the function subsystem has been created/synchronized
tl_pre_sync_systemsignatureport_hook	Called before synchronizing an existing port block with the corresponding DD SignaturePort object
tl_post_sync_systemsignatureport_hook	Called after synchronizing an existing port block with its corresponding DD SignaturePort object

For details on the hook scripts, refer to Function Subsystem Generation Hook Scripts (📖 TargetLink File Reference).

Related topics

Basics

Basics on Centrally Specifying Function System Signatures..... 245
POSCONTROL (📖 TargetLink Demo Models)

HowTos

How to Specify Function Block Data..... 258
How to Specify System Signature Data..... 260
How to Synchronize Function System Signatures..... 263

References

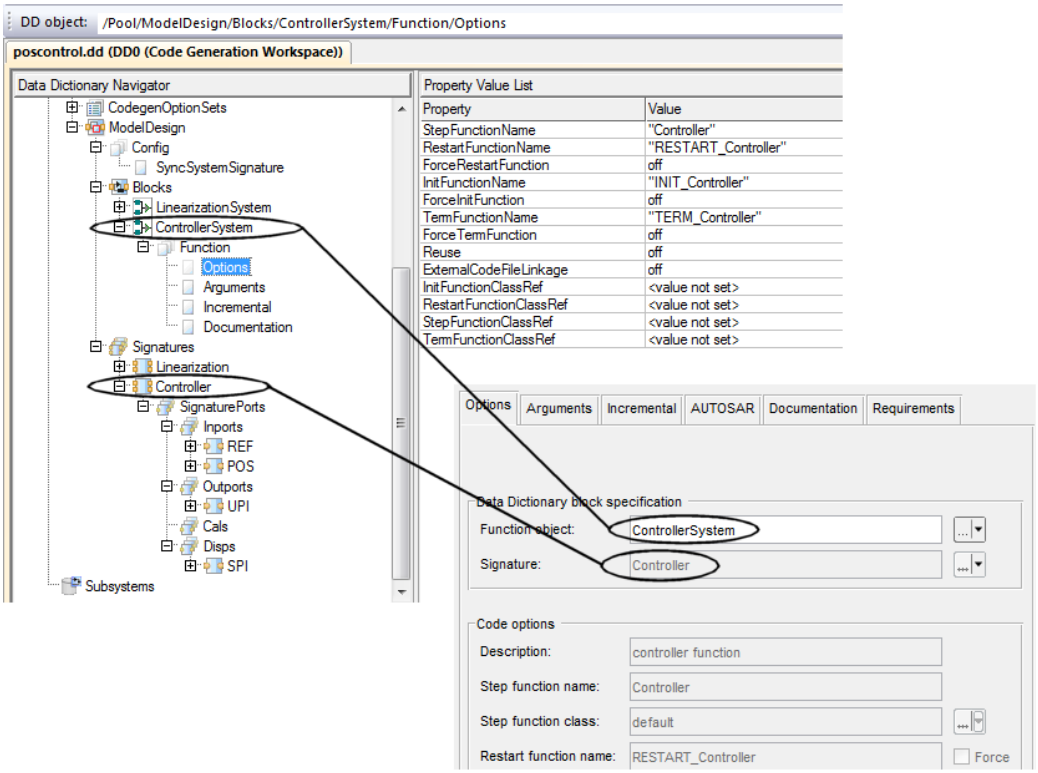
tl_post_add_ddsignatureport_hook (📖 TargetLink File Reference)
tl_post_add_systemsignatureport_hook (📖 TargetLink File Reference)
tl_post_create_stimuli_hook (📖 TargetLink File Reference)
tl_post_sync_systemsignatureport_hook (📖 TargetLink File Reference)
tl_post_syncsystemsignature_hook (📖 TargetLink File Reference)
tl_pre_add_ddsignatureport_hook (📖 TargetLink File Reference)
tl_pre_add_systemsignatureport_hook (📖 TargetLink File Reference)
tl_pre_create_stimuli_hook (📖 TargetLink File Reference)
tl_pre_sync_systemsignatureport_hook (📖 TargetLink File Reference)
tl_pre_syncsystemsignature_hook (📖 TargetLink File Reference)
tlSyncSystemSignature (📖 TargetLink API Reference)

How to Specify Function Block Data

Objective

In the Data Dictionary Manager, you can specify function block data in a DD **Function Block** object. The child objects of the DD **Function Block** object contain the elements of a TargetLink **Function** block (except AUTOSAR).

If you connect a DD **Function Block** object with a TargetLink **Function** block, the function block data is synchronized. You can then control the TargetLink **Function** block data in the connected DD **Function Block** object (the elements in the TargetLink **Function** block are grayed out as shown below).



You can create a DD **Function Block** object in two ways:

- Create a new DD **Function Block** object and specify the function block data for a TargetLink **Function** block
- Create and synchronize a new DD **Function Block** object from within an already existing TargetLink **Function** block

Precondition

The **ModelDesign** object tree exists in the **Pool** area.

Method 1

To create a new DD Function Block object and specify the Function block data

1 In the context menu of the `/Pool/ModelDesign/Blocks` object tree, select Create Function Block. This will create a DD Function Block object in the Data Dictionary.

> **Note**
>
> You can also create a Function Block object as a child element of a BlockGroup object.

2 Specify the function block data in the DD Function Block object's child objects.

3 In the model, open the Function block to be linked to the DD Function Block object.

4 In the Function object edit field of the Options page, select the DD Function Block object.

Method 2

To create and synchronize a new DD Function Block object from within an already existing TargetLink Function block

1 Open a TargetLink Function block that contains the data you want to synchronize.

2 Enter a name or a relative path for the new DD Function Block object in the Function object edit field of the Options page.

3 Press **Enter**.

The red name indicates that the DD Function Block object is not yet created.

4 Click Apply.

The DD Function Block object is created and the data is synchronized. The elements of the Function Block block are grayed out, because they are now controlled from within the DD Function Block object.

Result

You have specified function block data from within the Data Dictionary.

Related topics

Basics

Basics on Centrally Specifying Function System Signatures..... 245
Basics on Synchronizing Function System Signatures..... 252

HowTos

How to Specify System Signature Data..... 260
How to Synchronize Function System Signatures..... 263

How to Specify System Signature Data

Objective

You can specify the signature of a function subsystem in the Data Dictionary by using a DD **Signature** object. You can specify the ports of the subsystem within the **Signature** object.

In addition, if you create one DD Function Block object that references the specified DD Signature object, the inserted TargetLink Function block is linked and its data is synchronized with this Block object.

As an alternative, if you have modeled port data in a subsystem, you can also automatically generate the corresponding Signature objects containing the port data in the Data Dictionary (Model to DD). You can do this from within a Function block residing in the system containing the ports.

Creating a DD function block object

For information on DD **Function** block objects, refer to How to Specify Function Block Data on page 258.

Method 1

To specify a system's signature data from within the Data Dictionary Manager

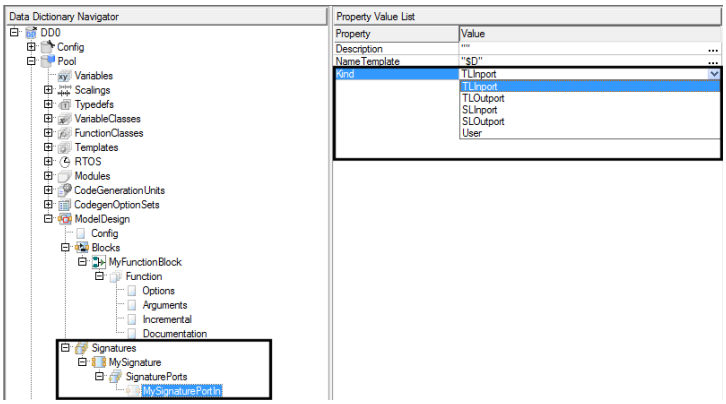
1. If the **ModelDesign** object tree does not exist, you must create it first. To do so, select **Create ModelDesign** from the context menu of the DD **Pool** object.
2. In the context menu of the `/Pool/ModelDesign/Signatures` object tree, select **Create Signature**. This will create a DD Signature object in the Data Dictionary.

   Note

   You can also create a DD Signature object as a child element of a DD SignatureGroup object.

3. In the context menu of the Signature object, select **Create SignaturePorts**.
4. In the context menu of the SignaturePorts object, select **Create SignaturePort**.

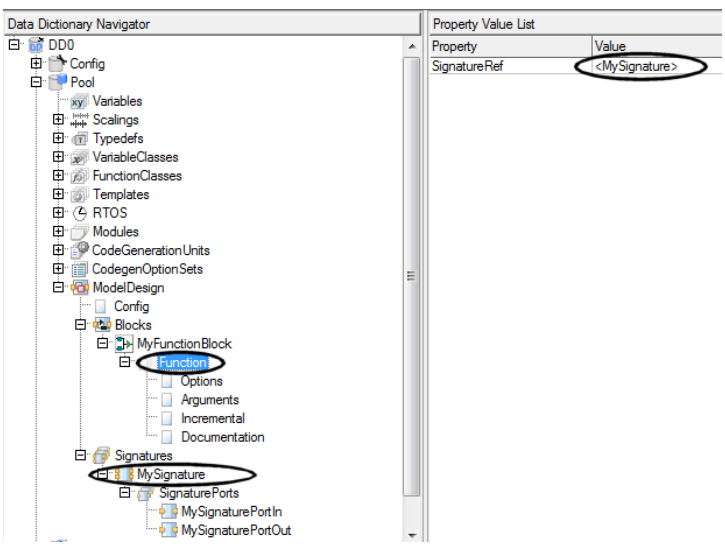
5 Specify the desired port block as shown below.



Create the ports according to your needs, for example, one TLInport and one TLOutport **SignaturePort** object.

6 You can now add the properties to configure the generation. If you do not specify the configuration, the default values apply as shown below:

7 Optionally, you can select a DD **Function Block** object (if already created). You must now specify the reference to the DD Signature object. In the context menu of the DD Function child object, specify the `SignatureRef` property. Select the DD **Signature** object as shown below.



Method 2

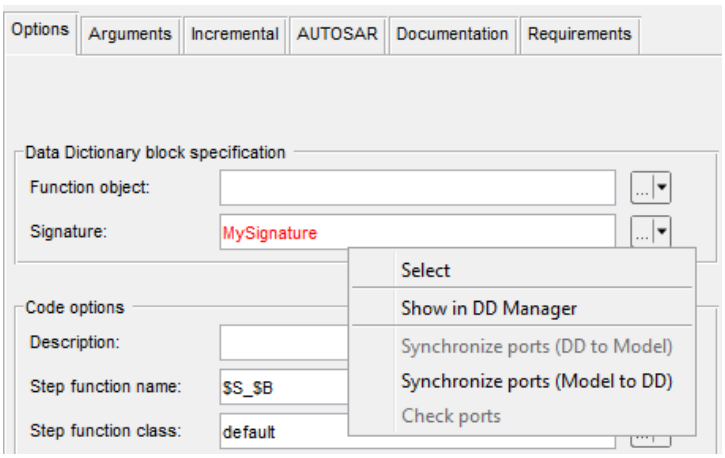
To specify a system's signature data from within the model (Model to DD)

1 Change to the Function block of the system you created port data for.

2 In the Function block, enter a name for the DD Signature object to be created.

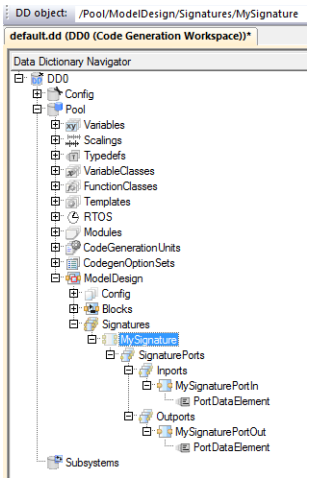
3 Press `Enter`.

The red name indicates that the DD Signature object is not yet created.



4 Select Synchronize Ports (Model to DD). Optionally, you can also reference a DD Function Block object *after* you have specified the DD Signature object.

The signature data is automatically generated in the Data Dictionary.



TargetLink API function

You can also work with the `tlSyncSystemSignature` TargetLink API function . For details, refer to tlSyncSystemSignature (📖 TargetLink API Reference).

Result

You have specified a subsystem's signature.

Related topics

Basics

Basics on Centrally Specifying Function System Signatures..... 245
Basics on Synchronizing Function System Signatures..... 252

HowTos

How to Specify Function Block Data..... 258
How to Synchronize Function System Signatures..... 263

How to Synchronize Function System Signatures

Objective

You can synchronize function system signature data from the Data Dictionary to the model (**DD to Model**). This allows you to centrally share signature port data to your team members.

Preconditions

In order to take over the signature port data to a model, the following preconditions must be met:

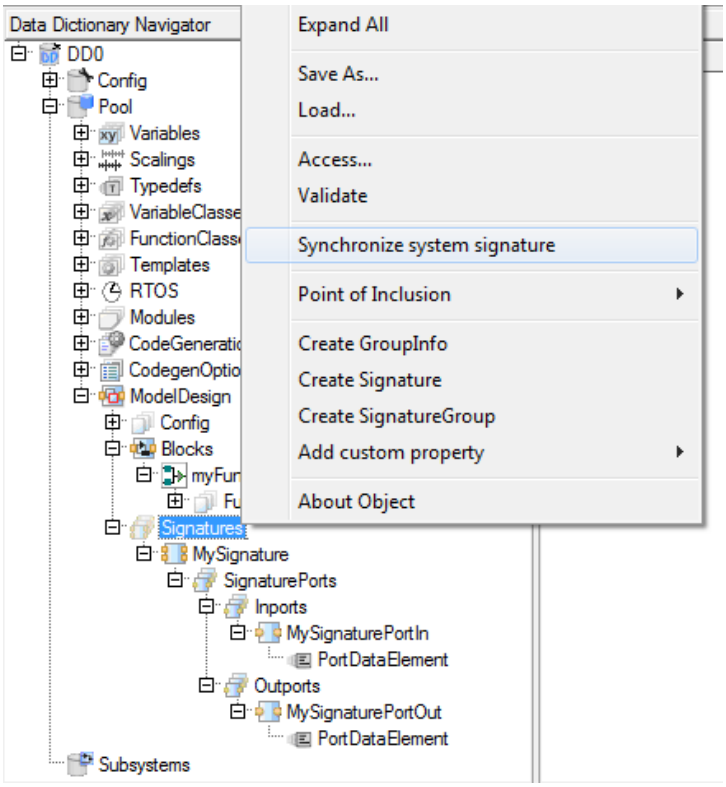
You have created a Signature object containing signature port data in the `/Pool/ModelDesign/Signatures` tree. For details, refer to How to Specify System Signature Data on page 260.

Method 1

To synchronize function system signature data from within the Data Dictionary

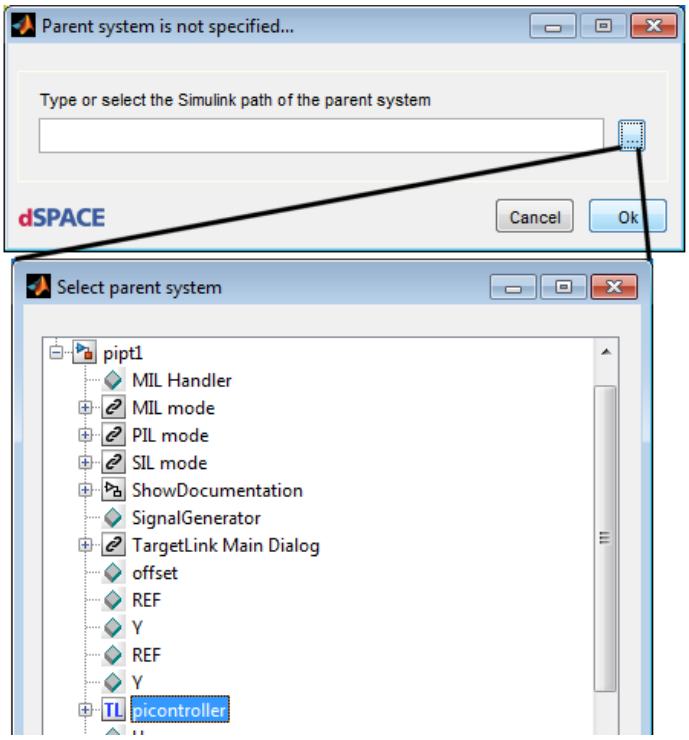
1. The `/Pool/ModelDesign/Config/SyncSystemSignature` object's properties let you adjust the settings for the synchronization. Set the properties according to your needs. You can select the mode of the synchronization, for example, **DD2Model** or show or hide the report. Depending on this settings, the synchronization rules apply. For details, refer to Basics on Synchronizing Function System Signatures on page 252.
2. Change to the `Pool/ModelDesign/Signatures` object tree.

3 In the context menu of the DD Signature object, select Synchronize system signature.



Optionally, you can perform a consistency check first. For details, refer to Basics on Synchronizing Function System Signatures on page 252.

4 If not already specified, you must type or select a parent system.



Depending on the synchronization criteria, the port data is synchronized with the model.

System Signature Synchronization Report
Synchronization date: [redacted]
Synchronization mode: DD2Model
Model name: untitled
Data Dictionary: default.dd
DD Signature object: //DD0/Pool/ModelDesign/Signatures/new_Signature

System: untitled/new_Signature

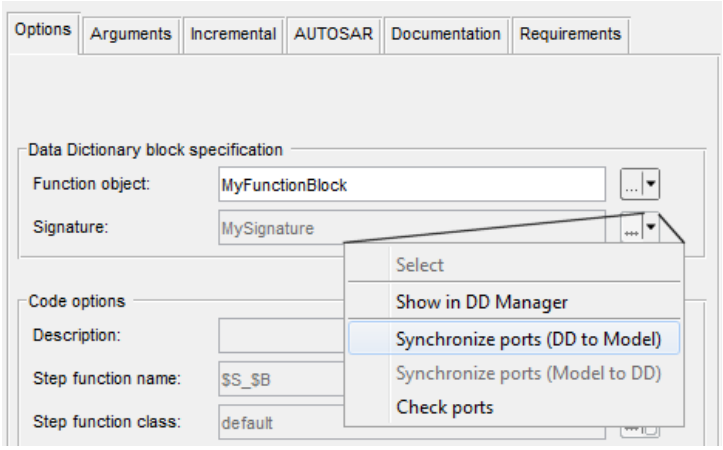
DD SignaturePort Object	Port Block	Synchronized Parameter	Previous Value	Current Value	Annotation
mySignaturePortIn	mySignaturePortIn				TL_InPort block added.
mySignaturePortOut	mySignaturePortOut				TL_OutPort block added.

Method 2

To synchronize function system signature data from within a Function block

1 Change to the Function block of the system to be synchronized with the port data from the Data Dictionary

2 Select Synchronize ports (DD to Model)



Optionally, you can perform a consistency check first. For details, refer to Basics on Synchronizing Function System Signatures on page 252.

Note

If you start a synchronization from within a Function block, the settings of the SyncSystemSignature object's `SyncMode` and `ShowReport` properties are overwritten:

- `SyncMode` is controlled by the Function block's Synchronize Ports (Model to DD), Synchronize Ports (DD to Model) or Check ports options.
- `ShowReport` is always set to on.

Depending on the synchronization criteria, the port data is synchronized. A report shows the synchronization details.

System Signature Synchronization Report
Synchronization date:
Synchronization mode: DD2Model
Model name: untitled
Data Dictionary: default.dd
DD Signature object: //DD0/Pool/ModelDesign/Signatures/new_Signature

System: untitled/new_Signature

DD SignaturePort Object	Port Block	Synchronized Parameter	Previous Value	Current Value	Annotation
mySignaturePortIn	mySignaturePortIn				TL_InPort block added.
mySignaturePortOut	mySignaturePortOut				TL_OutPort block added.

TargetLink API function

You can also work with the `tlSyncSystemSignature` TargetLink API function . For details, refer to tlSyncSystemSignature (📖 TargetLink API Reference).

Result

You have synchronized function system signature data.

Related topics

Basics

Basics on Centrally Specifying Function System Signatures..... 245
Basics on Synchronizing Function System Signatures..... 252

HowTos

How to Specify Function Block Data..... 258
How to Specify System Signature Data..... 260

Encapsulating Access to Data Signals and Parameters via Access Functions

Where to go from here

Information in this section

Introduction to Access Functions.....268
Implementing Access Functions.....274

Introduction to Access Functions

Where to go from here

Information in this section

Overview of Typical Use Cases for Access Functions.....268
Basics on Access Functions.....270

Overview of Typical Use Cases for Access Functions

Purpose of access functions

TargetLink provides access functions (AFs) as a way to encapsulate the access to data signals and parameters and abstract from the implementation via variables.

Stable interfaces

Access functions are usually an alternative to communication via global variables. They hide the actual variables from clients that only know the access functions. This allows exchanging the underlying variables without changing the interface that is used or provided by the functional production code.

Access functions are especially useful for implementing stable interfaces. No matter what the implementation is (single variable, struct member or union member, …), nothing changes on the client side. The client exclusively uses access functions to access variables or variable values.

Implementing access functions

Access functions can be implemented as functions or as macros:

- Function AFs break compile dependencies.

  The client code can be compiled against the interface and the implementation can be changed later.
- Macro AFs avoid function call overhead.

  Changes to macro AFs make recompiling all client code necessary.

Encapsulating the interface to external code (TargetLink production code as client)

The owner of an *external* interface can provide access functions that encapsulate read and write operations to the external variables. This lets the owner control how variables are accessed, e.g., by providing read-only or write-only interfaces.

To generate the client code with TargetLink, you specify access function encapsulation for the external variables in the Data Dictionary. Usually, the owner of the external interface provides the access functions in external modules. They allow you to read or change values of the underlying external variables. (Refer to External interfaces on page 292.) The external variables are used as block variables representing data signals or parameters in the Simulink model.

Hiding the implementation from TargetLink by calling external access functions can also be useful if TargetLink does not support the underlying implementation.

Note

When using access functions that are provided in external code at the interface of production code, you have read at inports and write at outports.

Encapsulating the production code's interface (external code as client)

You might want to hide the access to the variables of your model from *external code*, for example to prevent (accidental) modifications. You can do so by implementing access functions that the external code can use to read from or write to the interface of your model. The interface variables of one or more code generation units are specified in the model in the same way they are specified for an external interface. TargetLink then generates definitions of the access functions.

For this use case, TargetLink also generates the underlying variables. If the production code's accesses to the variables allow for it, the variables can be module-local, effectively preventing external access except via the provided function AF interface. The IgnoreAccessFunctionsInOwnModule Code Generator option can be useful in order to avoid function call overhead inside this module.

Note

When providing access functions in production code to use in external code, you have to write at inports and read at outports.

Implementing accesses with side effects

You can implement an access function as a macro in order to call external code that not only reads or writes variable values but also changes the state of a message queue or stack, for example. If an access function is implemented as a macro, it can be used as an adapter to existing functions with additional parameters or different parameter order.

Ease viewing and debugging of interfaces

Access functions are also helpful for tracing variable value changes and reading variable accesses at function interfaces.

Related topics

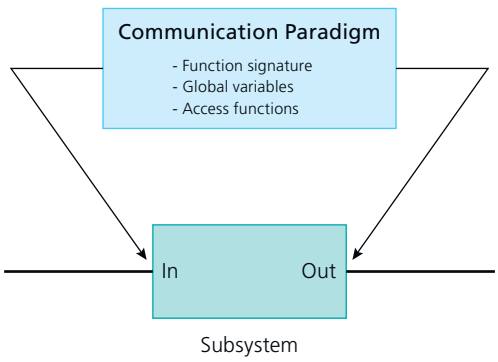
References

IgnoreAccessFunctionsInOwnModule (TargetLink Model Element Reference)

Basics on Access Functions

Access functions

Access functions (AFs) encapsulate the access to interface variables, which are similar to global variables. This means that they abstract and hide the way data signals and parameters are implemented, which lets you change the underlying variables or values without having to change the interface. That is, the production code implementing the model behavior stays the same. Access functions are possible for variables of global scope or external storage.



Access functions are primarily intended for variables that represent the interface of a subsystem:

- Output variables of **InPort** and **OutPort** blocks
- Data store variables
- Parameter variables

Two kinds of access functions

TargetLink offers two kinds of access functions:

- Read/write access functions
- Variable access functions

Read/write access functions

Read/write access functions are used to *encapsulate the instructions* for reading from/writing to a variable. You can use these access functions to get or set the value of the variable or one of its elements.

Example Consider the following example of a read/write access function:

Without Access Function	With Access Function

`x = In + 5.25F;`
…
`Sa4_Gain = 1.5F * x;`
	`SetX(In + 5.25F);`
	…
	`Sa4_Gain = 1.5F * GetX();`

Note

Read/write access functions can be implemented either as C functions or as C preprocessor macros.

Variable access functions

Variable access functions are used to *encapsulate the access* to a variable for reading or writing. Accessing the variable is possible by name or memory address.

Example of a variable access function

```
*(GetAddressX()) = In + 5.25F;
…
S14_Gain - 1.5F * *(GetAddressX())
```

Encapsulation by memory access is useful, for example, if the variable's address is to be passed as a function argument, because otherwise TargetLink creates an implicit variable and copies the value from and to the original variable (by calling its access functions).

TargetLink supports variable access functions that either return the address of the variable (address access) or the variable itself (direct access). Direct access means that access to a variable is directly mapped. Address access means that the variable's address is returned. This allows for the modification of the underlying storage area.

Note

Direct access functions can be implemented only as C preprocessor macros.

Degree of encapsulation

The degree of encapsulation depends on the access function's kind:

Degree of Encapsulation	Access Function Kind
High	Read/Write access function
Medium	Address access function
Low	Direct access function

This makes the access function kinds appropriate in different contexts:

Code Without Access Function	Read/Write Access Function	Address Access Function	Direct Access Function

`int16 v[5]`
`f(v); /* for reading */`
	Only possible with local value copy:
	`int16 _tmpu_v[5];`
	`for(AuxS32 = 0; AuxS32 < 5; Aux_S32 +1){`
	`  _tmpu_v[AuxS32] = ReadV(Aux_S32);`
	`}`
	`f(_tmpu_v);`
		Possible:
		`f(GetAddressV());`
			Possible:
			`f(AccessV())`

`int16 v[5]` `sizeof(v)`	Not possible	Not possible	`sizeof(AccessV())`

Example of a direct access function Suppose you want to use an access function to access the `MyTable` variable during the initialization of a variable with static storage duration. You therefore need an expression in production code that expands to the name of `MyTable` during compilation. This can be done by using a direct access function:

```
static const MAP_Tab1DS4I2T1563_a Sa2_Look_Up_Table_map = {
   25 /* Nx:  */,
   (const Int8 *) &(GetMyTable()[0]) /* z_table:  */,
   3 /* diffx:  */,
   -96 /* -12. */ /* startx:    Description: axis vector starting point
   LSB: 2^-3 OFF:  0 MIN/MAX:  -16 .. 15.875 */
};
```

Variable kinds

You have to associate access functions with one or more of the following variable kinds: matrix, scalar, structure, vector. That is, you have to tell TargetLink which variable kinds an access function specification applies to.

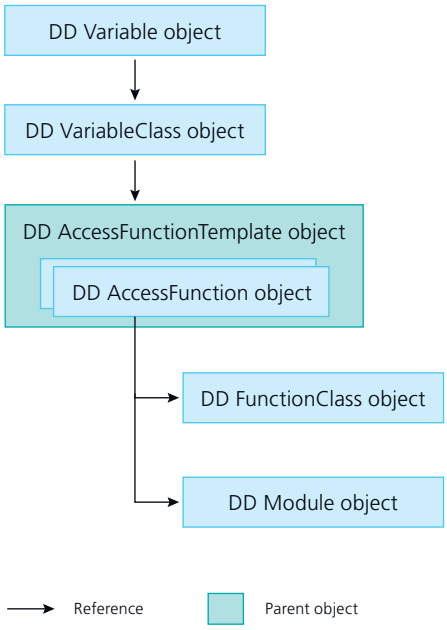
Additionally, you can configure how to propagate access function templates into structured variables in TargetLink.

Refer to Associating Access Functions With Variable Kinds on page 282 and Determining Access Function Kind According to the Use Case on page 276.

Specifying access functions

TargetLink groups access functions in access function templates. This is the case because access functions are used systematically for all the variables of a certain type. To specify an access function, you have to specify an AccessFunctionTemplate object in the Data Dictionary.

AccessFunctionTemplate and AccessFunction objects You specify that TargetLink uses the access functions belonging to a specific DD AccessFunctionTemplate object by referencing it at a DD VariableClass object. This ensures that all the variables derived from this variable class are subject to the access functions belonging to that DD AccessFunctionTemplate object.



Note

If a variable access cannot be replaced by the specified access function, the variable access stays in the generated production code and a warning message is displayed during code generation.

FunctionClass object Each DD AccessFunction object can reference a DD FunctionClass object to specify the access function's C attributes. This also lets you specify whether TargetLink should implement the access function as a C function or a C preprocessor macro.

For more information, refer to:

- Specifying Access Functions as C Functions on page 287
- Specifying Access Functions as Preprocessor Macros on page 289

Module object Each DD AccessFunction object can reference a DD Module object to specify the module in which to implement the access function. Refer to Mapping Access Functions to Modules on page 290.

Related topics

Basics

Basics on TargetLink Subsystems and Function Classes..... 230
Basics on Variable Classes..... 88
Details on Predefined DD Template Objects..... 209

HowTos

How to Create an Access Function Specification..... 275

References

Function Block (📖 TargetLink Model Element Reference)

Implementing Access Functions

Where to go from here

Information in this section

How to Create an Access Function Specification..... 275

Determining Access Function Kind According to the Use Case..... 276

Associating Access Functions With Variable Kinds..... 282

Changing Access Function Implementations Using Name Macros..... 286

Specifying Access Functions as C Functions..... 287

Specifying Access Functions as Preprocessor Macros..... 289

Mapping Access Functions to Modules..... 290

Controlling the Generation and Use of Local Copies (Access Functions)..... 292

Associating Access Functions with an API..... 295

How to Create an Access Function Specification

Objective

To show the workflow used to create an access function (AF) specification.

Restriction

Note

Do not specify DD AccessFunctionTemplate objects that contain read/write AFs and variable AFs that are applicable to the same variable. Further, do not specify DD AccessFunctionTemplate objects that contain at least one value copy AF and any other AF that is applicable to the same variable, i.e. if you specify a DD AccessFunction object with AccessFunctionKind set to `READ_VALUE_COPY` or `WRITE_VALUE_COPY`, then this is the only AccessFunction of the AccessFunctionTemplate that may set the respective VariableKindSpec properties.

Method

To create an access function specification

1. Create a DD VariableClass object.
2. Reference the DD VariableClass object created in step 1 at the DD Variable object(s) for which you want to use the access function.
3. Create a DD AccessFunctionTemplate object.
4. Reference the DD AccessFunctionTemplate object created in step 3 at the DD VariableClass object created in step 1.
5. Add a new DD AccessFunction object to the DD AccessFunctionTemplate object's Settings subtree.

Result

You created the required DD objects for an access function specification.

Next step

You now have to specify the properties of the DD AccessFunction object as required by your use case. Refer to Determining Access Function Kind According to the Use Case on page 276.

Related topics

Basics

Associating Access Functions With Variable Kinds..... 282
Basics on Access Functions..... 270
Changing Access Function Implementations Using Name Macros..... 286
Controlling the Generation and Use of Local Copies (Access Functions)..... 292
Determining Access Function Kind According to the Use Case..... 276
Mapping Access Functions to Modules..... 290
Specifying Access Functions as C Functions..... 287
Specifying Access Functions as Preprocessor Macros..... 289

Determining Access Function Kind According to the Use Case

Access function kind

TargetLink provides read/write access functions and variable access functions. For both kinds of access functions, you have to set a DD AccessFunction object's AccessFunctionKind, PassVariable, and VariableKindSpec properties according your use case. The following tables show you how to set them, how to implement them, and provide a code example.

Tip

The dsdd_master_advanced DD template contains several predefined access function templates that you can copy and adapt to your requirements. Refer to Overview of Predefined Access Function Templates.

Reading the value of a scalar variable

AccessFunctionKind	PassVariable	Possible VariableKindSpec	Possible Implementation		Code Example
			C Function	Preprocessor Macro	
READ_SINGLE_VALUE	Implicit	APPLY_TO_SCALAR	✓	✓	`auxScalar = Read_scalar();`
	AsAddressParameter				`auxScalar = Read_scalar(&scalar);`

Reading the value of elements of a vector variable

AccessFunctionKind	PassVariable	Possible VariableKindSpec	Possible Implementation		Code Example
			C Function	Preprocessor Macro	
READ_SINGLE_VALUE	Implicit	APPLY_TO_VECTOR	✓	✓	`auxVector[5] = Read_vector(5);`
	AsAddressParameter				`auxVector[5] = Read_vector(vector, 5);`

Reading the value of elements of a matrix variable

AccessFunctionKind	PassVariable	Possible VariableKindSpec	Possible Implementation		Code Example
			C Function	Preprocessor Macro	
READ_SINGLE_VALUE	Implicit	APPLY_TO_MATRIX	✓	✓	`auxMatrix[2][3] = Read_Matrix(2, 3);`
	AsAddressParameter				`auxMatrix[2][3] = Read_Matrix(Matrix, 2, 3);`

Reading an entire scalar variable

AccessFunctionKind	PassVariable	Possible VariableKindSpec	Possible Implementation		Code Example
			C Function	Preprocessor Macro	
READ_VALUE_COPY	Implicit	APPLY_TO_SCALAR	✓	-	`Read_scalar(&auxScalar);`

Reading an entire vector variable

AccessFunctionKind	PassVariable	Possible VariableKindSpec	Possible Implementation		Code Example
			C Function	Preprocessor Macro	
READ_VALUE_COPY	Implicit	APPLY_TO_VECTOR	✓	-	`Read_vector(auxVector);`

Reading an entire matrix variable

AccessFunctionKind	PassVariable	Possible VariableKindSpec	Possible Implementation		Code Example
			C Function	Preprocessor Macro	
READ_VALUE_COPY	Implicit	APPLY_TO_MATRIX	✓	-	`Read_Matrix(auxMatrix);`

Reading an entire structured variable

AccessFunctionKind	PassVariable	Possible VariableKindSpec	Possible Implementation		Code Example
			C Function	Preprocessor Macro	
READ_VALUE_COPY	Implicit	APPLY_TO_STRUCT	✓	-	`Read_Struct(&auxStruct);`

Writing the value of a scalar variable

AccessFunctionKind	PassVariable	Possible VariableKindSpec	Possible Implementation		Code Example
			C Function	Preprocessor Macro	
WRITE_SINGLE_VALUE	Implicit	APPLY_TO_SCALAR	✓	✓	`Write_scalar(auxScalar);`
	AsAddressParameter				`Write_scalar(&scalar, auxScalar);`

Writing the values of elements of a vector variable

AccessFunctionKind	PassVariable	Possible VariableKindSpec	Possible Implementation		Code Example
			C Function	Preprocessor Macro	
WRITE_SINGLE_VALUE	Implicit	APPLY_TO_VECTOR	✓	✓	`Write_vector(5, auxVector[5]);`
	AsAddressParameter				`Write_vector(vector, 5, auxVector[5]);`

Writing the values of elements of a matrix variable

AccessFunctionKind	PassVariable	Possible VariableKindSpec	Possible Implementation		Code Example
			C Function	Preprocessor Macro	
WRITE_SINGLE_VALUE	Implicit	APPLY_TO_MATRIX	✓	✓	`Write_Matrix(2, 3, auxMatrix[2][3]);`
	AsAddressParameter				`Write_Matrix(Matrix, 2, 3, auxMatrix[2][3]);`

Writing an entire scalar variable

AccessFunctionKind	PassVariable	Possible VariableKindSpec	Possible Implementation		Code Example
			C Function	Preprocessor Macro	
WRITE_VALUE_COPY	Implicit	APPLY_TO_SCALAR	✓	-	`Write_scalar(&auxScalar);`

Writing an entire vector variable

AccessFunctionKind	PassVariable	Possible VariableKindSpec	Possible Implementation		Code Example
			C Function	Preprocessor Macro	
WRITE_VALUE_COPY	Implicit	APPLY_TO_VECTOR	✓	-	`Write_vector(auxVector);`

Writing an entire matrix variable

AccessFunctionKind	PassVariable	Possible VariableKindSpec	Possible Implementation		Code Example
			C Function	Preprocessor Macro	
WRITE_VALUE_COPY	Implicit	APPLY_TO_MATRIX	✓	-	`Write_Matrix(auxMatrix);`

Writing an entire structured variable

AccessFunctionKind	PassVariable	Possible VariableKindSpec	Possible Implementation		Code Example
			C Function	Preprocessor Macro	
WRITE_VALUE_COPY	Implicit	APPLY_TO_STRUCT	✓	-	`Write_Struct(&auxStruct);`

Returning the address of a scalar variable

AccessFunctionKind	PassVariable	Possible VariableKindSpec	Possible Implementation		Code Example
			C Function	Preprocessor Macro	
ADDRESS	Implicit	APPLY_TO_SCALAR	✓	✓	`auxScalar = *AddressOf_scalar();`
	AsAddressParameter				`auxScalar = *AddressOf_scalar(&scalar);`

Returning the address of a vector variable

AccessFunctionKind	PassVariable	Possible VariableKindSpec	Possible Implementation		Code Example
			C Function	Preprocessor Macro	
ADDRESS	Implicit	APPLY_TO_VECTOR	✓	✓	`auxVector[5] = AddressOf_vector()[5];`
	AsAddressParameter				`auxVector[5] = AddressOf_vector(vector)[5];`

Returning the address of a matrix variable

AccessFunctionKind	PassVariable	Possible VariableKindSpec	Possible Implementation		Code Example
			C Function	Preprocessor Macro	
ADDRESS	Implicit	APPLY_TO_MATRIX	✓	✓	`auxMatrix[2][3] = AddressOf_Matrix()[2][3];`
	AsAddressParameter				`auxMatrix[2][3] = AddressOf_Matrix(Matrix)[2][3];`

Returning the address of a structured variable

AccessFunctionKind	PassVariable	Possible VariableKindSpec	Possible Implementation		Code Example
			C Function	Preprocessor Macro	
ADDRESS	Implicit	APPLY_TO_STRUCT	–	✓	`AddressOf_MyStruct()->comp1 = …`
	AsAddressParameter		✓	✓	`AddressOf_MyStruct(&MyStruct)->comp1 = …`

Directly accessing a scalar variable

AccessFunctionKind	PassVariable	Possible VariableKindSpec	Possible Implementation		Code Example
			C Function	Preprocessor Macro	
DIRECT	Implicit	APPLY_TO_SCALAR	–	✓	`f(&Access_scalar());`
	AsAddressParameter				`f(&Access_scalar(scalar));`

Directly accessing a vector variable

AccessFunctionKind	PassVariable	Possible VariableKindSpec	Possible Implementation		Code Example
			C Function	Preprocessor Macro	
DIRECT	Implicit	APPLY_TO_VECTOR	–	✓	`auxVector[5] = Access_vector()[5];`
	AsAddressParameter				`auxVector[5] = Access_vector(vector)[5];`

Directly accessing a matrix variable

AccessFunctionKind	PassVariable	Possible VariableKindSpec	Possible Implementation		Code Example
			C Function	Preprocessor Macro	
DIRECT	Implicit	APPLY_TO_MATRIX	–	✓	`auxMatrix[2][3] = Access_Matrix()[2][3];`
	AsAddressParameter				`auxMatrix[2][3] = Access_Matrix(Matrix)[2][3];`

Directly accessing a structured variable

AccessFunctionKind	PassVariable	Possible VariableKindSpec	Possible Implementation		Code Example
			C Function	Preprocessor Macro	
DIRECT	Implicit	APPLY_TO_STRUCT	–	✓	`f(&Access_MyStruct());`
	AsAddressParameter		–		`f(&Access_MyStruct(MyStruct));`

Accessing 2-D arrays in AUTOSAR code

AccessFunctionKind	PassVariable	Possible VariableKindSpec	Possible Implementation		Code Example
			C Function	Preprocessor Macro	
FIRST_ELEMENT_ADDRESS	Implicit	APPLY_TO_MATRIX	✓	–	`T (* pMatrix)[4] = (T (*)[4]) AddressOf_Matrix(); auxMatrix[2][3] = pMatrix[2][3];`

Related topics

Basics

Associating Access Functions With Variable Kinds 282
Basics on Access Functions 270
Specifying Access Functions as C Functions 287
Specifying Access Functions as Preprocessor Macros 289

HowTos

How to Create an Access Function Specification 275

Associating Access Functions With Variable Kinds

Specifying variable kinds

By default, TargetLink associates access functions (AFs) with scalar or vector variables. You can configure other behavior via the DD AccessFunction object's VariableKindSpec property.

The DD **VariableKindSpec** property allows for a combination of the following values:

Value	Description
`APPLY_TO_SCALAR`	Applies the access function to scalar variables.
`APPLY_TO_VECTOR`	Applies the access function to vector variables.
`APPLY_TO_STRUCT`	Applies the access function to structure variables.
`APPLY_TO_MATRIX`	Applies the access function to matrix variables.

This way, you can specify separate or common access functions for different variable kinds within one access function template (and therefore via one DD **VariableClass** object).

Read/write access functions for vectors, matrices, and structs

The read/write access functions provided by TargetLink support two different kinds of access:

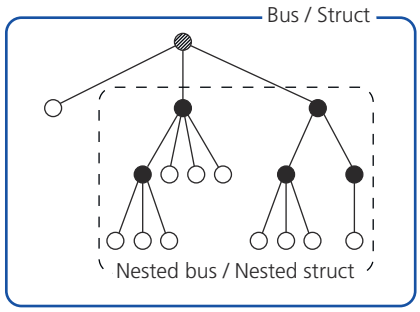
Kind of Access	Supported Variable Kinds				Description
	Scalar	Vector	Matrix	Struct	
Accessing single values element wise	✓	✓ [1]	✓ [1]	- [2]	Read access functions return a value that must be a scalar value according to C semantics. For this reason and also for symmetry, read and write access functions for vector/matrix variables work for one vector/matrix element at a time and require an index parameter. This index parameter has the same data type as a loop variable used to iterate across the entire vector/matrix. You can specify to generate `for` loops for vectors/matrices via the **LoopUnrollThreshold** Code Generator option. Refer to Basics on Processing Vectors and Matrices (📖 TargetLink Preparation and Simulation Guide).
Accessing an entire variable value	✓	✓ [3]	✓ [3]	✓ [3]	In TargetLink, value copy AFs are to be used only at the interface of code generation units (CGUs). When accessing the interface of a CGU from within, value copy AFs with `READ_VALUE_COPY` are to be used at inports or parameters and `WRITE_VALUE_COPY` at outports. Refer to Encapsulating the interface to external code (TargetLink production code as client) on page 269. When accessing the interface of a CGU from the outside, value copy AFs with `WRITE_VALUE_COPY` are to be used at inports and `READ_VALUE_COPY` at outports. Refer to Encapsulating the production code's interface (external code as client) on page 269.

[1] Requires one or more indices.
[2] According to C semantics.
[3] Requires a auxiliary variable as a buffer, potentially having static storage duration. This variable is passed by reference to the access function.

Propagating into structures

You can use the PropagateStructToComponents property to control whether TargetLink propagates access function templates into struct variables:

Property	Value	Description
PropagateStructToComponents	`<value not set>/on`	TargetLink propagates the access function template into struct variables.
	`off`	TargetLink does not propagate the access function template.



Root bus — Root struct
Bus element — Struct component
Leaf bus element — Leaf struct component

Propagation rules TargetLink propagates access function templates to a struct component only if all of the following conditions are true for that component:

- The component does not have an access function template specification (via its referenced variable class).
- The component's variable kind is either struct or matches the VariableKindSpec of at least one access function of the access function template.

For structured components, propagation continues recursively.

Note

The PropagateStructToComponents property of a DD AccessFunctionTemplate object is ignored if it contains a value copy AF whose VariableKindSpec property is set to `APPLY_TO_STRUCT`. This is the case because the access to the struct and all of its components is accomplished via an auxiliary buffer variable.

Use cases

You can associate multiple access functions belonging to *one* access function template (and therefore to one DD VariableClass object) to scalar, vector, matrix, and structure variables.

For example, the following use cases are possible:

Different macro bodies, one access function template: With TargetLink, you can provide *one* access function template containing different macro AFs for each variable kind.

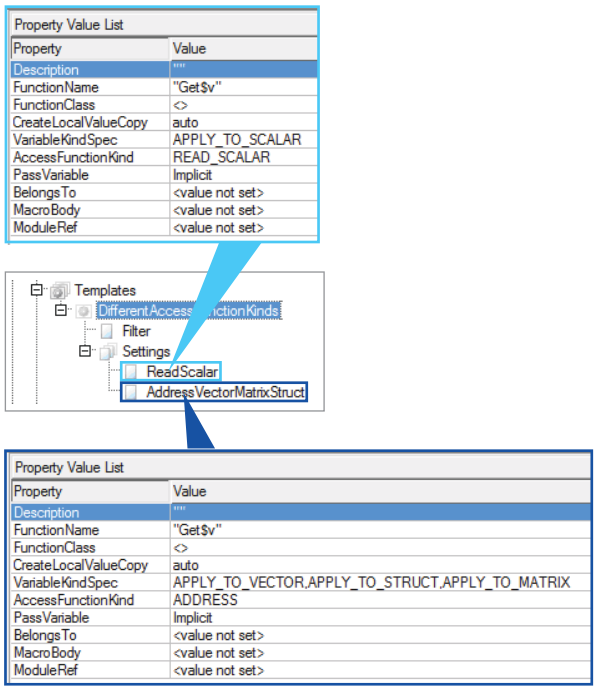
Create one DD AccessFunction object for each variable kind and specify its MacroBody and VariableKindSpec properties as required.

Different access function kinds, one access function template: With TargetLink, you can provide *one* access function template containing different access function kinds for each variable kind.

Create one DD AccessFunction object for each variable kind and specify its VariableKindSpec properties as required.

For example, you can specify one DD AccessFunctionTemplate object containing the following DD AccessFunction objects:

- One DD AccessFunction object with its AccessFunctionKind property set to `READ_SINGLE_VALUE` and its VariableKindSpec property to `APPLY_TO_SCALAR`
- One DD AccessFunction object with its AccessFunctionKind property set to `ADDRESS` and its VariableKindSpec property set to `APPLY_TO_STRUCT`, `APPLY_TO_MATRIX`, and `APPLY_TO_VECTOR`.



Related topics

Basics

Basics on Access Functions..... 270
Changing Access Function Implementations Using Name Macros..... 286
Determining Access Function Kind According to the Use Case..... 276
Specifying Access Functions as C Functions..... 287
Specifying Access Functions as Preprocessor Macros..... 289

References

LoopUnrollThreshold (TargetLink Model Element Reference)

Changing Access Function Implementations Using Name Macros

Influencing implementation via name macros

You can use name macros as (parts of) values of the DD AccessFunction object's FunctionName and MacroBody properties. This lets you specify the access function's identifier, and the access function's implementation for macro AFs.

Name macros for access functions

The following name macros are allowed as (parts of) values of the DD AccessFunction object's FunctionName and MacroBody properties:

Name Macro	Description
Common Name Macros	
`$v` `$V`	The variable name as is (case-sensitive) The variable name in capital letters
`$R`[1)]	Used to enable automatic name expansion to resolve name ambiguities. Has no effect if there is no name conflict with another function or macro.
`$T`	The shortcut of the base type specified in the TargetLink Data Dictionary at the DD Typedef object
`$(Type)`	▪ Name of the user-defined typedef, if the IsBaseType property is set to `off` ▪ Name of the base type, if the IsBaseType property is set to `on`.
`$(BaseType)`	Name of the base types specified in the TargetLink Data Dictionary.
`$(Module)`	The name of the module in which the object is defined.
`$(Dim1Width)`, `$(Dim2Width)`	▪ `$(Dim1Width)` - Replaced by the number of elements in the first dimension (row of a 2-D variable). ▪ `$(Dim2Width)` - Replaced by the number of elements in the second dimension (column of a 2-D variable). No effect for scalar variables. For vector variables, only `$(Dim1Width)` is applicable.
Access Function Specific Name Macros	
`$(Sv)`	Variable name of the root structure variable
`$(SV)`	Variable name of the root structure variable in capital letters
`$(SType)`	The value of $(Type) for the root structure variable.

Name Macro	Description
`$(SPath)`	The path from the root structure to the component, excluding root structure and component ▪ If $(SPath) is used in a macro body, the substructures are separated by dots. ▪ If $(SPath) is used in a C identifier, the substructures are separated by underscores.
`$(VarAccess)`[2]	This macro is allowed in access function templates to specify macro bodies and gives correct access to the relevant variable, as and when required: For non-struct component variables, it is replaced in the same way as `$v`, for struct component variables, it is replaced in the same way as `$(Sv)$(SPath).$v`.

[1] Only for the FunctionName value.
[2] Only for the MacroBody value.

Access function templates shipped with TargetLink

TargetLink provides ready-to-use access function templates that illustrate how to use name macros. Refer to Predefined Access Function Templates.

Related topics

Basics

Basics on Access Functions..... 270
Basics on Using Name Macros..... 78
Specifying Access Functions as C Functions..... 287
Specifying Access Functions as Preprocessor Macros..... 289

Specifying Access Functions as C Functions

Access functions as C functions

You can instruct TargetLink to implement an access function (AF) as a C function as follows:

- Provide a DD FunctionClass object whose Macro property is set to `off`.
- Reference the DD FunctionClass object via the DD AccessFunction object's FunctionClass property.

Note

If the access function is implemented as a C function, TargetLink does not evaluate the DD AccessFunction object's MacroBody property.

Formal parameters for function AF definitions

TargetLink uses the following formal parameters in its function AF definitions. Which parameters are used depends on the access function's kind. If more than one parameter is generated, their relative order is as follows:

Parameter	Formal Parameter
Variable	varptr (pointer)
Index of vector elements	index

Parameter	Formal Parameter
Row index of matrix elements	index1
Column index of matrix elements	index2
Value for write access	value
Value pointer for scalar and struct value copy	valueptr (pointer)
Value buffer for vector and matrix value copy	valuebuf[1)]

[1)] For matrix AFs whose BelongsTo property is set to `RTE_API` or `customRTE_API`, TargetLink passes a pointer to the first element (valueptr).

Example An access function has the following property settings:

Property	Value
AccessFunctionKind	`WRITE_SINGLE_VALUE`
PassVariable	`AsAddressParameter`
VariableKindSpec	`APPLY_TO_MATRIX`

When used for setting the value of a 2-D variable, its function definition looks like this:

```
void SetMyMatrix(Int16 (*varptr)[5], Int32 index1, Int32 index2, Int16 value)
{
   varptr[index1][index2] = value;
}
```

Customizing the function body

You can provide an external function to use instead of TargetLink's default function AF's function body. Refer to External interfaces on page 292.

Related topics

Basics

Basics on Access Functions..... 270
Changing Access Function Implementations Using Name Macros..... 286
Specifying Access Functions as Preprocessor Macros..... 289

HowTos

How to Create an Access Function Specification..... 275

Specifying Access Functions as Preprocessor Macros

Access functions as macros

You can instruct TargetLink to implement an access function (AF) as a macro as follows:

- Provide a DD FunctionClass object whose Macro property is set to on.
- Reference the DD FunctionClass object via the DD AccessFunction object's FunctionClass property.

The access function kind determines whether the call of the macro AF contains arguments.

> **Note**
>
> You cannot use macro AFs to access variables of static storage duration (module-local variables), because the accessed variable in the macro body is not visible in other modules according to C semantics.

Macro definition

TargetLink generates the definitions of macro AFs according to the following schema:

Schema for Macro Access Function Definitions

```
#define <MacroIdentifier>[(<MacroArgument(s)>)] <ReplacementList>
```

This schema consists of the following elements:

Element	Description
<MacroIdentifier>	Derived from the expanded value of the DD AccessFunction object's FunctionName property.[1]
<MacroArgument(s)>	Derived from the value of the DD AccessFunction object's AccessFunctionKind property. A combination of the following macro arguments is possible, depending on the access function kind: ▪ `_var` - Variable ▪ `_value` - Value for write access ▪ `_index` - Index of vector elements ▪ `_index1` - Row index of matrix elements ▪ `_index2` - Column index of matrix elements
<MacroBody>	Derived from the expanded value of the DD AccessFunction object's MacroBody property.[1] If the MacroBody property is not set, TargetLink uses a default macro body, depending on the values of the DD AccessFunction object's AccessFunctionKind and VariableKindSpec properties.

[1] The value has to be expanded because it can contain name macros.

Example An access function has the following property settings:

Property	Value
AccessFunctionKind	`WRITE_SINGLE_VALUE`
PassVariable	`Implicit`
VariableKindSpec	`APPLY_TO_MATRIX`

When used for setting the value of a 2-D variable, its macro definition looks like this:

```
#define MyMatrixIndexed(_index1, _index2, _value)\
   (MyMatrix[_index1][_index2] = (_value))
```

In the C file, the macro call looks like this:

```
MyMatrixIndexed(Aux_S32, Aux_S32_a, MyVector[Aux_S32] * MyVector[Aux_S32_a]);
```

Customizing the macro body

When implementing an access function as a macro, TargetLink generates a suitable macro body. You can customize the macro body via the DD AccessFunction object's MacroBody property.

The value of the MacroBody property can contain name macros.

Access function templates shipped with TargetLink

TargetLink provides ready-to-use access function templates that illustrate how to use name macros. Refer to Predefined Access Function Templates.

Examples of use cases

Because they do not cause function call overhead, macro AFs are useful for the following:

- Wrapping calls to existing user-defined access functions with a different parameter order or with additional parameters
- Adding side effects for logging or debugging
- Hiding the implementation of the variable from TargetLink

For an overview of use cases that are not restricted to macro AFs, refer to Overview of Typical Use Cases for Access Functions on page 268.

Related topics

Basics

Basics on Access Functions 270
Changing Access Function Implementations Using Name Macros 286
Overview of Typical Use Cases for Access Functions 268
Specifying Access Functions as C Functions 287

HowTos

How to Create an Access Function Specification 275

Mapping Access Functions to Modules

Implementation module and module ownership

You can make the Code Generator generate variables and access functions (AFs) into different modules. The implementation of an access function can be

generated either in its own implementation module or in the module of the variable that is accessed by the access function.

Note

If an access function is used to access several different variables, i.e., if variables from arbitrary modules are passed to the access function by parameter, then it is recommended to specify one module for all access functions of the access function template (AFT) in order to avoid problems with cyclic includes due to the way TargetLink would otherwise determine the module.

However, if you create different modules for variables and access functions, the owners of the modules might or might not be different code generation units (CGUs). It is beneficial to specify a module and a module owner for interface variables or parameter variables, as this ensures that each variable and its access functions are generated only once.

Keep in mind that production code for the access function module is generated only during code generation for the owner subsystem or code generation unit, but stub code versions are generated for the foreign code generation units. For details, refer to Mapping Code Generation Units to Code (Module Ownership) on page 382.

Note

If you specified a module reference at a DD **AccessFunction** or a DD **FunctionClass** object, you must also specify a module reference at the associated DD **Variable** and/or DD **VariableClass** object. This ensures that the variables are always generated into the same module.
If you do not have any special requirements for the access function and variable locations, reference the *same* module to get clear module dependencies.

Specifying the implementation module

You can control where TargetLink generates the implementation of an access function. That is, you can specify the access function's implementation module.

You can specify the implementation module in two ways:

- Via the DD **AccessFunction** object's **ModuleRef** property
- Via the DD **FunctionClass** object referenced at the DD **AccessFunction** object's **FunctionClass** property

If you do not specify a module, TargetLink uses the implementation module of the variable (for the implementation) or, if that implementation module of the variable does not belong to TargetLink, the TargetLink root file.

The owner of the access functions is the code generation unit that generates them, unless otherwise specified. Because access functions usually describe interfaces, directly specifying the ownership for the respective module helps to prevent interfaces that are unstable due to different definitions or file contents

for different code generation units (refer to Mapping Code Generation Units to Code (Module Ownership)).

If an access function is defined outside of the generated code, then using a DD **Module** object with the **Responsibility** property set to `External` allows the following:

- Grouping of the access function definitions
- Compile time checks whether the called access functions exist and are compatible

  This is possible because an include is used instead of forward declarations.

External interfaces

External interfaces represented by access functions can be specified in a similar way to external global variables.

It is best to create a user-specified DD **Module** object, set its **Responsibility** property to `External` and reference it at the DD **AccessFunction** objects used for that interface.

You can also specify the interface at the DD **FunctionClass** object referenced at the DD **AccessFunction** object. This can be done by setting the **Storage** property to `extern` or by setting the **Alias** property to `on`.

Note

When specifying external interfaces represented by access functions via the DD **FunctionClass** object referenced at a DD **AccessFunction** object, you have to provide an **Addfile** block in each subsystem accessing that interface.

Related topics

Basics

Basics on Access Functions..... 270
Basics on Modules and Module Ownership..... 388
Mapping Code Generation Units to Code (Module Ownership)..... 382

HowTos

How to Create an Access Function Specification..... 275

Controlling the Generation and Use of Local Copies (Access Functions)

Local copies

When access functions (AFs) are used, it can be necessary/beneficial to introduce local copies (implicit variables) as a cache or buffer mechanism. It is

necessary, for example, when a whole vector is accessed but only element-wise read and write access functions are specified.

Example

```
int16 _tmpu_v[5];
for(AuxS32 = 0; AuxS32 < 5; Aux_S32 +1){
   _tmpu_v[AuxS32] = ReadV(Aux_S32);
}
f(_tmpu_v);
```

In other cases, it is beneficial to use local copies to reduce the number of access function calls.

Controlling local copies

DD AccessFunction objects have the CreateLocalValueCopy property. This property lets you control whether TargetLink generates local copies to reduce the number of access function calls by storing the results in them.

The following values are available:

Value	Description
`avoid`	TargetLink only uses implicit variables for value copies where they cannot be avoided. The number of access function calls is not reduced.
`auto` (default)	TargetLink tries to use implicit variables to optimize access patterns for variables of typical modeling situations. This does *not* apply to structure components accessed via an access function for a parent struct variable. TargetLink *never* uses implicit struct variables, and in this mode does not use implicit scalar or vector variables for struct components.
`always`	TargetLink uses implicit variables for every single access. For detected access patterns, the code looks like the code for `auto`, even for struct components. However, implicit variables for struct components are now also accessed via a parent struct variable's access function. If TargetLink does not detect any pattern, it uses implicit variables anyway. The resulting code may not be as efficient as without implicit variables even though the implicit variables are often optimized later.

Note

For access functions whose AccessFunctionKind property is set to `READ_VALUE_COPY` or `WRITE_VALUE_COPY`, TargetLink always generates an implicit buffer variable.

Note

If multiple DD AccessFunction objects are associated with the same variable kind, do not set the CreateLocalValueCopy property of some to `avoid` and that of others to `always`.

Read access functions and TargetLink subsystem inports

When you access TargetLink subsystem inports with read access functions, you might have to create value copies (buffering) to reduce the number of access function calls. This can be necessary if the call has side effects or beneficial if additional calls impact the execution time of the generated code.

Controlling the buffering You can either have TargetLink perform the buffering or model the buffering explicitly.

> **Note**
>
> If you want full control of buffering, it is best to set the DD AccessFunction object's CreateLocalValueCopy property to `avoid` and model the buffering explicitly, if required.

Buffering by TargetLink TargetLink buffers to reduce the number of function calls to one under the following conditions:

Access Function is Side-Effect-Free	Value of CreateLocalValueCopy	Buffering by TargetLink[1)]
-	`avoid`	No
Yes	`auto`	No
No	`auto`	Yes
-	`always`	Yes

1) By introducing an implicit global variable that can be used by all other functions in production code.

> **Note**
>
> For access functions whose AccessFunctionKind property is set to `READ_VALUE_COPY`, TargetLink only evaluates the CreateLocalValueCopy property if it is set to `always`.

Buffering explicitly modeled You can also model this buffering and thus influence the buffer variable by making it explicit:

Introduce a Rescaler block after the TargetLink InPort block. The output variable of the Rescaler block must have a variable class in which the `ERASABLE` value of the Optimization property is not set. You also have to set the CreateLocalValueCopy to `avoid`.

TargetLink default behavior

When the DD CreateLocalValueCopy property is set to `auto`, TargetLink introduces a buffer variable if the access function is assumed to have side effects. Side effects essentially mean that the access function call has the potential to modify the underlying (read) variable or additional variables. The duplication of calls for access functions with side effects is not necessarily critical, e.g., if there is only a built-in call counter.

There is no direct way to specify *call has critical side effect*. Instead, TargetLink derives a decision from the access function's DD FunctionClass object: If the

Optimization property is set to `SIDE_EFFECT_FREE`, then no buffer variable is introduced.

Note

This use of the `SIDE_EFFECT_FREE` optimization property contrasts with its use in function optimization, where it does not involve buffering and does not mean *free of all side effects*.
For the use of the `SIDE_EFFECT_FREE` optimization property in function optimization, refer to Basics on Optimizing Functions via Property Values on page 524.

Related topics

Basics

Basics on Optimizing Functions via Property Values..... 524

Associating Access Functions with an API

Introduction

TargetLink internally uses its access function template mechanism in the following use cases:

AUTOSAR code generation Generating RTE API functions for AUTOSAR code generation.

Real-time support Generating RCP API functions for real-time platform support.

Associating an API

You associate an access function with an API by setting the DD AccessFunction object's BelongsTo property.

AUTOSAR RTE API You associate an access function with the AUTOSAR RTE API by setting the BelongsTo property to `RTE_API`.

In AUTOSAR code generation mode, TargetLink generates calls to the RTE API functions.

RCP API for real-time support You associate an access function with the RCP API for real-time platform support by setting the BelongsTo property to `RCP_API`.

If the RealTimePlatformSupport Code Generator option is enabled, TargetLink generates additional calls to entry and exit functions that are required for simulations on real-time platforms such as SCALEXIO.

Related topics

Basics

Comprehensive Modeling of RTE Code Pattern (📖 TargetLink Classic AUTOSAR Modeling Guide)
Interoperating with ConfigurationDesk (📖 TargetLink Interoperation and Exchange Guide)

References

RealTimePlatformSupport (📖 TargetLink Model Element Reference)

Reusing C Functions and Variables for Identical Model Parts (Function Reuse)

Introduction

The multiple use (reuse) of a single C function or variable reduces the size of the program code. It also allows a programming style that speeds up the development process by reusing existing functionality and increases reliability by testing smaller units.

Where to go from here

Information in this section

Introduction to Function Reuse.....297
Making Model Parts Reusable.....306
Defining the Code Pattern of Reused Variables.....308
Specifying Different Initial Values for Instance-Specific Variables.....313
Managing Special Indirect Reuse Scenarios.....316

Introduction to Function Reuse

Introduction

To become familiar with the basics of modeling function reuse and the code patterns for function reuse. Furthermore, reusing variables of neighboring blocks is explained, which is based on function reuse.

Where to go from here

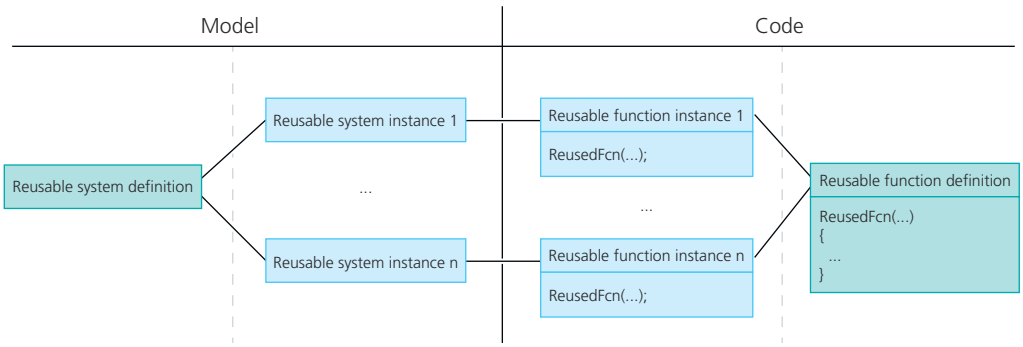
Information in this section

Basics on Function Reuse.....298
Basics on Reusing Variables of Preceding and Subsequent Blocks.....303

Basics on Function Reuse

Function reuse

Simulink® lets you reuse certain model parts. If a C function is assigned to a reused model part in the TargetLink model, this C function can be reused. The generated code therefore includes multiple function call instances.



Model-specific terms The reusable model part is called reusable system definition. If the reusable system definition is instantiated n times in the model, there are n reusable system instances in the model.

Code-specific terms If n reusable system instances reside in the model, then the generated code contains n reusable function instances. Each reusable function instance calls the one reusable function definition.

Function reuse helps facilliate the following:

- Smaller code size
- Better code readability

Demo model The following demo model introduces you to function reuse:

- FUNCTION_REUSE (TargetLink Demo Models)

Reusable model parts

The following model parts can become a reusable system definition:

- Library subsystems
- Referenced models
- Referenced subsystems

Refer to How to Turn Reuseable Model Parts into Reusable System Definitions on page 307.

Use Cases

Reusable Model Part	Use Case
Library subsystem	Reusing small model parts.
Referenced model	Reusing large model parts with long MIL simulations and long model initialization times.
Referenced subsystem	Reusing model parts across multiple projects.

Nested subsystems Subsystem levels inside a reusable system definition are automatically reused. If a nested subsystem is a reusable system definition, its reusable function instance is not integrated into the reusable function instance of the outer system and only the one reusable function definition is called.

Instance-specific versus shared variables

For all the variables used in a reusable system definition, you can decide whether to make them instance-specific variables or shared variables.

By default, TargetLink automatically determines whether a variable must become instance-specific or shared. However, you can influence this automatism.

Code patterns for instance-specific variables

Code Pattern		Specification	Instruction
Instance-specific variable in a reuse struct:	Direct reuse	Automatic or user-defined	How to Implement Instance-Specific Variables via Direct Reuse on page 308
	Indirect reuse	Only user-defined	How to Implement Instance-Specific Variables via Indirect Reuse on page 309
Instance-specific variable as function arguments		Automatic or user-defined	How to Implement Instance-Specific Variables as Function Arguments on page 311

Use cases for instance-specific and shared variables

Variable	Code Pattern	Use Case
Instance-specific	Direct reuse	▪ Reduce stack consumption if the function interface has inputs/outputs. ▪ Give a parameter or a state different values in each instance.
Instance-specific	Function argument	▪ Reduce the size of the reuse structs.
Instance-specific — or — Shared — or — Combination of both	Indirect reuse	▪ Include variables in the reuse struct but leave them at their original position, e.g., to generate (external) variables into a specific module. ▪ Combine instance-specific variables with shared variables, e.g., two instances use gain parameter A but a third instance uses gain parameter B.
Shared	One variable	▪ For inputs if re-entrance is not required. ▪ Have all instances use the same parameter, e.g., a calibratable gain parameter with the same initial value in all instances.

Tip

TargetLink can use variables of predecessor and successor blocks of a reusable system definition without generating additional interface variables. Refer to Basics on Reusing Variables of Preceding and Subsequent Blocks on page 303.

Naming indirectly reused variables

If you want to reuse variables via indirect reuse, you have to specify instance-specific variable names.

$E macro as name template You can easily specify instance-specific variable names by using the `$E` name macro. Depending on whether you define the name of the variable by using a mask parameter or a name mapping MATLAB script, the name template of the indirectly reused variable must read as follows in the reusable system definition:

Name is Defined via ...	Required Name Template
Mask parameters	`$E(<mask parameter name>)` e.g., `$E(NameTemplateKi)`
Name mapping MATLAB script	`$E(<file name>($THIS))` e.g., `$E(getInstanceNames($THIS))`

Refer to Example of Instance-Specific Variable Naming via MATLAB Scripts (Indirect Reuse) on page 317.

Implementation details for reuse structs

TargetLink always creates a main reuse struct for instance-specific variables during code generation. It also creates sub reuse structs if required.

Main reuse struct By default, the main reuse struct is implicitly created as follows:

Property	Value
InitAtDefinition[1)]	on
Const[1)]	off
TypePrefix[1)]	""
CodeDecorationSetRef[1), 2)]	""

1) You can change these properties via a DD StructTemplate object whose Filter.StructSpec property is set to `FcnReuse` and whose Settings.VariableClass property references a DD VariableClass object whose InitAtDefinition, Const, TypePrefix, and CodeDecorationSetRef properties are set as required.

2) Instead of specifying a DD StructTemplate you can also define a DD CodeDecorationSet via the DefaultDecorationSetRef property of the DD `/Config/TargetLink` object. TargetLink then uses this object for all implicit variables. Refer to Basics on Code Decorations on page 183.

Reused variables are generated into the main reuse struct if their properties match the properties of the main reuse struct shown in the preceding table.

The main reuse struct is passed to the reused functions as function parameter (pointer). The main reuse struct itself can contain pointers to sub reuse structs, and also pointers to the main reuse structs of other reused functions nested in the reused system.

To control how reused variables are assigned to reuse structs, refer to How to Change the Default Variable Class of Main Reuse Structs on page 316.

Sub reuse struct According to common C conventions, variables that reside in *different memory sections* cannot reside in the same struct. Therefore, TargetLink generates as many sub reuse structs as necessary to collect all the reused variables that do not belong in the main reuse struct. Each sub reuse struct collects all the reused variables whose following properties have the same value:

Property	Defined By
Const	Explicitly defined via the variable class of the variable.
InitAtDefinition	
TypePrefix	
WidthSpec	Implicitly defined by the data type of the variable.
DeclarationStatements	Explicitly defined via the code decoration that matches the variable.
SectionName	

You can use DD **SubStructTemplate** objects to change the variable class of each sub reuse struct. Define the filter of the object to match the properties shown in the preceding table.

Note

If your DD **SubStructTemplate** object does not reference a DD **VariableClass** object to be used for the sub reuse struct and the **ReuseStructHandling** Code Generator option is set to `1 = SeparateInherit`, TargetLink uses the the values of the following properties from the reused variables themselves for the sub reuse struct:

- TypePrefix
- DeclarationStatements
- SectionName

Function reuse as a process

Complete the following process steps when using the function reuse feature of TargetLink:

1. How to Turn Reuseable Model Parts into Reusable System Definitions on page 307
2. Defining the Code Pattern of Reused Variables on page 308
3. Instantiating the reusable system definition (1...n reusable system instances)
4. Specifying Different Initial Values for Instance-Specific Variables on page 313

Do the following when using a reusable system definition in multiple code generation units:

- Specify a unique module that holds the reusable function definition. Refer to How to Create Module Specifications in the Data Dictionary on page 391.
- If several code generation units exist, define one unit as the module owner of the reusable system definition. Refer to Basics on Modules and Module Ownership on page 388.

Current restrictions for function reuse

Function reuse has the following restrictions:

- External variables are supported only if they are referenced via pointers in the generated reuse struct. Refer to How to Implement Instance-Specific Variables via Indirect Reuse on page 309.
- Access functions are not supported for instance-specific variables.
- Logging settings of blocks in areusable system instance are ignored.
- Data variants are not supported.
- Components of DD structs are supported only if they are referenced via pointers in the generated reuse struct.

Limitation

Reused systems cannot be called via a function call from outside the TargetLink subsystem in which they reside.

For the sample times of reused and reusing systems, all the following conditions must be met:

- All sample times in a reused system instance must be identical.
- If at least one instance of a reused system is not part of another reused system instance, one of the following conditions must be fulfilled:
  - All the instances of the reused system have sample times that are *equal* to the sample time of their reusing system.
  - All the instances of the reused system have sample times that *differ* from the sample time of their reusing system.
- If a reused system contains a TargetLink **Discrete-Time Integrator** block, all its instances must have the same sample time.

Related topics

Basics

Basics on Code Decorations 183
Basics on Modules and Module Ownership 388
Basics on Reusing Variables of Preceding and Subsequent Blocks 303
Basics on Using Name Macros 78
FUNCTION_REUSE (TargetLink Demo Models)

HowTos

How to Create Module Specifications in the Data Dictionary 391
How to Implement Instance-Specific Variables as Function Arguments 311
How to Implement Instance-Specific Variables via Direct Reuse 308
How to Implement Instance-Specific Variables via Indirect Reuse 309
How to Turn Reuseable Model Parts into Reusable System Definitions 307

Examples

Example of Instance-Specific Variable Naming via MATLAB Scripts (Indirect Reuse) 317

References

ReuseStructHandling (TargetLink Model Element Reference)
StructTemplate (TargetLink Data Dictionary Reference)

Basics on Reusing Variables of Preceding and Subsequent Blocks

Reusing variables of predecessor and successor blocks

In a reusable system definition, TargetLink can use variables of predecessor and successor blocks of the reusable system definition without generating additional interface variables. If you want to make a variable of a predecessor or successor block instance-specific, you have to reference a DD variable class at either a TargetLink **InPort** block, a TargetLink **OutPort** block, a Stateflow Input or Output object, or at the boundaries of the reusable system definition by using the following property settings (explicit DD specifications):

Variable Class Property	Property Value	Results In
FunctionReuse	`instanceSpecific`	The variable is reused.
FunctionReuseAccessByPointer[1]	`on`	A const pointer to the variable will be a component of the reuse structures (indirect reuse).
Scope[2]	`ref_param`	Variable propagation (also implemented via pointers).

[1] Determines how to implement the integration in the reuse structures.
[2] Determines whether variable propagation or indirect reuse is applied.

Reusing variables of predecessor and successor blocks is useful for:

- Reducing the number of update assigns required before and after the call of reusable function instances in the code.
- Generating highly efficient production code, i.e., code that is optimized regarding run time, code size, stack, and RAM consumption.

  This especially applies to cascaded reuse, where signals are propagated from the outside to the inner reusable system definition.

Overruling the TargetLink default behavior With the above, explicit DD specifications, you force the Code Generator to implement either a pointer as a component in the reuse structure pointing to the instance-specific variable (? indirect reuse), or to implement variable propagation. Refer to Basics on Function Reuse on page 298.

If you set Scope to `ref_param`, TargetLink uses the variable from the predecessor block (for InPort), or from the successor block (for OutPort) also for the current inport or outport. This process is called variable propagation. Ports providing the required settings are called propagation ports and the variables being propagated are called propagation variables.

Propagation pointer If you specified variable propagation, TargetLink generates propagation pointers in the reuse structure of the innermost reusable system definitions in which the variable propagation ends. If variable propagation is specified at TargetLink InPorts, there can be multiple inner reusable system definitions (due to branches in the signal line). The generated pointers point to the propagation variables. Accesses in the reusable system definitions involved in the propagation are carried out only through these propagation pointers. The settings name, type, and the variable class at the first propagation port of the reusable system definition containing the propagation pointers are used for the propagation pointer.

Propagation scope In a reusable system definition, variable propagation is performed along the signal line or into nested reusable system definitions, if only propagation ports are on the signal line and these propagation ports can use the propagation variable of their predecessor or successor propagation port. This optimization aims to reduce the number of propagation pointers and to avoid copy and update actions. If the variable cannot be used for propagation, i.e., optimization is not possible because a precondition is not fulfilled, then variable propagation of the predecessor or successor propagation port ends and a new variable propagations starts at this port. For the variable that cannot be used for propagation, TargetLink creates a separate interface variable for the propagation port and writes the interface variable and the particular reason to a report.

Preconditions for propagation variables

For variable propagation, the variable of the immediately preceding or following block, including all calculating blocks, e.g., Gain blocks, routing blocks such as Simulink port blocks, etc. is used as the propagation variable if it complies with the following, most important preconditions (further preconditions are available in the report):

- The Type and Scaling (LSB, Offset) properties at the propagation variable and the propagation port must be identical.

- The Const property of the variable class referenced at the propagation inport is set to on. This is mandatory only if the propagation variable is const and is also only relevant for input propagation (outputs must never be Const). Setting the Const property is beneficial if there are non-constant and constant predecessor blocks in the instances.

Limitations for variable propagation

Variable propagation is not supported in the following cases:

- For matrix and bus signals
- Outside of reused systems
- At blocks other than TargetLink InPorts, TargetLink OutPorts, Stateflow Inputs/Outputs
- At outputs of subsystems that are enabled, triggered, function-call-triggered or action-port-triggered

Unsupported predecessor blocks For propagation inports, the following predecessor blocks are not supported:

- Bus Creator
- Bus Selector
- Demux
- Enhanced outport
- Matrix Concatenate
- Multiport Switch
- Mux
- Permute Dimensions
- Reshape
- Selector
- Switch
- Unenhanced inport/outport
- Vector Concatenate

Supported successor blocks For propagation outports, exactly one successor block is supported, i.e., there should not be any branches from the signal line. The supported blocks are:

- Data Store Write
- Merge
- OutPort
- Rescaler

Tip

Any errors and messages on mismatching port block signal specifications that occur during code generation are summarized in a report that helps you locate critical or inefficient code parts. Therefore, you are recommended to enable report generation during reuse propagation configuration. For details, refer to Basics on the Code Generation Report (📖 TargetLink Preparation and Simulation Guide).

Demo model

The following demo model introduces you to variable propagation:

- FUNCTION_REUSE (📖 TargetLink Demo Models)

Related topics

Basics

Basics on Function Reuse..... 298
Basics on the Code Generation Report (📖 TargetLink Preparation and Simulation Guide)
FUNCTION_REUSE (📖 TargetLink Demo Models)

HowTos

How to Customize the Code Generation Report (📖 TargetLink Preparation and Simulation Guide)

References

EnableReportGeneration (📖 TargetLink Model Element Reference)

Making Model Parts Reusable

Introduction

To make a library subsystem, a referenced model, or a referenced subsystem a reusable system definition which can then be instantiated in a model multiple times (= multiple reusable system instances). This is a mandatory preparation step for the implementation of instance-specific and shared variables.

How to Turn Reuseable Model Parts into Reusable System Definitions

Restrictions

Function–call–trigger inputs and outputs are not supported.

Preconditions

Reusable Model Part	**Simulink® Preconditions**	**TargetLink Preconditions**
Library subsystem	The library subsystem resides in a Simulink library and the library link is active. It is highly recommended that you enable the **Treat as atomic unit** Simulink option for the subsystem.	▪ The subsystem contains a TargetLink **Function** block at the top level. ▪ The subsystem inport and outport blocks must be enhanced to TargetLink **InPort** and **OutPort** blocks.
Referenced model	–	
Referenced subsystem	The referenced subsystem is specified as a subsystem reference in Simulink.	▪ The subsystem is TargetLink compliant. Refer to How to Make Referenced Subsystems TargetLink-Compliant (📖 TargetLink Preparation and Simulation Guide). ▪ The subsystem contains a TargetLink **Function** block at the top level. ▪ The subsystem inport and outport blocks must be enhanced to TargetLink **InPort** and **OutPort** blocks.

Method

To turn reusable model parts into reusable system definitions

1. Open the reusable model part.
2. In the dialog of the **Function** block, specify fixed function names.
   The following name macros are not supported:
   - $C
   - $R
   - $S
3. Specify a unique subsystem ID.
4. Select the **Make function reusable** checkbox.

Result

You turned a reusable model part into a reusable system definition.

Next steps

Refer to Basics on Function Reuse on page 298.

Related topics

Basics

Basics on Function Reuse..... 298

HowTos

How to Make Referenced Subsystems TargetLink-Compliant (📖 TargetLink Preparation and Simulation Guide)

Defining the Code Pattern of Reused Variables

Introduction

To explicitly choose one of the three code patterns that TargetLink offers for implementing instance-specific variables (ISVs). However, if a parameter does not need instance-specific variables, you can implement one shared variable, instead.

Where to go from here

Information in this section

How to Implement Instance-Specific Variables via Direct Reuse..... 308

How to Implement Instance-Specific Variables via Indirect Reuse..... 309

How to Implement Instance-Specific Variables as Function Arguments..... 311

How to Use Shared Variables..... 312

How to Implement Instance-Specific Variables via Direct Reuse

Objective

To implement an instance-specific variable via direct reuse.

Precondition

You created a reusable system definition. Refer to How to Turn Reuseable Model Parts into Reusable System Definitions on page 307.

Method

To implement instance-specific variables via direct reuse

1. Open the reusable system definition.
2. Assign a global variable class to the variables that you want to reuse directly.

3 Enter a modifiable, unique block output or parameter name, for example, **$S_$B**, in the relevant **Name** edit field.

Note

A name that contains **$S**, **$C**, or **$R** causes the instance-specific implementation of the variable. These name macros enable automatic name expansion to resolve name ambiguities and make it possible to place the variable in instance structures for function reuse. For more information on block output or parameter names, refer to Details on the Expansion of Name Macros on page 80. If you enter a fixed name instead, the same variable is shared by all instances of the reused subsystem.

4 Repeat steps 1 to 3 for all outputs and parameters concerned.

Result

You implemented an instance-specific variable via direct reuse.

Next step

Enter the instance-specific values. Refer to Specifying Different Initial Values for Instance-Specific Variables on page 313.

Related topics

Basics

Basics on Function Reuse..... 298
Details on the Expansion of Name Macros..... 80

HowTos

How to Turn Reuseable Model Parts into Reusable System Definitions..... 307

Examples

Example of Instance-Specific Variable Naming via MATLAB Scripts (Indirect Reuse)..... 317

How to Implement Instance-Specific Variables via Indirect Reuse

Objective

To implement an instance-specific variable via indirect reuse.

Preconditions

- You created a reusable system definition. Refer to How to Turn Reuseable Model Parts into Reusable System Definitions on page 307.

- You specified a DD **VariableClass** object as follows:

Property	Value
Scope	global
FunctionReuse	instanceSpecific
FunctionReuseAccessByPointer	on

Refer to How to Define New Variable Classes on page 93.

Method

To implement instance-specific variables via indirect reuse

1. Open the reusable system definition.
2. Assign the specified DD **VariableClass** object to the variables that you want to reuse indirectly.
3. If you want pointers to the same variable to be merged, set the **SameIndirectReusePointerForSameDestination** global code generation option to **on** in the **TargetLink Main Dialog** block.

   Note

   You need advanced knowledge of TargetLink code generation if you make use of the merge option.

4. Repeat steps 1 to 3 for all variables concerned.

Result

You implemented an instance-specific variable via indirect reuse.

Next step

Enter the instance-specific values. Refer to Specifying Different Initial Values for Instance-Specific Variables on page 313.

Related topics

HowTos

How to Define New Variable Classes..... 93
How to Turn Reuseable Model Parts into Reusable System Definitions..... 307

References

FunctionReuse (📖 TargetLink Data Dictionary Reference)
FunctionReuseAccessByPointer (📖 TargetLink Data Dictionary Reference)
SameIndirectReusePointerForSameDestination (📖 TargetLink Model Element Reference)
Scope (📖 TargetLink Data Dictionary Reference)

How to Implement Instance-Specific Variables as Function Arguments

Objective

To implement an instance-specific variable in the function interface to reduce either RAM or ROM consumption. No reuse struct is implemented.

Preconditions

You created a reusable system definition. Refer to How to Turn Reuseable Model Parts into Reusable System Definitions on page 307.

Method

To implement instance-specific variables as function arguments

1. Open the reusable system definition.
2. Specify a DD **VariableClass** object whose **Scope** property is set to either`fcn_param` or `ref_param` to the variables you want to reuse via function arguments.

   You can specify the variable classes and naming of the formal and the actual parameter separately. The name of the actual parameter is crucial for the implementation either as an instance-specific variable or shared variable. Refer to:

   - Defining the Function Interface on page 241
   - Example of Specifying Block Parameters as Input Arguments of Functions on page 114

   If you do not specify a name for the actual parameter, TargetLink automatically implements the parameter as an instance-specific variable.
3. Repeat steps 1 to 2 for all variables concerned.

Result

You implemented an instance-specific variable as a function argument.

Next step

Generally, you have to enter the instance-specific values. Refer to Defining the Code Pattern of Reused Variables on page 308.

Related topics

Basics

Basics on Function Reuse..... 298
Defining the Code Pattern of Reused Variables..... 308
Defining the Function Interface..... 241

HowTos

How to Turn Reuseable Model Parts into Reusable System Definitions..... 307

Examples

Example of Specifying Block Parameters as Input Arguments of Functions..... 114

How to Use Shared Variables

Objective

To use a shared variable for all reusable system instances if the parameter values are identical to make parameter values superfluous in the instance structure.

Preconditions

You created a reusable system definition. Refer to How to Turn Reuseable Model Parts into Reusable System Definitions on page 307.

Method

To use shared variables

1. Open the reusable system definition.
2. Set the parameter's Class to MERGEABLE_GLOBAL.
3. In the Name edit field, enter a fixed name.
   The following name macros are not supported:
   - $C
   - $R
   - $S
4. Repeat steps 1 to 3 for all parameters concerned.

Result

You created a shared variable for all instances. The shared variable itself is therefore not part of each instance. The Code Generator does not add the single code variable to the reuse structure.

Related topics

Basics

Basics on Function Reuse..... 298

HowTos

How to Turn Reuseable Model Parts into Reusable System Definitions..... 307

Specifying Different Initial Values for Instance-Specific Variables

Introduction

To specify different initial values for instance-specific variables (i.e., states and parameters).

Where to go from here

Information in this section

How to Specify Instance-Specific Values for Library Subsystems..... 313

How to Specify Instance-Specific Values for Referenced Models..... 315

How to Specify Instance-Specific Values for Library Subsystems

Objective

To specify instance specific parameter values or instance specific initial conditions of states for library subsystems.

Demo model The instructions are based on the FUNCTION_REUSE (TargetLink Demo Models) demo model.

Restrictions

You can use instance-specific values only for instance-specific variables.

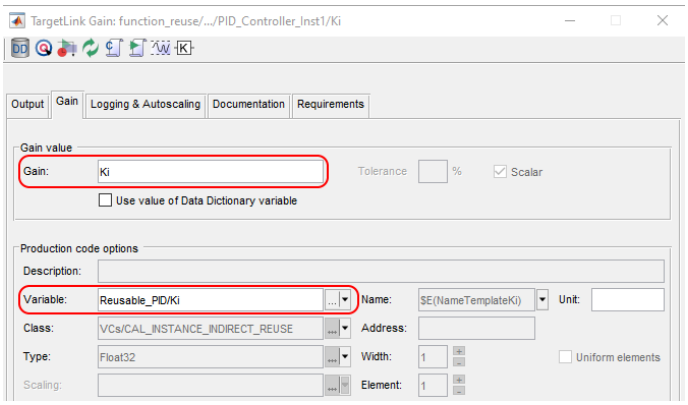
Preconditions

You created a reusable system definition. Refer to How to Turn Reuseable Model Parts into Reusable System Definitions on page 307.

Method

To specify instance-specific values for library subsystems

1. In the Simulink® Mask Editor for the reusable system instance, add mask parameters for each parameter with instance-specific values.
2. Open the block dialog for the parameter for which you want to specify instance-specific values.
3. For different initial values in different reusable system instances, specify the name of the mask parameter in the parameter field, e.g., `Ki`.
4. For indirect reuse of parameters, specify a DD Variable object whose NameTemplate property uses the $E name macro, e.g., `Reuseable_PID/Ki`.



5. Repeat step 3 and 4 for all parameters for which you want to specify instance-specific values.
6. In the mask of the reusable system instance, enter the instance-specific parameter values.

Result

You specified instance-specific parameter values for library subsystems.

Related topics

Basics

FUNCTION_REUSE (TargetLink Demo Models)

HowTos

How to Turn Reuseable Model Parts into Reusable System Definitions..... 307

How to Specify Instance-Specific Values for Referenced Models

Objective

To specify instance-specific parameter values for referenced models.

Demo model The instructions are based on the MULTIPLE_INSTANCES_REFMODEL (TargetLink Demo Models) demo model.

Restrictions

You can use instance-specific values only for instance-specific variables.

Preconditions

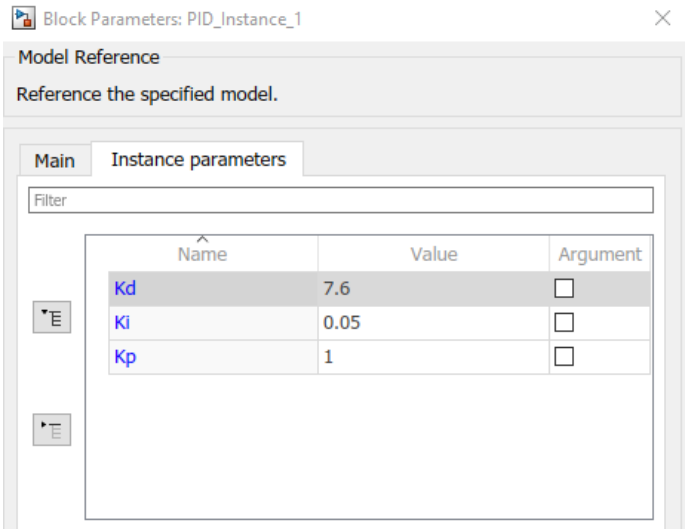
- You created a reusable system definition. Refer to How to Turn Reuseable Model Parts into Reusable System Definitions on page 307.
- The reusable system instance must reside in the parent model, which must be represented by a **Model** block.
- The referenced model contains **Model arguments** that are used as values for the instance-specific variables.

Method

To specify instance-specific values for referenced models

1 Open the Simulink® **Block Parameters (ModelReference)** dialog for the reusable system instance.

2 On the **Instance parameter** page, specify the instance-specific parameter values.



Result

You specified instance-specific parameter values for referenced models.

Related topics

Basics

MULTIPLE_INSTANCES_REFMODEL (📖 TargetLink Demo Models)

HowTos

How to Turn Reuseable Model Parts into Reusable System Definitions..... 307

Managing Special Indirect Reuse Scenarios

Where to go from here

Information in this section

How to Change the Default Variable Class of Main Reuse Structs..... 316

Example of Instance-Specific Variable Naming via MATLAB Scripts (Indirect Reuse)..... 317

How to Change the Default Variable Class of Main Reuse Structs

Objective

To move pointers to ISV from sub reuse structs into main reuse structs, which reduces memory size (ROM) and execution time.

Variable class of main reuse structs

Indirect reuse pointers that reside in a reuse struct always have their Const property set to `On`. By default, the constness of main reuse structs is set to `Off`, which does not match the constness setting of these pointers. Hence, these pointers are generated into a sub reuse struct.

Method

To change the default variable class of main reuse structs

1. In the Data Dictionary, create a DD StructTemplate object.
2. Select its Filter object and set its StructSpec property to `FcnReuse`.
3. Create a DD VariableClass object and set its Const property to `on`.
4. Reference the DD VariableClass object at the Settings.VariableClass property of the StructTemplate object.

Result

You changed the default constness setting of main reuse structs to on. Hence, the main reuse structs can now hold pointers, which shortens variable access: e.g., `pISV->pSubStruct->x` becomes `pISV->x`.

Related topics

Basics

Basics on Function Reuse..... 298
Basics on Using Templates..... 203
Details on Predefined DD Template Objects..... 209

HowTos

How to Create DD Template Objects..... 213

Example of Instance-Specific Variable Naming via MATLAB Scripts (Indirect Reuse)

Introduction

To provide instance-specific variable naming for a reusable system instance via MATLAB script if mask parameters are not or cannot be used, for example, in nested reused subsystem..

Demo model

The example is based on the FUNCTION_REUSE (TargetLink Demo Models) demo model. The demo model has to be modified as follows:

- A name mapping MATLAB script as described in the following resides on a MATLAB search path.
- The name template of the reused variable references the script via the `$E` name macro.

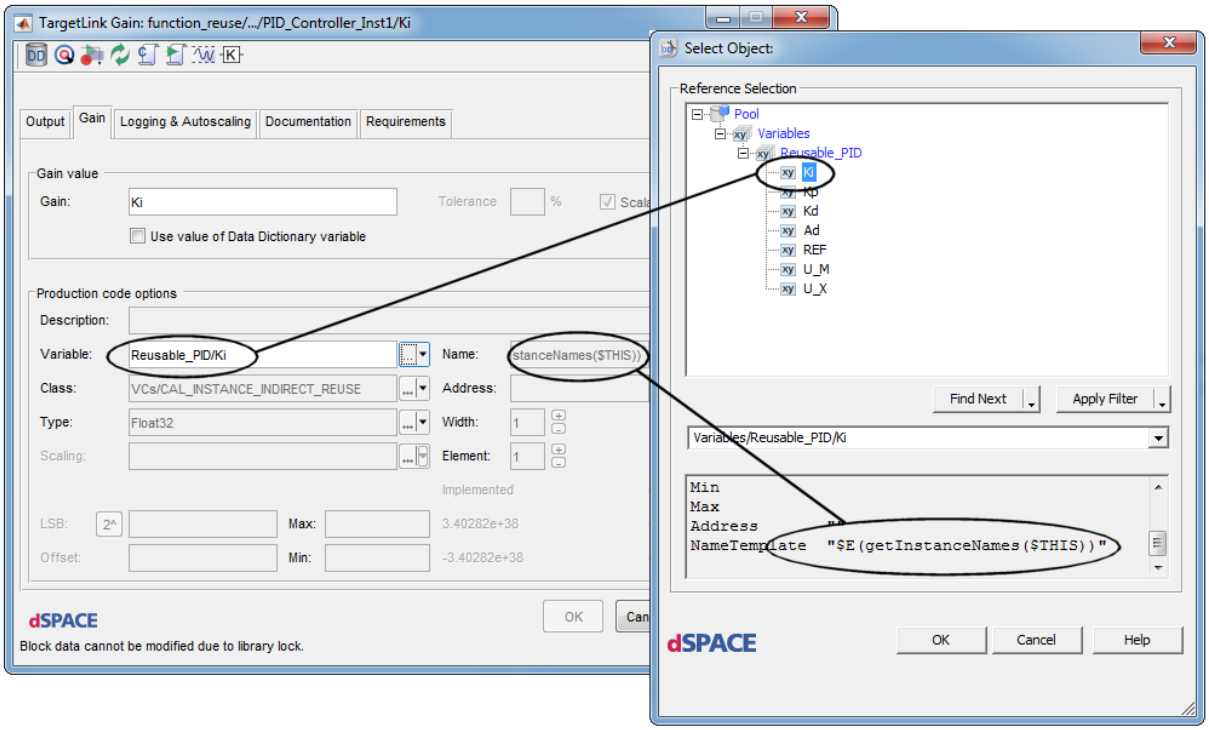
Refer to Applying a name mapping MATLAB script.

Applying a name mapping MATLAB script

You can name a variable with respect to its context by applying a user-defined name mapping MATLAB script. For the relevant Ki gain parameters, the variable name is specified by using one of the following:

- The `$E` macro
- A user-defined name mapping script

In the following example, the user-defined `getInstanceNames` name mapping MATLAB script assigns Ki1 (PID_Controller_Inst1 and PID_Controller_Inst2) and Ki2 (PID_Controller_Inst3) as variable names in the reusable system instances. Refer to Example script.



Note

The parameter name in the block dialog must match the file name of the name mapping MATLAB script. In this example, it is <name>.

$THIS The $THIS macro in `getInstanceNames($THIS)` is replaced by the handle of a block, Stateflow object, or ML Fcn object during code generation. Thus, the `getInstanceNames` MATLAB script can distinguish between different instances and blocks, and it returns the appropriate name of the variable, which is then evaluated by the $E macro.

Tip

To get the instance-specific block handle of a Stateflow chart or MATLAB function block, also use the `$CHART` macro. Refer to $E, $THIS, and $CHART on page 84.

Example script

The content of the user-defined `getInstanceNames.m` name mapping MATLAB script, which is an example script, might look like this:

```
function varName = getInstanceNames(THIS)
varName = 'unknown$R';

blockName = getfullname(THIS);

systemPrefix = 'function_reuse/pidcontroller/Subsystem/pidcontroller/';

if strcmp(blockName, [ systemPrefix 'PID_Controller_Inst1/Ki' ] )
    varName = 'Ki1';
elseif strcmp(blockName, [ systemPrefix 'PID_Controller_Inst2/Ki' ] )
    varName = 'Ki1';
elseif strcmp(blockName, [ systemPrefix 'PID_Controller_Inst3/Ki' ] )
    varName = 'Ki2';
end
```

Note

The script must reside on a MATLAB search path.

Result

The instance-specific names of the Ki gain parameter in the generated code (indirect reuse struct) read as follows:

Reusable System Instance	Name of Ki
PID_Controller_Inst1	Ki1
PID_Controller_Inst2	Ki1
PID_Controller_Inst3	Ki2

Related topics

Basics

Basics on Function Reuse 298

Implementing Scaling-Invariant Functions

Introduction

To use the same implementation of a function with differently scaled input signals but without scaling adjustment code at its boundaries.

Where to go from here

Information in this section

Basics on Scaling-Invariant Functions..... 320

Details on the Impact of Rescaling..... 322

Details on the Scaling Propagation Function..... 324

How to Specify Subsystems as Scaling-Invariant..... 329

How to Specify Stateflow Charts and Graphical Functions as Scaling-Invariant..... 331

Basics on Scaling-Invariant Functions

Implementing scaling-invariant functions

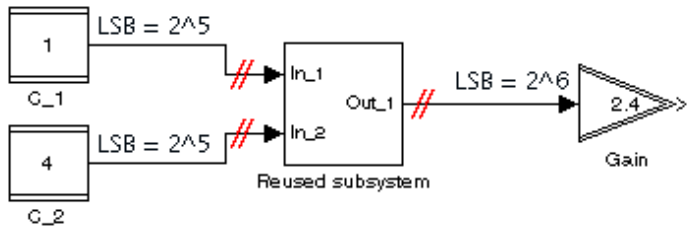
A function is a representation, for example, of a subsystem, a chart or a graphical function. The description below takes a scaling-invariant subsystem as an example, but also applies to graphical functions and charts. The function in the example takes scaled integer values as input and produces scaled integer values as output. There are also instructions on activating scaling invariance for charts and graphical functions.

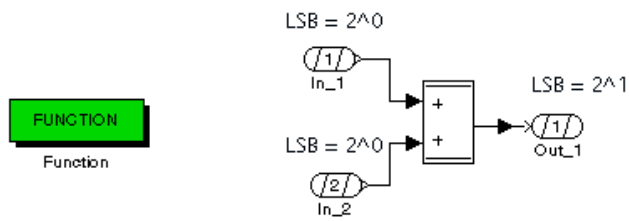
Scaling adjustment code

Sometimes, a function can be implemented in such a way that no scaling adjustment code is generated at its boundaries, although the LSB and offset values of its input/output signals differ from the LSB and offset values of the related **InPort/OutPort** blocks.

Example

Suppose that the signals that enter a subsystem have an LSB = 2^5. However, both inports of the subsystem have an LSB = 2^0. In addition, the LSB value of the calculated output signal differs from the LSB value specified for the subsequent **Gain** block. The offsets of all signals are assumed to be 0 in this example.





Thus, the LSB values of input/output signals and **InPort/OutPort** blocks are different. This difference would require scaling adjustment code at the boundaries of the subsystem. However, TargetLink can make use of the following circumstance:

The LSB value of the output signal can be derived from the LSB values of the subsystem's input signals. By applying a 1-bit left shift to one of the input LSB values (both inport LSB values are equal), you obtain the output LSB value. In other words, if TargetLink knows the scaling propagation function of the subsystem, it can evaluate the LSB and offset values of the input signals and calculate the LSB and offset values of the output signals. Thus, in this example, if the input signals have an LSB = 2^5 and the scaling transfer function is a 1-bit left shift, the LSB of the output signal is calculated as LSB = 2^6.

Benefit

If you want to reuse a subsystem and implement it as scaling-invariant, you get round the following tradeoff that a scaling-variant system involves:

- Implementing several subsystems which differ only in the LSB and offset values of their **InPort** and **OutPort** blocks
- Implementing only one subsystem, but having to rescale signals at the subsystem boundary, which means additional code. For details, refer to Details on the Impact of Rescaling on page 322.

The idea is to use the same implementation of a function with differently scaled input signals but without scaling adjustment code at its boundaries.

Note

You can implement scaling-invariant functions as reused functions. This gives you the advantage of implementing only a single function, but with no scaling adjustment code. Refer to Reusing C Functions and Variables for Identical Model Parts (Function Reuse) on page 297.

Limitation

For scaling-invariant Stateflow functions, only these actuals are allowed:

- Stateflow data objects (optionally providing index access)
- Calls of a Stateflow function
- Enumeration constants

Related topics

Basics

Defining the Function Interface 241
Details on the Impact of Rescaling 322
Reusing C Functions and Variables for Identical Model Parts (Function Reuse) 297

HowTos

How to Specify Stateflow Charts and Graphical Functions as Scaling-Invariant 331
How to Specify Subsystems as Scaling-Invariant 329

Details on the Impact of Rescaling

Rescaling definition

Rescaling means transforming a given integer value and its LSB and offset values into another integer value with other LSB and offset values, so that both integer values represent the same physical value.

Example

The following integer values represent the same physical value although they differ, which is due to the adapted, that is, rescaled LSB value:

Integer Value	**00010110**	**01011000**
Type	Int8	Int8
LSB	2^-2	2^-4
Offset	0	0
Physical Value	**5.5**	**5.5**

Rescaling is achieved by at least a bit-shift operation. Unlike power-of-two scaling, arbitrary scaling additionally requires multiplication and/or division. However, rescaling always causes additional code and can influence an implementation as follows:

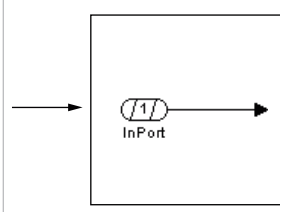
- Loss of lower bits (accuracy) if the LSB of the input signal is finer than the LSB of the **InPort** or Rescaler block.
- Loss of upper bits (range) if the LSB of the input signal is rougher than the LSB of the **InPort** or Rescaler block, which may necessitate a larger data type to avoid the loss of upper bits.

Example

Suppose a voltage signal is the input of a subsystem. If the related inport is scaling-variant, TargetLink rescales this signal as follows:

Use case 1: The LSB of the input signal (2^-2) is finer than the LSB of the InPort block (2^0):

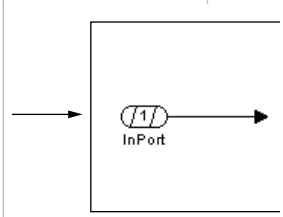
	Input Signal	Inport Block	Output Signal
Type:	UInt8	UInt8	UInt8
LSB:	2^-2	2^0	2^0
Offset:	0	0	0
Binary value:	00001101		00000011[01]
Physical value:	3.25 V		3 V



For the production code, a 2-bit right shift is implemented. Due to the shift operation, the two lower bits are truncated, which causes a loss of accuracy: 3.25 V becomes 3 V.

Use case 2: The LSB of the input signal (2^2) is rougher than the LSB of the InPort block (2^0):

	Input Signal	Inport Block	Output Signal
Type:	UInt8	UInt8	UInt8
LSB:	2^2	2^0	2^0
Offset:	0	0	0
Binary value:	01000001		[01]00000100
Physical value:	260 V		4 V



For the production code, a 2-bit left shift is implemented. Due to the shift operation, the two upper bits are truncated, which causes a loss of range: 260 V becomes 4 V. This would not happen if the data type was set to UInt16 instead of UInt8.

Related topics

Basics

Discrete-Time Integrator Block (TargetLink Preparation and Simulation Guide)
FIR Filter Block (TargetLink Preparation and Simulation Guide)
Integer Value Representation (TargetLink Preparation and Simulation Guide)

Details on the Scaling Propagation Function

Defining scaling propagation functions

A scaling propagation function belongs to a specific subsystem, chart, or graphical function. It relates the scalings of the outports to the scalings of the relevant inports. The relationship must be independent of the actual scaling parameters of incoming signals.

Specification of scaling propagation functions

A scaling propagation function is to be specified as an M script in the Function block that resides in the related subsystem. The M script defining the scaling propagation function accesses a structure, `SII_In(i)`, which is provided by the Code Generator for each inport `i` of the subsystem. This structure describes the properties of an inport `i` of a subsystem. It can be read in the M script and contains the following objects:

Note

TargetLink does not parse a scaling propagation function. You have to guarantee its validity.

Inport properties provided by the Code Generator

```
SII_In(i).Width
SII_In(i).Type
SII_In(i).Lsb
SII_In(i).Offset
SII_In(i).Arbitrary
SII_In(i).Min
SII_In(i).Max
SII_In(i).IsDataValid
```

Tip

The `IsDataValid` flag indicates if an inport's scaling data is valid so that it can be used to calculate the output's scaling. This is helpful if the scaling propagation function needs to be executed several times until an inport's scaling data becomes valid. Refer to the two examples in this section.

The M script assigns `i` the `Ret_In(i)` structure for each inport and `j` the `Ret_Out(j)` structure for each outport.

Structure for inport specification `Ret_In(i)`	Structure for outport specification `Ret_Out(j)`

`Ret_In(i).Invariant`	`Ret_Out(j).Invariant`
	`Ret_Out(j).Lsb`
	`Ret_Out(j).Offset`
	`Ret_Out(j).Min`
	`Ret_Out(j).Max`

If the Invariant field is set to 1, the inport/outport is marked as invariant to the scaling. The fields LSB, Offset, Min and Max are to be calculated by the M script.

Access to system handle In the scaling propagation function, you can use the `SISystem` property. This property returns the Simulink block handle of the scaling-invariant subsystem which the scaling propagation function belongs to. This is particularly helpful if the scaling depends not only on inputs but also on instance-specific block data, e.g., specified in mask variables. If the scaling propagation function belongs to a graphical function (Stateflow), the property returns the handle of the associated Stateflow chart.

For example:

```
SISystemName = get_param(SISystem, 'Name');
```

Inports and outports

A scaling propagation function must not necessarily address all the inports and outports of a subsystem. It is possible to specify just certain ports as scaling-invariant.

Mapping outports to inports For the outports of scaling-invariant subsystems, as an option you can specify the inport(s) which an outport's scaling should depend on.

The following conditions must be met:

- The UtilizeExplicitDependenciesForScalingInvariantSystems Code Generator option must be enabled.

  Note

  The port mapping is helpful if the scaling-invariant subsystem has a *feedback loop* (refer to Example #2: Feedback loop). If this option is disabled, TargetLink assumes that the outputs of a scaling-invariant subsystem depend on *all* the inputs, which is a worst-case scenario.

- Scaling propagation function:
  - The ports can be specified by their port indices, e.g.:

    ```
    Ret_Out(1).DependsOnInports = [1 2];
    ```

- The `IsDataValid` flag indicates if an inport's scaling data is valid, e.g.:

```
if SII_In(1).IsDataValid && SII_In(2).IsDataValid
   % calculate the scaling and ranges of the first outport
   ...
end
```

Error handling and Code Generator messages

Using the predefined `Failure` and `FailureMsg` variables in your M scripts enables you to identify the blocks that do not meet the preconditions for implementing scaling-invariant functions. `Failure` is a Boolean control variable (1 means true), and `FailureMsg` provides the message text. If the Failure variable is set to 1, TargetLink emits a message containing the message text specified in the FailureMsg variable. You can highlight the block that causes the message via the Show button in the dSPACE Message Browser.

For details, refer to the `else` branch in the example below.

Restriction on the initialization of state variables

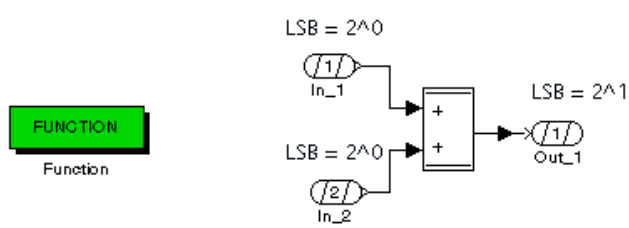
If a scaling-invariant subsystem contains state variables that interact with the output of inports marked as scaling-invariant, the following restriction applies to such state variables:

- Do not use initial values other than 0, otherwise SIL/PIL simulation differs from MIL simulation.

A possible workaround is to place blocks which contain state variables outside the scaling-invariant subsystem.

Example #1: 1-bit left shift

Suppose you have the following use case: the inports of a subsystem both have an LSB = 2^0. The LSB of the outport can be derived from the LSB of the inports by applying a 1-bit left shift (1<<). Thus the scaling propagation function is to represent a 1-bit left shift.



The specification of the scaling propagation function may read as follows:

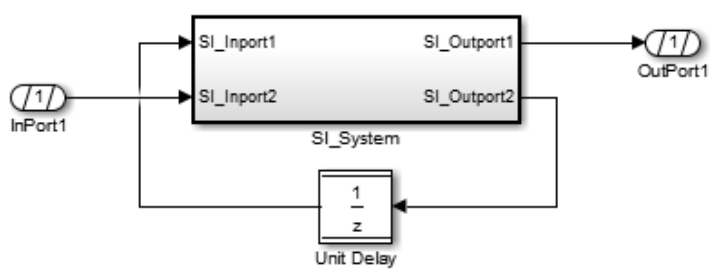
```
% Mark all inports and outports as scaling-invariant
Ret_In(1).Invariant  = 1;
Ret_In(2).Invariant  = 1;
Ret_Out(1).Invariant = 1;

if SII_In(1).IsDataValid && SII_In(2).IsDataValid

    if      isequal(SII_In(1).Lsb, SII_In(2).Lsb) ...
      && isequal(SII_In(1).Offset, SII_In(2).Offset) ...
      && strcmp(SII_In(1).Type, 'Int16') ...
      && strcmp(SII_In(2).Type, 'Int16')

        % Calculate the scaling and the ranges of the output
        Ret_Out(1).Lsb    = SII_In(1).Lsb * 2;
        Ret_Out(1).Offset = SII_In(1).Offset + SII_In(2).Offset;
        Ret_Out(1).Min    = SII_In(1).Min + SII_In(2).Min;
        Ret_Out(1).Max    = SII_In(1).Max + SII_In(2).Max;
    else
        % Error: Unsatisfied preconditions
        Failure = 1;
        FailureMsg = ['Warning: The input signals do not satisfy the precondition of the subsystem! ' ...
                      'Either the Scalings of the input signals are not equal or ' ...
                      'the input signals do not have the required type. ', ...
                      'Scaling invariance will not be used!'];
    end
end
```

Note

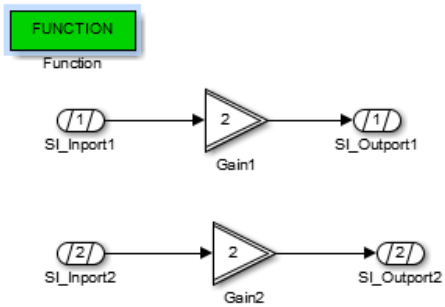
The specification of the scaling propagation function does not appear in the generated code.

Example #2: Feedback loop

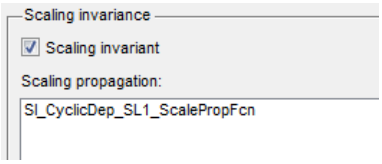
Suppose you have the following subsystem, which has a feedback loop.



The inputs and the outputs are connected as follows:



The scaling propagation function is specified on the Arguments page of the Function's block dialog.



The `SI_CyclicDep_SL1_ScalePropFcn.m` file reads as follows:

```
% Mark all inports and outports as scaling-invariant
Ret_In(1).Invariant  = 1;
Ret_In(2).Invariant  = 1;
Ret_Out(1).Invariant = 1;
Ret_Out(2).Invariant = 1;

if SII_In(1).IsDataValid
    if  strcmp(SII_In(1).Type, 'Int16')

        % Calculate the scaling and the ranges of output 1
        Ret_Out(1).Lsb    = SII_In(1).Lsb * 2;
        Ret_Out(1).Offset = SII_In(1).Offset;
        Ret_Out(1).Min    = SII_In(1).Min;
        Ret_Out(1).Max    = SII_In(1).Max;
    else
        % ------------------------------------------------------------
        % Error: Preconditions for invariant scaling are not satisfied.
        % ------------------------------------------------------------
        Failure = 1;

        FailureMsg = 'Error: The data type of the input signal of Inport 1 is not Int16';
    end
end

if SII_In(2).IsDataValid
    if strcmp(SII_In(2).Type, 'Int16')

        % Calculate the scaling and the ranges of output 2
        Ret_Out(2).Lsb    = SII_In(2).Lsb * 2;
        Ret_Out(2).Offset = SII_In(2).Offset;
        Ret_Out(2).Min    = SII_In(2).Min;
        Ret_Out(2).Max    = SII_In(2).Max;
    else
        % ------------------------------------------------------------
```

```
        % Error: Preconditions for invariant scaling are not satisfied.
        % ------------------------------------------------------------
        Failure = 1;
        FailureMsg = 'Error: The data type of the input signal of Inport 2 is not Int16';
    end
end
```

Related topics

HowTos

How to Specify Subsystems as Scaling-Invariant 329

References

UtilizeExplicitDependenciesForScalingInvariantSystems (📖 TargetLink Model Element Reference)

How to Specify Subsystems as Scaling-Invariant

Objective

To use a subsystem without additional code due to rescaling at the inports and outports, you must specify it as scaling-invariant.

Characteristics of scaling-invariant subsystems

If you want to specify a subsystem as scaling-invariant, the following rules apply:

- Incoming and outgoing signals must have the same base data type as their related port.
- The constrained limits (Min, Max) of merged, scaling-invariant output variables must match.
- The function interface of the subsystem must be completely specified by means of TargetLink **InPort** and **OutPort** blocks.

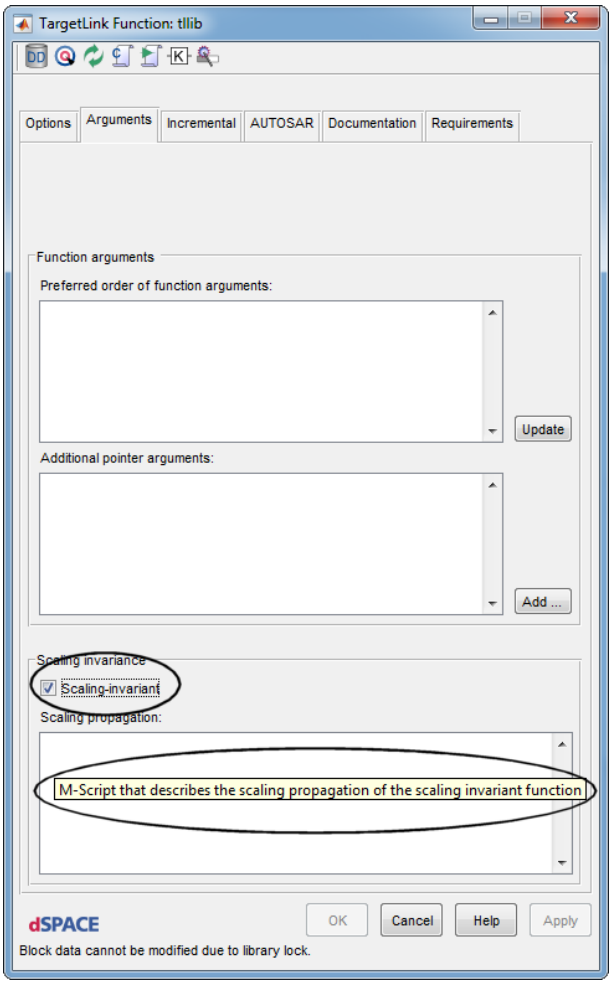
Method

To specify a subsystem as scaling-invariant

1. Insert a **Function** block.

   This ensures that the program code of the subsystem is executed within a function.

2 Open the dialog of the **Function** block and select the **Arguments** page.



3 Select the **scaling-invariant** checkbox.

The **Scaling propagation** edit field is enabled.

4 Enter the scaling propagation function (M script) in the **Scaling propagation** edit field. For a specification example of a scaling propagation function, refer to Details on the Scaling Propagation Function on page 324.

Result

The subsystem is specified as scaling-invariant.

Tip

You should implement scaling-invariant functions as reused functions. This gives you the advantage of implementing only a single function, but with no scaling adjustment code. Refer to Reusing C Functions and Variables for Identical Model Parts (Function Reuse) on page 297.

Related topics

Basics

Details on the Scaling Propagation Function..... 324
Reusing C Functions and Variables for Identical Model Parts (Function Reuse)..... 297

References

Function Block (TargetLink Model Element Reference)

How to Specify Stateflow Charts and Graphical Functions as Scaling-Invariant

Objective

To use a Stateflow chart or a Stateflow graphical function without additional code due to rescaling at the inports and outports, you must specify it as scaling-invariant.

Preconditions

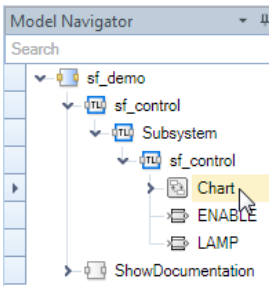
- A model must be open.
- Both charts and graphical functions must have scaled integer inputs and scaled integer outputs.
- Incoming and leaving signals must have the same base data type as their related port.
- The constrained limits (Min, Max) of merged, scaling-invariant output variables must match.

Method

To specify a Stateflow chart or a graphical function as scaling-invariant

1 In the MATLAB Command Window, type `tlPropman('Start')` to open the Property Manager.

2 In the Model Navigator, select the Stateflow chart or graphical function.

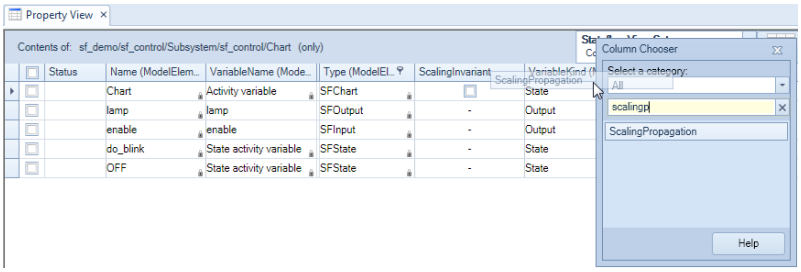


The properties of the related model elements are displayed in the Property View.

3 Configure the Property View to display the relevant properties.

For example, select the **Stateflow View Set**, which contains the Stateflow object properties. Refer to How to Optimize the Property View for Your Use Cases (📖 TargetLink Preparation and Simulation Guide).

Add the **ScalingInvariant** and the **ScalingPropagation** columns via the **Column Chooser**. For more information, refer to How to Customize Columns in the Property View and Validation Summary (📖 TargetLink Preparation and Simulation Guide).

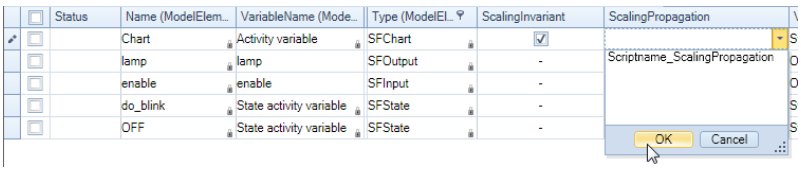


4 Select the checkbox in the **ScalingInvariant** column.

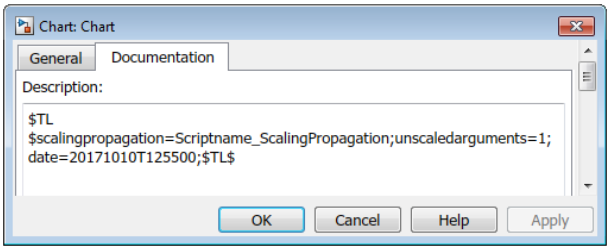
Note

The **unscaledarguments** API property corresponds to the scaling-invariant checkbox of the **Function** block. For more information, refer to How to Specify Subsystems as Scaling-Invariant on page 329.

5 In the **ScalingPropagation** column, enter the name of the script that contains the scaling propagation function and click **OK**.



The **Description** field on the **Documentation** page of the chart's **Properties** dialog looks as in the following illustration.



Result

You specified a Stateflow chart or a graphical function as scaling-invariant.

Tip

You can implement scaling-invariant Stateflow charts as reused functions. This gives you the advantage of implementing only a single function, but with no scaling adjustment code. Refer to Reusing C Functions and Variables for Identical Model Parts (Function Reuse) on page 297.

Related topics

Basics

Modifying Multiple Properties at Once via the Property Manager (📖 TargetLink Preparation and Simulation Guide)
Reusing C Functions and Variables for Identical Model Parts (Function Reuse)..... 297
Working with Stateflow (📖 TargetLink Preparation and Simulation Guide)

HowTos

How to Specify Subsystems as Scaling-Invariant..... 329

Controlling Subsystem Execution

Introduction

When implementing control flows with TargetLink you should familiarize yourself with the modeling requirements described below.

Where to go from here

Information in this section

Basics on Controlling Subsystem Execution..... 334

Basics on Resettable Subsystems..... 335

Details on Implementing Externally Function-Call-Triggered Subsystems..... 336

Details on Implementing Function-Call-Triggered Subsystems..... 340

Details on Executing Enabled Subsystems..... 342

Details on Executing Triggered Subsystems..... 342

State Reset in Conditionally Executed Subsystems: Differences of Simulink and TargetLink..... 345

Basics on Controlling Subsystem Execution

Supported methods

Simulink provides several methods of implementing control flows and thus controlling the execution of subsystems. The most important methods supported by TargetLink are:

- Event-based control flow (external and internal function-call triggered subsystems)
- Data-based control flow (enabled or edge-triggered subsystems)

Additional information on creating control flows, for example, on using a For Iterator subsystem, is available in Modeling Control Flows without Using Stateflow (🕮 TargetLink Preparation and Simulation Guide).

Note

There are important limitations which you have to take into account when implementing conditionally executed subsystems. For details, refer to Subsystem Creation Limitations (🕮 TargetLink Limitation Reference).

Event-based control flow

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. For information on events that originate outside a TargetLink subsystem, refer to Details on Implementing Externally Function-Call-Triggered Subsystems on page 336. Events originating and ending inside a subsystem are described in Details on Implementing Function-Call-Triggered Subsystems on page 340.

Note

Currently, TargetLink does not support function-call events that originate inside a TargetLink subsystem and control subsystems or Stateflow blocks outside it.

Data-based control flow

When implementing a data-based control flow, you can control the execution of a subsystem either by adding an Enable block to a subsystem, which means you are implementing an enabled subsystem, or by adding a Trigger block implementing a triggered subsystem. For information on the two implementation methods, refer to Details on Executing Enabled Subsystems on page 342 and Details on Executing Triggered Subsystems on page 342.

Related topics

Basics

Details on Executing Enabled Subsystems..... 342
Details on Executing Triggered Subsystems..... 342
Modeling Control Flows without Using Stateflow (📖 TargetLink Preparation and Simulation Guide)

HowTos

How to Specify Conditional Control Flows via Enabled Subsystems..... 621

References

Subsystem Creation Limitations (📖 TargetLink Limitation Reference)

Basics on Resettable Subsystems

Resetting state variables of blocks in resettable subsystems

Resettable subsystems conditionally reset state variables of contained blocks before execution. You can use resettable subsystems to simplify modeling: Suppose your model contains several blocks and you want to reset their state

variables. If you include all blocks in a resettable subsystem, you do not need to work on each block manually. Create resettable subsystems in two ways:

- You can insert a Simulink Resettable Subsystem in your TargetLink subsystem.
- You can copy the ResetPort block from inside a Resettable Subsystem and insert it into a subsystem to make the subsystem resettable.

When an external trigger is received the resettable subsystem resets all state variables in its contained blocks before it is executed. In contrast to enabled subsystems, which must be inactive before their states are reset, resettable subsystems are always active. The reset and the subsequent execution of the resettable subsystem are performed in the same time step.

Note

The following limitation applies when working with ResetPort blocks:

- You cannot place a ResetPort block on the root level of your TargetLink subsystem.

Related topics

Basics

Details on Implementing Function-Call-Triggered Subsystems..... 340

References

Code-Relevant Simulink Blocks (📖 TargetLink Model Element Reference)

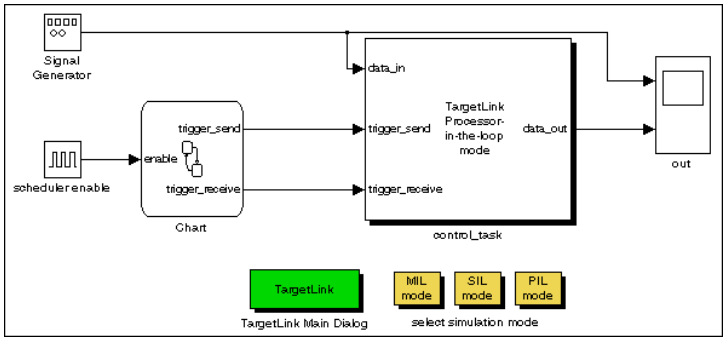
Details on Implementing Externally Function-Call-Triggered Subsystems

External function-call-triggered subsystems

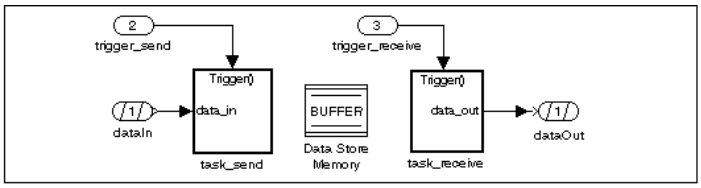
External function-call-triggered subsystems are a powerful means to integrate generated code into an environment where the function calls are controlled from outside the TargetLink subsystem.

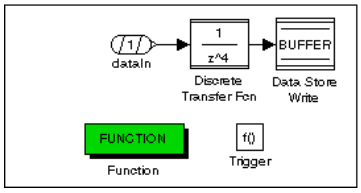
Example

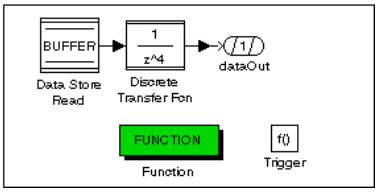
This example shows how to control a function-call-triggered subsystem from outside the TargetLink subsystem.



Suppose the task scheduler is modeled as a state-machine via a Stateflow chart. The two function-call events trigger_send and trigger_receive are sent alternatively whenever the scheduler is enabled. The following illustration shows two subsystems that are implemented as the user-defined functions `send()` and `receive()`:







Note

- To ensure data consistency, TargetLink specifies global communication between externally-called functions automatically.
- When externally-called functions are used, TargetLink cannot ensure the correct handling of concurrent executions, for example, any *read-after-write* and *write-after-read* dependencies. You must therefore ensure correct handling yourself.

Example

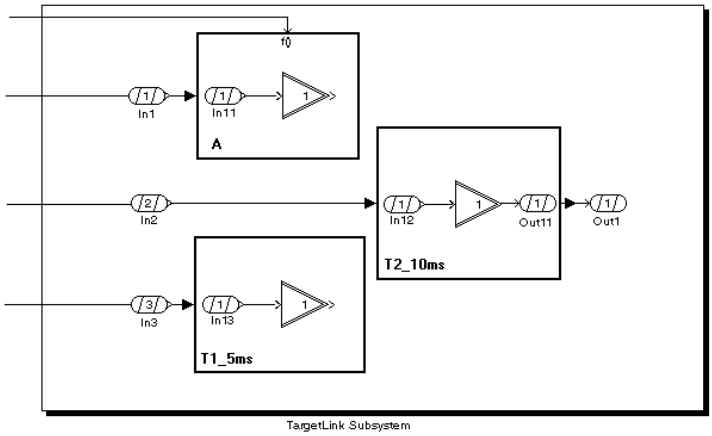
With these settings for the example, TargetLink generates the following code: two externally called functions: `send()`, `receive()` and an empty model step function: `control_task()`.

```
...
void receive(void)
{
   /* SLStaticLocalInit: Default storage class for static local variables with
      initvalue */
   static Float32 X_Sa2_Discrete_Transfer_Fcn[4] =
   {
      /*[0..3]*/ 0.F, 0.F, 0.F, 0.F
      /* 0.F, 0.F, 0.F, 0.F */
   };
   /* Discrete Transfer Function: output */
   data_out = X_Sa2_Discrete_Transfer_Fcn[3];
   /*  --- update section ---  */
   X_Sa2_Discrete_Transfer_Fcn[3] = X_Sa2_Discrete_Transfer_Fcn[2];
   X_Sa2_Discrete_Transfer_Fcn[2] = X_Sa2_Discrete_Transfer_Fcn[1];
   X_Sa2_Discrete_Transfer_Fcn[1] = X_Sa2_Discrete_Transfer_Fcn[0];
   X_Sa2_Discrete_Transfer_Fcn[0] = BUFFER;
}
...
void send(void)
{
   /* SLStaticLocalInit: Default storage class for static local variables with
      initvalue */
   static Float32 X_Sa3_Discrete_Transfer_Fcn[4] =
   {
      /*[0..3]*/ 0.F, 0.F, 0.F, 0.F
      /* 0.F, 0.F, 0.F, 0.F */
   };
   /* Discrete Transfer Function: output */
   BUFFER = X_Sa3_Discrete_Transfer_Fcn[3];
   /*  --- update section ---  */
   X_Sa3_Discrete_Transfer_Fcn[3] = X_Sa3_Discrete_Transfer_Fcn[2];
   X_Sa3_Discrete_Transfer_Fcn[2] = X_Sa3_Discrete_Transfer_Fcn[1];
   X_Sa3_Discrete_Transfer_Fcn[1] = X_Sa3_Discrete_Transfer_Fcn[0];
   X_Sa3_Discrete_Transfer_Fcn[0] = Sa3_data_in;
}
...
```

Simulink Inport and Outport blocks on the topmost level of a TargetLink subsystem must be connected to the corresponding TargetLink **InPort/Bus Inport** and **OutPort/Bus Outport** blocks. Refer to Subsystem Creation Limitations (📖 TargetLink Limitation Reference). During production code generation, TargetLink ignores these TargetLink port blocks if their ⍰ variable class is default and no saturation is specified for them. Additionally, the base types and scaling parameters must be the same as the base types and scaling parameters of the TargetLink port block in the subsystem they are connected to.

Example model

Consider the following example model:



In the model above, the TargetLink **InPort** block In1 is ignored during production code generation, if all of the following preconditions are fulfilled:

- Variable class (In1) = default
- No saturation is specified for In1, i.e., the Min and Max edit fields on the Output page of the In1 (InPort) dialog must be empty.
- Scaling (In1) = scaling (In11)
- Base type (In1) = base type (In11)

The same applies to In2, In3, and Out1. Additionally, the Saturation checkboxes on the Output page of the Out1 (OutPort) dialog must be cleared for Out1.

Related topics

Basics

Details on Executing Enabled Subsystems..... 342
Details on Executing Triggered Subsystems..... 342
Details on Implementing Function-Call-Triggered Subsystems..... 340

HowTos

How to Specify Function Interfaces via Function Parameters..... 242

References

Subsystem Creation Limitations (📖 TargetLink Limitation Reference)

Details on Implementing Function-Call-Triggered Subsystems

Function-call-triggered subsystem

A function-call event is sent from its source to its sink. Stateflow charts may 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.

Code generation and execution

To generate code and execute a function-call-triggered subsystem, the subsystem must be triggered by a function-call source. If it is not triggered (not executed), neither the function call nor the function definition is available in the generated code.

If function-call-triggered subsystems are dependent on each other, they can directly exchange data with one another by means of data store semantics (type `sl_subsys_semantics` in the MATLAB Command Window for details).

Note

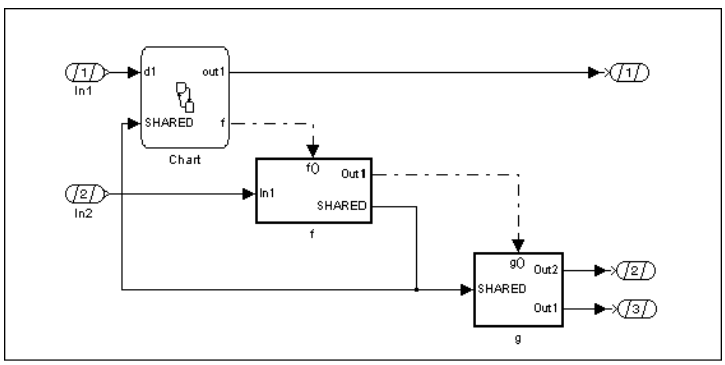
The following limitation applies when implementing function-call-triggered subsystems:

- TargetLink does not support the **reset** setting of the **Output when disabled** property of the Outport block in a function-call-triggered subsystem.

For details, refer to Subsystem Creation Limitations (📖 TargetLink Limitation Reference).

Data sharing between function-call subsystems

The model below shows how data sharing between function-called subsystems using data store semantics can be realized.



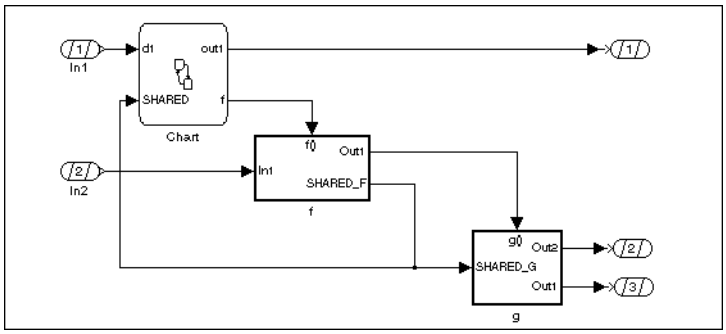
In this model the Stateflow chart and both subsystems (f and g) share a single variable for data exchange, named SHARED. If **default** is selected for this variable in the chart and in both subsystems, TargetLink generates code according to the data store semantics.

Note

- User-specified function interfaces may interfere with the data store semantics of function-call-triggered subsystems.
- TargetLink accepts conflicting specifications, which means you can use either strict modeling semantics (data store semantics) or your own user-specific function interface.

Example

Suppose the model above has been modified so that the Stateflow variable SHARED is left at its **default** settings, but the variables of both subsystems (f and g) are implemented as separate **global** variables, named SHARED_F and SHARED_G.



Instead of using SHARED, the code generated for the Stateflow chart uses SHARED_F. To ensure correct computation results of function g, SHARED_G must be updated with SHARED_F before the function is called. The code that performs the update is as follows:

```
static void Sa2_f(void)
{
      /* update of inport  for function Subsystem/g. */
      SHARED_G = SHARED_F;
      /* Call of function: Sa3_g */
      Sa3_g();
}
```

Related topics

Basics

Details on Executing Enabled Subsystems..... 342
Details on Executing Triggered Subsystems..... 342
Details on Implementing Externally Function-Call-Triggered Subsystems..... 336

References

Subsystem Creation Limitations (📖 TargetLink Limitation Reference)

Details on Executing Enabled Subsystems

Enabled subsystems

Enabled subsystems are executed if the enabling signal is greater than zero. They 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*.

Note

There are important limitations which you have to take into account when implementing conditionally executed subsystems. For details, refer to Subsystem Creation Limitations (📖 TargetLink Limitation Reference).

Tip

An example of an enabled subsystem, named `ENABLSYS`, is available in the tllib_demos library.

Related topics

Basics

Basics on Controlling Subsystem Execution..... 334
Details on Executing Triggered Subsystems..... 342
Details on Implementing Externally Function-Call-Triggered Subsystems..... 336
Details on Implementing Function-Call-Triggered Subsystems..... 340

HowTos

How to Specify Conditional Control Flows via Enabled Subsystems..... 621

References

Subsystem Creation Limitations (📖 TargetLink Limitation Reference)

Details on Executing Triggered Subsystems

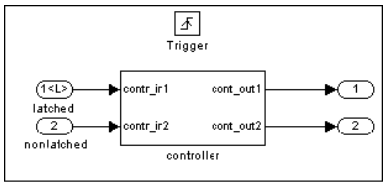
Triggered subsystems

A triggered subsystem is executed when a rising or a falling edge (with respect to zero) is detected on the signal driving the trigger port. Thus, you can control the initial value of the output signal and the method with which an input signal is transferred inside the triggered subsystem.

The transfer of an incoming signal can be implemented as:

- Latched, which involves delayed data transfer to the output of the Simulink inport. This is also known as hardware triggering.
- Nonlatched, which involves direct and immediate data transfer to the output of the Simulink inport. This is also known as software triggering.

You can configure each inport independently, that is, latched or nonlatched. The subsystem controller shown below is triggered when a rising edge is detected.



Note

There are important limitations which you have to take into account when implementing conditionally executed subsystems. For details, refer to Subsystem Creation Limitations (📖 TargetLink Limitation Reference).

The code excerpt shows that the first parameter `Sa2_latched` of the `Sa3_controller` function is delayed by one time step compared to the second parameter `Sa1_InPort2.`

```
void latch_semantics(
   Int16   Sa1_InPort   /* LSB: 2^0 OFF:  0 MIN/MAX: -32768 .. 32767 */,
   ...
{
   /*
      defaultClass_slStaticLocalInit: Default storage class for
      static local variables with initvalue */
   ...
   static Int16   Sa2_latched = 0; /* LSB: 2^0 OFF: 0 MIN/MAX: -32768 .. 32767 */
   /* Condition for rising edge trigger */
   if ( TriggerCondition ) {
      /* defaultClass_slLocal: Default storage class for local variables */
      ...
     /* Call of function: latch_semantics/Subsystem/controller
     # combined # update of inport  for function latch_semantics/Subsystem. */
  Sa3_controller(
     Sa2_latched,
     Sa1_InPort2,
     &(Sa3_cont_out1),
     &(Sa3_cont_out2));
      /* Outport: latch_semantics/Subsystem/latched_out */
   ...
   /* ---- Update code of subsystem: latch_semantics ---- */
    /* update of inport  for function latch_semantics/Subsystem. */
   Sa2_latched = Sa1_InPort;
 }
```

Time evaluation in triggered subsystems

To calculate the output signal of blocks that reside in triggered subsystems, one of the following time values is required:

- The current sample time value
- The elapsed time value with respect to the last sample step

Sample time

The current sample time value of a triggered subsystem is commonly calculated by an independent counter or timer. In TargetLink, the sample time is represented by the SystemTime variable. In addition, it encapsulates the details of access to the related counter or timer.

> **Note**
>
> The sample time is to be inherited from the driving block, as the time between two samples (elapsed time) is not necessarily constant in triggered subsystems.

TargetLink Data Dictionary TargetLink provides the SystemTime variable in `/Pool/Variables/RTOS/SystemTime` in the TargetLink Data Dictionary.

SIL and PIL simulation During a SIL and PIL simulation, the SystemTime variable is automatically updated with the current simulation time if needed.

Access function As you can specify an access function via the variable class of the SystemTime variable (see Basics on Access Functions on page 270), you can adapt access to the current time value to your real environment.

Elapsed time

To calculate the time elapsed, the following two sample time values are required:

- Current sample time value T(k)
- Previous sample time value T(k-1)

The elapsed time T is calculated as follows:

T = T(k) -T(k-1)

T is calculated only once within a subsystem. All blocks that require it use the result of this calculation.

For a triggered subsystem, the calculation of the elapsed time can be implemented as follows:

```
static UInt16 Sb2_LastTime = 0;
/* Subsystem: Access to the current system time */
Sb2_CurrentTime = GetSystemTime();
/* Subsystem: Elapsed time calculation */
Sb2_ElapsedTime = (UInt16)(Sb2_CurrentTime - Sb2_LastTime);
```

Overflow management

An overflow of the SystemTime variable does not necessarily lead to wrong calculation of the elapsed time value. The main thing is that the elapsed time of the trigger event does not exceed the physical range of the SystemTime variable.

Otherwise, the subsystem must be restarted just before the elapsed time of the trigger event exceeds the physical range of the SystemTime variable.

Tip

To calculate T, TargetLink applies the 'compute-through-overflow' principle (see Compute-Through-Overflows Property of Two's Complement Arithmetic (📖 TargetLink Preparation and Simulation Guide)).

Related topics

Basics

Compute-Through-Overflows Property of Two's Complement Arithmetic (📖 TargetLink Preparation and Simulation Guide)
Details on Executing Enabled Subsystems..... 342
Details on Implementing Externally Function-Call-Triggered Subsystems..... 336
Details on Implementing Function-Call-Triggered Subsystems..... 340

References

Subsystem Creation Limitations (📖 TargetLink Limitation Reference)

State Reset in Conditionally Executed Subsystems: Differences of Simulink and TargetLink

Introduction

For specific modeling approaches, Simulink and TargetLink perform different state resets in conditionally executed subsystems. In consequence, TargetLink cannot reset the states in the same way as Simulink does. This can lead to differences of MIL and SIL/PIL simulations.

State reset in TargetLink subsystems

TargetLink does not perform a state reset if both of the following two conditions apply to the TargetLink subsystem:

- The TargetLink subsystem is an **Enabled Subsystem** or **Function-Call-Triggered Subsystem** whose **States when enabling** parameter set to **Held**.
- The TargetLink subsystem calls a **Function-Call-Triggered Subsystem** whose **States when enabling** parameter is set to **Reset**.

TargetLink does not perform a state reset in subsystem B if all the following three conditions apply:

- The TargetLink subsystem is an **Enabled Subsystem** or **Function-Call-Triggered Subsystem** whose **States when enabling** parameter set to **Reset**.

- The TargetLink subsystem calls an **Enabled Subsystem** or **Function-Call-Triggered Subsystem** (subsystem A) whose **States when enabling** parameter is set to **Held**.
- Subsystem A calls an **Enabled Subsystem** or **Function-Call-Triggered Subsystem** (subsystem B) whose **States when enabling** parameter is set to **Reset**.

State reset in function-call subsystems

TargetLink's state reset behavior in a called subsystem differs from Simulink's state reset behavior if the subsystem is called via multiple paths by different calling subsystems, and the subsystems have the following characteristics:

Scenario #1

Called subsystem The called subsystem is a function-called-triggered subsystem that has its **States when enabling** parameter set to **inherit**.

Calling subsystems At least one of the calling subsystems is of one of the following subsystem types:

- an enabled subsystem
- a triggered and enabled subsystem
- an action-port-triggered subsystem

The **States when enabling** or **States when action is resumed** parameter of the calling subsystems is set to **reset**.

Scenario #2

Called subsystem The called subsystem is a function-called-triggered subsystem that has its **States when enabling** parameter set to **reset**.

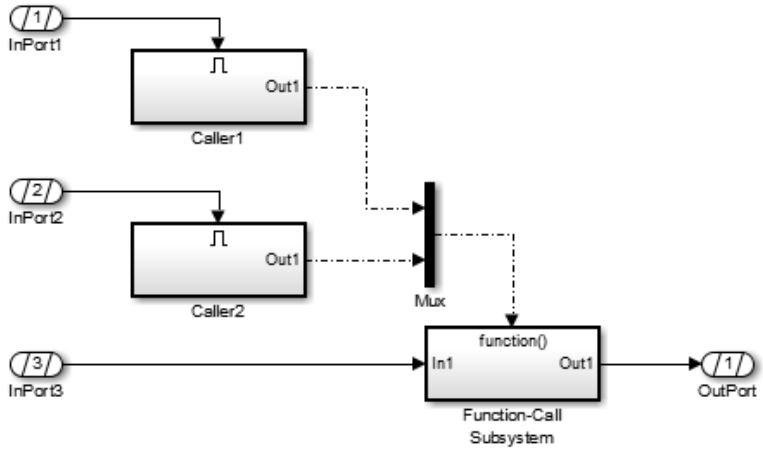
Calling subsystems At least one of the calling subsystems is of one of the following subsystem types:

- an enabled subsystem
- a triggered and enabled subsystem
- an action-port-triggered subsystem

It does not matter whether the calling subsystem has its **States when enabling** or **States when action is resumed** parameter set to **held** or **reset**.

Example of state reset in function-call subsystems

In this example, the **Function-Call Subsystem** is the system called, its **States when enabling** parameter is set to **inherit**. The **Caller1** and **Caller2** subsystems are calling systems that are also enabled subsystems. Their **States when enabling** parameter is set to **reset**.



State reset in enabled subsystems

TargetLink's state reset behavior in a called subsystem differs from Simulink's state reset behavior if the subsystem is called via multiple paths by different calling subsystems:

Called subsystem The called subsystem is an enabled or an enabled and triggered subsystem with its States when enabling parameter set to reset at the enable port.

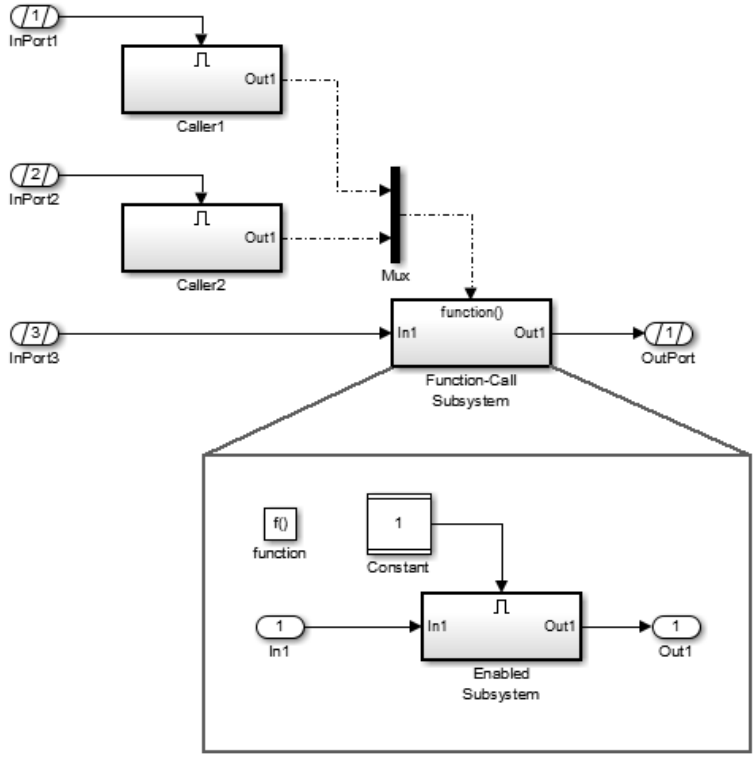
Calling subsystems At least one of the calling subsystems is of one of the following subsystem types:

- an enabled subsystem
- a triggered and enabled subsystem
- an action-port-triggered subsystem

It does not matter whether the calling subsystem has its States when enabling or States when action is resumed parameter set to held or reset.

Example of state reset in enabled subsystems

In this example, the Enabled Subsystem subsystem is the system called, its States when enabling parameter is set to reset. The Caller1 and Caller2 subsystems are calling systems that are also enabled subsystems. Enabled Subsystem is called via multiple paths, because the function-call signals are emitted by several subsystems, i.e., Caller1 and Caller2.



State reset in action-port-triggered subsystems

TargetLink's state reset behavior in a called subsystem differs from Simulink's state reset behavior if the subsystem is called via multiple paths by different calling subsystems, and the subsystems have the following characteristics:

Called subsystem The called subsystem is an action-port-triggered subsystem with the action port's States when action is resumed parameter set to reset.

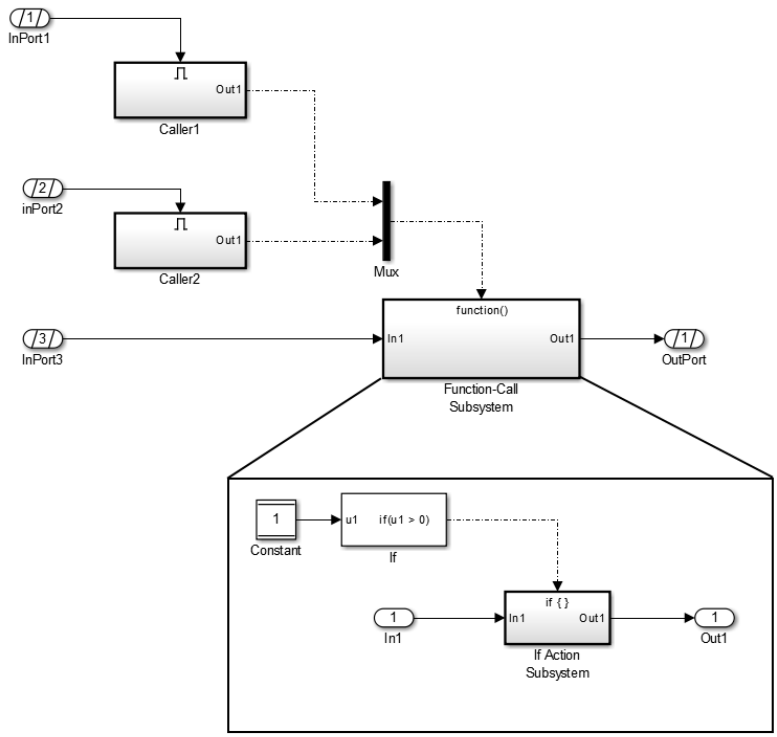
Calling subsystems At least one of the calling subsystems is of one of the following subsystem types:

- an enabled subsystem
- a triggered and enabled subsystem
- an action-port-triggered subsystem

It does not matter whether the calling subsystem has its States when enabling or States when action is resumed parameter set to held or reset.

Example of state reset in action-port-triggered subsystems

In this example, the If Action Subsystem subsystem is the system called, its States when action is resumed parameter is set to reset. The Caller1 and Caller2 subsystems are calling systems that are enabled subsystems. The If Action Subsystem is called via multiple paths, because the function-call signals are output by different subsystems, i.e., Caller1 and Caller2.



Avoiding simulation differences

Use one of the following actions to avoid different state resets with Simulink and TargetLink:

- Set the **States when enabling** or **States when action is resumed** parameter of the called **Enabled Subsystem** subsystem to **held**.
- Avoid calling the called subsystem via multiple paths.

Related topics

Basics

Details on Executing Enabled Subsystems..... 342
Details on Executing Triggered Subsystems..... 342

References

Subsystem Creation Limitations (📖 TargetLink Limitation Reference)

Controlling Function Inlining

Introduction

Function inlining is used to embed the code for a generated function in the code of the calling subsystem's function. This eliminates the overhead for the function call statement and possibly allows further optimization by TargetLink.

Where to go from here

Information in this section

Basics on Function Inlining 350
How to Inline Specific Functions During Code Generation 353
How to Flag a Function for Compiler Inlining 354
Example of Inlining a Function During Code Generation 355

Basics on Function Inlining

Introduction

Inline expansion is the process of replacing a function call with the code contained in the body of the function. TargetLink supports inline expansion as follows:

- Function inlining during code generation by TargetLink. Refer to Automatic function inlining during code generation on page 350.
- Compiler inlining during compilation by the C compiler. Refer to Explicitly specifying functions for C compiler inlining on page 351.

Automatic function inlining during code generation

By default, TargetLink automatically inlines functions during code generation if this increases code efficiency. You can partly influence automatic inlining via the InliningThreshold Code Generator option.

When TargetLink inlines a function, additional code transformations can be applied internally,including the evaluation of ranges. This results in more efficient code.

Inlining threshold You can specify the desired value for the Inlining threshold on the Advanced page of the TargetLink Main Dialog. If you set the inlining threshold to `inf`, TargetLink always inlines the generated code. If you set it to `0`, TargetLink never inlines the generated code.

Note

- Functions generated for function-call-triggered, edge-triggered or enabled subsystems that do not contain a Function block might be inlined.
- Functions generated for subsystems containing a Function block are not inlined if their function class is set to `default`.
- For Stateflow blocks, functions are mainly used to implement actions of charts and graphical functions as well as entry and exit actions of states. These functions will also be inlined.

Example There is a tradeoff between the size of the code and its execution speed. If a subsystem is controlled by multiple trigger signals, inlining results in multiple copies of the code. For large functions, the code size increases significantly. If the code of the function contains only a few statements, the function call overhead can exceed the function implementation. In this case, inlining results in only a small increase in code size (if any), but the overall execution time is reduced.

Explicitly specifying functions for inlining during code generation

By using the Optimization property of the DD FunctionClass object, you can explicitly specify whether to inline a function.

Property Value[1)]	Description
`CG_AUTO_INLINE`	If the `CG_AUTO_INLINE` property value is set, inlining the function algorithm of functions that reference the related DD FunctionClass object is allowed (but not enforced). Whether function inlining is performed depends on the value of the InliningThreshold Code Generator option and if inlining is technically possible.
`CG_FORCE_INLINE`	If the `CG_FORCE_INLINE` property value is set, inlining the function algorithm of functions that reference the related DD FunctionClass object is always performed (if technically possible).
`CG_NO_INLINE`	If the `CG_NO_INLINE` property value is set, inlining the function algorithm of functions that reference the related DD FunctionClass object is never performed.

1) The property values `CG_AUTO_INLINE` , `CG_FORCE_INLINE` and `CG_NO_INLINE` are mutually exclusive. If you select more than one of the mutually exclusive property values, an error messages occurs. If none of the three property values is selected, TargetLink does not perform inlining for functions associated with this function class.

In general, function inlining is always used for functions called only once provided that the Optimization property is not explicitly set to `CG_NO_INLINE` or left empty. For `CG_AUTO_INLINE`, the value of the InliningThreshold Code Generator option must be greater than `0`.

Example For an example, refer to Example of Inlining a Function During Code Generation on page 355.

Explicitly specifying functions for C compiler inlining

Via the CompilerInline property of DD FunctionClass objects, you can instruct TargetLink to generate the declarations and definitions of functions to contain a compiler-specific inline keyword.

Note

By default, compiler inlining works only for the `Generic C99` code generation target and Target Optimization Modules (TOM). You can also support arbitrary platforms as described in the *Generating TargetLink Code and ASAP2 Files for an Arbitrary Platform* application note, which you can download from the TargetLink Support Center.

In TargetLink, C compiler inlining is restricted to functions with static storage duration.

Specifying the inline keyword The inline keyword must be specified as a `ddObj` XML element in the respective `TargetConfig.xml` file:

```
<ddObj name="InlineKeywords" Kind="GenericObject">
   <ddProperty Name="DeclarationKeyword" Type="String">__inline</ddProperty>
   <ddProperty Name="DefinitionKeyword" Type="String">__inline</ddProperty>
</ddObj>
```

Note

TargetLink preserves whitespaces and line breaks within the `ddProperty` XML elements. This lets you control whether the keyword is placed in the same line as the definition or declaration of the function.

Not all compilers can inline all kinds of functions in all situations. This can be the case for the following functions:

- Recursive functions
- Functions that are referenced by a pointer elsewhere in the translation unit
- Functions that contain local static variables

As with normal functions, there is no defined order in which the arguments are evaluated for an inline function. This means that the evaluation order might differ if the arguments are passed with a normal function call protocol. For more information, refer to the documentation of your target compiler.

Example Example code for a function generated for the Generic C99 target looks like this:

```
inline static void MyInlineFunction(void)
{
   /* TargetLink outport: Subsystem/Subsystem/OutPort
      # combined # Gain: Subsystem/Subsystem/Gain */
   Sa2_OutPort = Sa2_InPort;
}
```

Related topics

Basics

Basics on Configuring the Code Generator for Production Code Generation..... 49

HowTos

How to Flag a Function for Compiler Inlining..... 354
How to Inline Specific Functions During Code Generation..... 353

Examples

Example of Inlining a Function During Code Generation..... 355

References

InliningThreshold (📖 TargetLink Model Element Reference)
TargetLink Main Dialog Block (📖 TargetLink Model Element Reference)

How to Inline Specific Functions During Code Generation

Objective

TargetLink lets you eliminate function call overhead by inlining functions. Instead of calling the function from the code generated for the calling subsystem, the function call is replaced by the code of the function itself.

Method

To inline a specific function with the Code Generator

1. Open your DD project file and search for the function class you want to modify.
2. Set the **Optimization** property to `CG_FORCE_INLINE`. For a description on more inlining options, refer to Basics on Function Inlining on page 350.
3. Save the DD project file.

Result

You inlined a specific function with the Code Generator. The code of the function body is inlined.

Related topics

Basics

Basics on Function Inlining..... 350

How to Flag a Function for Compiler Inlining

Restriction

By default, compiler inlining works only for the `Generic C99` code generation target and Target Optimization Modules (TOM). You can also support arbitrary platforms as described in the *Generating TargetLink Code and ASAP2 Files for an Arbitrary Platform* application note, which you can download from the TargetLink Support Center.

In TargetLink, C compiler inlining is restricted to functions with static storage duration.

Precondition

The inline keyword must be specified as a `ddObj` XML element in the respective `TargetConfig.xml` file:

```
<ddObj name="InlineKeywords" Kind="GenericObject">
   <ddProperty Name="DeclarationKeyword" Type="String">__inline</ddProperty>
   <ddProperty Name="DefinitionKeyword" Type="String">__inline</ddProperty>
</ddObj>
```

Note

TargetLink preserves whitespaces and line breaks within the `ddProperty` XML elements. This lets you control whether the keyword is placed in the same line as the definition or declaration of the function.

Method

To flag a function for compiler inlining

1. Open your Data Dictionary project file (`*.dd`) and search for the function class you want to modify.
2. Set the **CompilerInline** property to **on**.
3. Save the Data Dictionary project file (`*.dd`).

Result

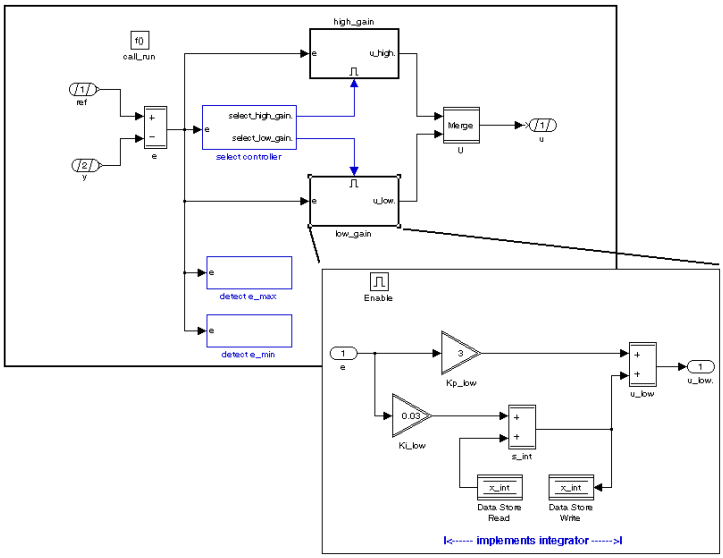
You flagged a function to be generated with a compiler keyword for compiler inlining. For the **Generic C99** target, its code looks like this:

```
inline static void MyInlineFunction(void)
{
   /* TargetLink outport: Subsystem/Subsystem/OutPort
      # combined # Gain: Subsystem/Subsystem/Gain */
   Sa2_OutPort = Sa2_InPort;
}
```

Example of Inlining a Function During Code Generation

ENABLSYS model

Suppose you work with the ENABLSYS model contained in TargetLink's demo library. This demo model contains the two enabled subsystems high_gain and low_gain.



Code without function inlining

If you set the Inlining threshold on the Advanced page of the TargetLink Main Dialog to 0 and generate code for the model, the code segment that contains the function calls for the two enabled subsystems will look like this:

```
static void Sa6_high_gain(void);
static void Sa7_low_gain(void);
...
if (!(Sa8_select_low_gain)) {
   /* call of function: controller/controller/high_gain */
   Sa6_high_gain();
}
...
if (Sa8_select_low_gain) {
   /* call of function: controller/controller/low_gain */
   Sa7_low_gain();
}
```

The generated code for the two functions representing the subsystems is similar. The code for the **low_gain** subsystem will look like this:

```
static void Sa7_low_gain(void)
{
   /* SLLocal: Default storage class for local variables | Width: 16 */
   Int16 Sa7_s_int /* LSB: 2^-7 OFF:  0 MIN/MAX:  -256 .. 255.9921875 */;

   /* Sum: controller/controller/low_gain/s_int
      # combined # Gain: controller/controller/low_gain/Ki_low
      # combined # Data store read: controller/controller/low_gain/Data Store Read */
   Sa7_s_int = (Int16) (((UInt16) (Int16) (((Int16) ((((Int32) Sa2_e) * 31457) >> 15)) >> 8)) +
    ((UInt16) x_int));

   /* update of variable(s) associated with controller/controller/low_gain/u_low
      # combined # Sum: controller/controller/low_gain/u_low
      # combined # Gain: controller/controller/low_gain/Kp_low */
   u = (Int16) (((UInt16) (Int16) ((((Int32) Sa2_e) * 3) >> 3)) + ((UInt16) Sa7_s_int));

   /* Data store write: controller/controller/low_gain/Data Store Write */
   x_int = Sa7_s_int;
}
```

Code with function inlining

If you set the inlining threshold on the Advanced page of the TargetLink Main Dialog to a large enough value, for example,**5**, before you generate code, the code looks like this:

```
if (!(Sa8_select_low_gain)) {
...
}
...
if (Sa8_select_low_gain) {
   /* SLLocal: Default storage class for local variables | Width: 16 */
   Int16 Sa7_s_int /* LSB: 2^-7 OFF:  0 MIN/MAX:  -256 .. 255.9921875 */;

   /* Sum: controller/controller/low_gain/s_int
      # combined # Gain: controller/controller/low_gain/Ki_low
      # combined # Data store read: controller/controller/low_gain/Data Store Read */
   Sa7_s_int = (Int16) (((UInt16) (Int16) (((Int16) ((((Int32) Sa2_e) * 31457) >> 15)) >> 8)) +
    ((UInt16) x_int));

   /* update of variable(s) associated with controller/controller/low_gain/u_low
      # combined # Sum: controller/controller/low_gain/u_low
      # combined # Gain: controller/controller/low_gain/Kp_low */
   u = (Int16) (((UInt16) (Int16) ((((Int32) Sa2_e) * 3) >> 3)) + ((UInt16) Sa7_s_int));

   /* Data store write: controller/controller/low_gain/Data Store Write */
   x_int = Sa7_s_int;
}
```

The Code Generator has inlined both functions, which reduces function-call overhead and improves code performance.

Related topics

References

TargetLink Main Dialog Block (📖 TargetLink Model Element Reference)

Embedding External Code

Introduction

TargetLink offers various methods for embedding external code.

Where to go from here

Information in this section

Introduction 358
Embedding External Functions in a TargetLink Subsystem 360
Embedding External Functions That Use Column-Major Order 369
Including External Code via the Data Dictionary 376

Introduction

Overview of Methods for Embedding External Code

Introduction

TargetLink offers various methods for embedding external code. The following table gives you a brief overview of the possible use cases. Use the table to find the most suitable use case and refer to the documentation for more detailed information.

Method	Use Case	Description	Refer to
Custom Code block	Embedding external code as statements or functions.	Insert the external code directly in a **Custom Code** block. Use a **Custom Code** (type I) block to create your own simulation block. If the used external code contains custom definitions, user-supplied source code files can be added via an **Addfile** block or linked via DD **Module** objects. In addition, a **Custom Code** (type II) block can be used to prepare unsupported Simulink blocks.	▪ How to Create User-Defined Simulation Blocks via Custom Code (Type I) Block (📖 TargetLink Preparation and Simulation Guide) ▪ How to Prepare Unsupported Simulink Blocks via Custom Code (Type II) Blocks (📖 TargetLink Preparation and Simulation Guide) ▪ Example of Bypassing Target-Specific Code for MIL/SIL Simulation By Using a Custom Code Block (📖 TargetLink Preparation and Simulation Guide)

Method	Use Case	Description	Refer to
Function block	Embedding external code for a subsystem.	Insert the external code for subsystem via Function blocks with its step function class property Storage set to `extern`. In SIL simulation and PIL simulation, the subsystem is replaced by the external code. In MIL simulation, the modeled subsystem is used. To model the subsystem without block restrictions, use the TL_BlackBox mask type for its contents.	▪ Embedding External Functions in a TargetLink Subsystem on page 360 ▪ Example of Bypassing Target-Specific Code for SIL Simulation When Embedding External Functions on page 365
Look-Up Table block	Embedding customer-specific functions for look-up tables.	Insert customer-specific functions for look-up tables.	Basics on Using Custom Look-Up Functions (📖 TargetLink Preparation and Simulation Guide)
MATLAB Function block	Embedding external code via MATLAB Function blocks.	Insert the external code via MATLAB Function blocks. In SIL simulation and PIL simulation, the external code should be linked via an Addfile block or DD Module objects. In MIL simulation, the code from the MATLAB local function is used.	Basics on Embedding External Code in a MATLAB® Function (📖 TargetLink Code Generation Guide for MATLAB® Code in Simulink® Models)
DD Module object	Embedding external code files without using an Addfile block.	Link the external code files via DD Module objects with DD Responsibility set to `External`. The external code is linked and compiled for SIL and PIL simulation.	▪ How to Include External Code Files via DD Modules on page 379 ▪ How to Build and Link External Modules via File Specifications on page 378
Stateflow® diagram	Embedding external code in Stateflow via scripting mechanism.	Embed external code directly in Stateflow diagrams. To call external functions, the function interface can be provided via a scripting mechanism.	▪ Using External Code in Stateflow via a Scripting Mechanism (Customer-Specific Functions) (📖 TargetLink Preparation and Simulation Guide) ▪ Using External Code in Stateflow via Extern C Stateflow Symbols (📖 TargetLink Preparation and Simulation Guide)
	Embedding external code in Stateflow via external C Stateflow symbols.	By default, every symbol that is used in Stateflow has to be defined in Stateflow itself. If you want to use external C Stateflow symbols, activate the EnableExternCStateflowSymbols Code Generator option.	Using External Code in Stateflow via Extern C Stateflow Symbols (📖 TargetLink Preparation and Simulation Guide)
	Embedding external code via graphical functions.	Embed external code via graphical functions and DD Module objects with DD Responsibility property set to `External`.	Basics on Using Graphical Functions (📖 TargetLink Preparation and Simulation Guide)

Demo model

The following demo model gives you a first impression of embedding external code (not all methods mentioned above are included in the demo model):

- EMBEDDING_EXTERNAL_CODE (📖 TargetLink Demo Models)

Embedding External Functions in a TargetLink Subsystem

Integrating C functions into production code

Embedding external functions is one more powerful mechanism to integrate C functions into the C code generated by TargetLink.

Where to go from here

Information in this section

How to Embed External Functions in a TargetLink Subsystem..... 360

Example of Bypassing Target-Specific Code for SIL Simulation When Embedding External Functions..... 365

Details on Controlling the Generation of Initialization, Termination and Restart Functions for External Step Functions by Using Function Classes..... 368

How to Embed External Functions in a TargetLink Subsystem

Objective

To call your own existing C function in the generated code, you must know how to embed external functions in a TargetLink subsystem.

Typical use case

Suppose your target ECU is equipped with special hardware features, for example, a unit for PWM generation or a hardware timer that can be accessed by external C functions. To allow offline simulations, your controller model contains a rather complicated implementation of the hardware unit by basic blocks. In this case you do not want TargetLink to generate C code for the model of the hardware function, which is needed only for offline simulations. Instead you want TargetLink to call your own existing C function in the generated code. You can achieve this behavior by embedding an external function. Make sure that the interface for the communication between your custom C function and TargetLink is defined unambiguously.

Representation of embedded external functions

An embedded external function is represented by a subsystem that contains a TargetLink Function block with its step function class property Storage set to `extern`.

Note

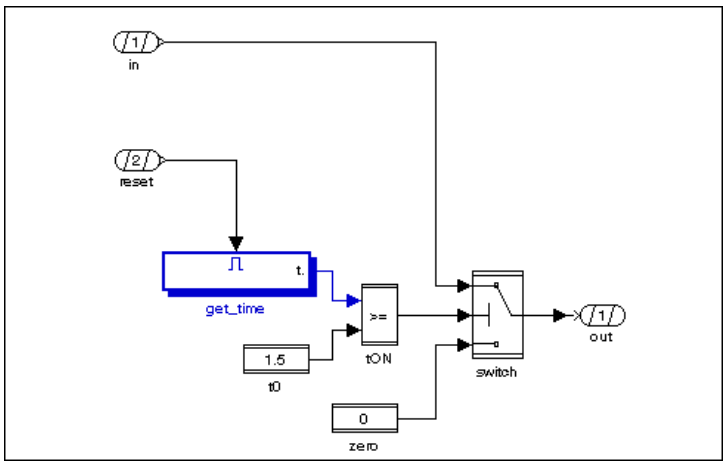
Function call triggers are not supported as inports for an external system.

Example

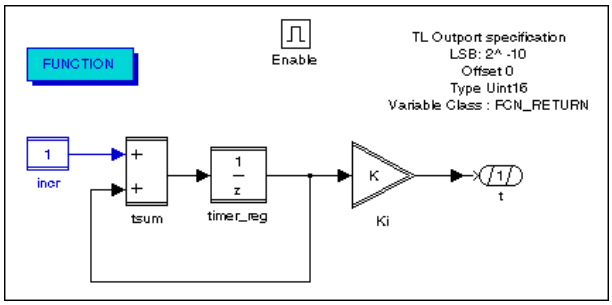
Suppose your ECU software uses the external `get_time()` function, which returns the current time. The function prototype is defined as `unsigned int get_time(void);` The time resolution is 2^{-10}. The counter is reset by reinitializing its internal state via the `reset_timer` function. This reset function has the prototype `void reset_timer(void);` Both functions are contained in the `get_timer.c` module. The corresponding header file is `get_timer.h`.

In your controller, you implemented the `get_time()` function with basic blocks in the `get_time` subsystem. In the generated code, you want to apply your own `get_time()` function from the source file `get_time.c`.

The model might look like this:



The get_time subsystem is shown below:



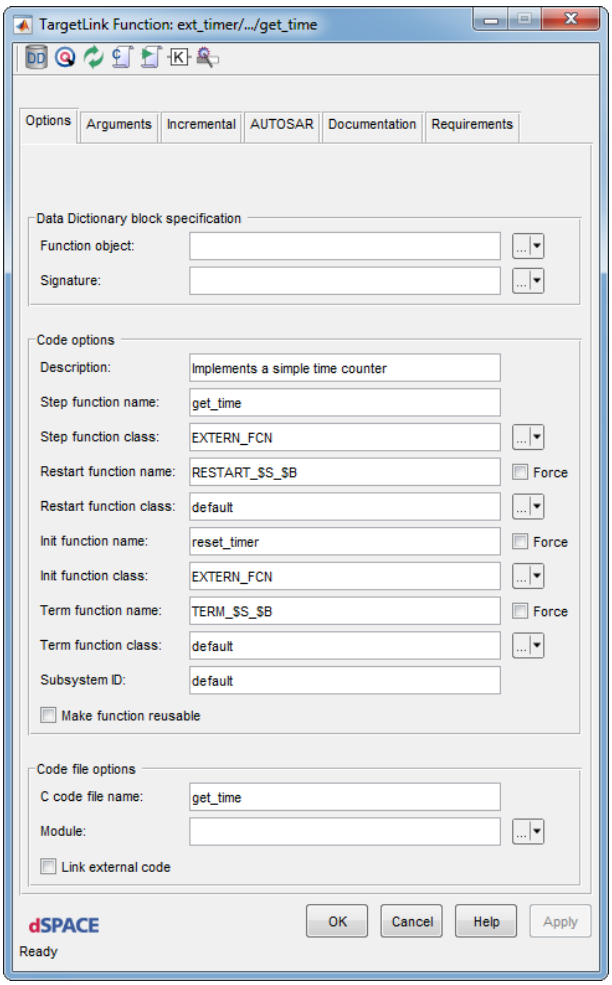
If you want to implement the model according to the above example, carry out the following instructions.

Method

To embed an external function in a TargetLink subsystem

1 Open the subsystem that represents the external function.

2 Place a TargetLink Function block in the subsystem and open it to specify the settings.



3 Enter the Step function name `get_time` and the Initialization function name `reset_timer`, and select the EXTERN_FCN function class for both functions.

4 The Restart and Term functions do not need to be generated, so the entries can be left unchanged.

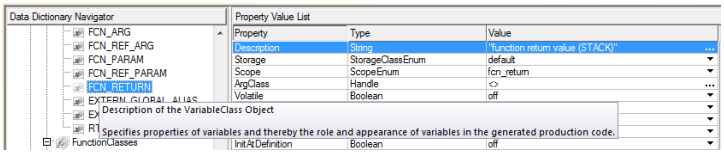
5 In the Code file options group box, enter the name of the code file (without extension) that you want to link to the generated code.

6 Insert InPorts and/or OutPorts to define the function interface in the same way that you define the ingoing and outgoing signals for the Simulink simulation of the subsystem.

The example requires an enhanced TargetLink OutPort block containing information on the function parameter to be generated.

7 Configure the **InPort** and **OutPort** blocks in the function subsystem so that the data type and scaling factors match the prototype of your function.

8 Select the **FCN_RETURN** variable class for the output signal of the example.



Note

- If you use several outputs, you have to select the **FCN_REF_PARAM** function class for all other output signals. You usually also have to configure the **InPort** blocks of your function, which is not shown in the example.
- You should use fixed variable names that define the function interface, especially if you use global variables. As an alternative, you can use the **FCN_PARAM** variable class for the input signals.
- If the ranges of the output signals of your function are known, specify them in the constrained range edit fields of the Output page in the **OutPort** Block dialog. This helps the Code Generator optimize the code.

Result

With these settings, TargetLink generates the following code. Keep in mind that the function bodies of `get_time()` and `reset_timer()` are not generated by TargetLink, but linked from the `get_time.c` file:

```
void Subsystem(void)
{
   /* SLStaticLocalInit: Default storage class for static local variables with initvalue | Width: 32  */
   static Float32 S02_t = 0.F;

   /* SLStaticLocalInit: Default storage class for static local variables with initvalue | Width: N.A. */
   static struct tag_GIBFS_0__tp GIBFS_0_ = {
      0 /* S02_ROWD: LSB: 2^0 OFF:  0 MIN/MAX:  0 .. 1 */,
      0 /* S02_RSWE: LSB: 2^0 OFF:  0 MIN/MAX:  0 .. 1 */
   };

   /* Subsystem/get_time/Enable: Enable condition */
   if (S01_reset > 0) {
      if (!(GIBFS_0_.S02_RSWE)) {
         /* initialization of subsystem: Subsystem/get_time */
         reset_timer();

         /* set system state to 'enabled' */
         GIBFS_0_.S02_RSWE = 1;

         /* outport reset necessary if system will be disabled */
         GIBFS_0_.S02_ROWD = 1;
      }

      /* call of function: Subsystem/get_time */
      S02_t = get_time();
   }
   else {
      /* Subsystem/get_time: Omitted comparison with constant. */
      if (GIBFS_0_.S02_ROWD) {
         /* outport reset */
         S02_t = 0.F;

         /* outport reset not necessary in the next step */
         GIBFS_0_.S02_ROWD = 0;

         /* set system state to 'disabled': Subsystem/get_time */
         GIBFS_0_.S02_RSWE = 0;
      }
   }

   /* Switch: Subsystem/Switch
      Subsystem/Switch: Omitted comparison with constant.
      # combined # Relational: Subsystem/tON */
      if (S02_t >= 1.5F) {
         /* # combined # TargetLink outport: Subsystem/out */
         S01_out = S01_in;
      }
      else {
         /* # combined # TargetLink outport: Subsystem/out */
         S01_out = 0.F;
      }
}
```

Related topics

Basics

Basics on Using Subsystems 226

HowTos

How to Implement Subsystems as Functions 228
How to Specify Function Interfaces via Function Parameters 242

References

Function Block (📖 TargetLink Model Element Reference)
Subsystem Block (📖 TargetLink Model Element Reference)

Example of Bypassing Target-Specific Code for SIL Simulation When Embedding External Functions

Introduction

Suppose you have external code that you want to embed as an external function. However, it contains target-specific code parts, which cannot be handled by the host compiler, so you have to bypass them to enable software-in-the-loop (SIL) simulations.

In this example we take the function `my_hardware_specific_function()` to be target-specific and `my_hardware_specific_function_TLSIM()` to be used for SIL simulation. Consider their function definitions:

Definition Containing Target-Specific Code	Definition of Function for SIL Simulation

`void my_hardware_specific_function(Int16 in, Int16* out)`
`{`
`  /* implementation for target */`
`  XXXTargetSpecificCodeXXX`
`}`
	`void my_hardware_specific_function_TLSIM(Int16 in, Int16* out)`
	`{`
	`  /* implementation for simulation */`
	`}`

Conditional compilation

To bypass the target-specific parts of your external code, you can prepare your code for conditional compilation using preprocessor directives. TargetLink provides the following predefined macro you can use:

Preprocessor Macro	Simulation Mode	Description
_DSHOST	SIL	TargetLink defines this macro when instructed to build a SIL simulation application for a TargetLink subsystem.

Method

To bypass target-specific code for SIL simulation of an externally embedded function

1 Prepare your external code for conditional compilation using the `#ifdef _DSHOST` directive:

```
void my_external_function(Int16 in, Int16* out)
{
    #ifdef _DSHOST
    my_hardware_specific_function_TLSIM(in, out);
    #else
    my_hardware_specific_function(in, out);
    #endif
}

#ifdef _DSHOST
void my_hardware_specific_function_TLSIM(Int16 in, Int16* out)
{
    /* implementation for simulation */
}
#else
void my_hardware_specific_function(Int16 in, Int16* out)

{
    /* implementation for target */
    XXXTargetSpecificCodeXXX
}
#endif
```

This ensures that your target-specific external code is compiled only for PIL, whereas for SIL the generic external code is taken.

2 In the Data Dictionary Manager, create a DD **Module** object called `<MyExternalCode>` and specify it as follows:

DD Object	Property	Value
`<MyExternalCode>/ModuleInfo`	Responsibility	`External`
	NameTemplate	`<MyExternalCode>`

3 Add two DD **FileSpecification** objects, specified as follows:

DD Object	Property	Value
…/FileSpecifications/FileSpecification	EmitFile	off
	FileKind	SourceFile
	FilePath	<PathToSourceFile>
…/FileSpecifications/FileSpecification(#2)	EmitFile	off
	FileKind	HeaderFile
	FilePath	<PathToHeaderFile>
	CompileAndLinkCorrespondingSourceFile	on

4 Specify your external function in the model via a **Function** block placed in an atomic subsystem.

On the **Options** page of the **Function** block, make at least the following settings:

Group box	Property	Value
Code options	Step function name	`<MyExternalFunction>`
	Step function class	`EXTERN_FCN`
Code file options	Module	`<MyExternalCode>`
	Link external code	Selected

5 On the **Output** page of the subsystem's **InPort/OutPort** block dialog, specify the function's interface:

Block	Property	Value
InPort	Name	`in`
	Class	`FCN_ARG`
OutPort	Name	`out`
	Class	`FCN_REF_ARG`

6 Generate Code.

Result

You have prepared your external code using preprocessor directives, allowing to conditionally compile your code for SIL/PIL.

Related topics

Basics

Customizing the Appearance of Functions via Subsystems and Function Classes..... 230
Defining the Function Interface..... 241
Details on Handling Objects in Preprocessor Directives..... 619
Embedding External Functions in a TargetLink Subsystem..... 360
Mapping Code Generation Units to Code (Module Ownership)..... 382
Overview of Methods for Embedding External Code..... 358
Specifying Conditional Control Flows..... 616

HowTos

How to Build and Link External Modules via File Specifications..... 378
How to Include External Code Files via DD Modules..... 379

Examples

Example of Bypassing Target-Specific Code for MIL/SIL Simulation By Using a Custom Code Block (📖 TargetLink Preparation and Simulation Guide)

References

Function Block (📖 TargetLink Model Element Reference)

Details on Controlling the Generation of Initialization, Termination and Restart Functions for External Step Functions by Using Function Classes

Supported functions

For an embedded external function, you can control the generation of the following functions via function classes:

- Initialization
- Termination
- Restart

Note

You can even force an empty function to be generated for these functions, via the Force checkbox in the Function block dialog. This lets you ensure that a specific system interface is always implemented.

Class combinations

The table below shows possible combinations of:

- Step function class
- Initialization function class
- Termination function class
- Restart function class.

The term *default* always refers to the **default** function class setting. The terms *static* and *external* refer to function classes with the corresponding `Storage` class attribute.

Function	Class	Result for ... Non-External Step Function	External Step Function
Step	–	Always generated.	Call and external declaration are generated. The external function must be available.
Initialization	default	Generated if required for the C code of the model.	Call and declaration are not generated.
	static		Not allowed.
	external	Call and external declaration are generated if required for the C code of the model.	
Termination	default	Generated if the Function block resides in the TargetLink subsystem and the function is required.	Call and declaration are not generated.
	static		Not allowed.
	external	Call and external declaration are generated if the Function block resides in the TargetLink subsystem. The external function must be available.	Call and external declaration are generated. The external function must be available.
Restart	default	Generated if the Function block resides in the TargetLink subsystem and the function is required.	Call and declaration are not generated.
	static		Not allowed.
	external	Call and external declaration are generated if the Function block resides in the TargetLink subsystem. The external function must be available.	Call and external declaration are generated. The external function must be available.

Related topics

HowTos

How to Embed External Functions in a TargetLink Subsystem..... 360
How to Specify Subsystem Interfaces..... 236

Examples

Example of Initializing Variables via Restart Functions..... 120

References

Function Block (📖 TargetLink Model Element Reference)

Embedding External Functions That Use Column-Major Order

Where to go from here

Information in this section

Basics on Adapting Column-Major Functions for Use With TargetLink..... 369

How to Adapt a Column-Major Legacy Function Using C Code in a Custom Code Template..... 371

How to Adapt a Column-Major Legacy Function via Permute Dimensions Blocks in the Model..... 373

Basics on Adapting Column-Major Functions for Use With TargetLink

FORTRAN external code

External code stemming from FORTRAN that consumes matrices has to be prepared for use with TargetLink. This is because the matrices generated by TargetLink will be stored in memory in row-major order, as demanded by the ANSI C/ISO C standard, while FORTRAN demands that matrices are stored in memory in column-major order. For details, refer to Principles of Row-Major and Column-Major Order (Storage of Matrices in Memory) (📖 TargetLink Preparation and Simulation Guide).

Legacy function

Because your legacy function call will be part of the production code, the matrices are created by TargetLink and are therefore stored in row-major order.

You therefore must transform each matrix into column-major order before calling the function.

C Code view From a C code point of view, the transformation looks as follows:

```
/* Input is A[ROWS][COLS].
* Define temporary storage for transposed matrix AT
 */
int AT[COLS][ROWS];
for (i=0;i<ROWS;i++)
    for(j=0;j<COLS;j++)
        AT[j][i] = A[i][j];
```

There are two methods for transforming a matrix stored in column-major order to a matrix stored in row-major order. Both methods are based on the fact that the column-major representation of matrix A is identical to the row-major representation of the transposed matrix A^T:

- Adapting to column-major functions via C code
- Adapting to column-major functions on the model level

Adapting to column-major functions via C code

Suppose, for example, that you converted the block that represents your column-major order legacy function to a **Custom Code** (type II) block. In the generated code, TargetLink provides the input in row-major format, so the custom code template must contain code to convert the input to column-major format before calling the original function.

For details, refer to How to Adapt a Column-Major Legacy Function Using C Code in a Custom Code Template on page 371.

Adapting to column-major functions on the model level

Suppose, for example, that you explicitly modeled the transformation of the matrix model wise. This is done by using two Permute Dimensions blocks: one for production code, and one for MIL simulation mode. While only the model part containing the column-major function is converted into a **Custom Code** (type II) block to hide it from TargetLink, the whole modeling construct is placed within a subsystem to be used as the substitution block in your model.

For details, refer to How to Adapt a Column-Major Legacy Function via Permute Dimensions Blocks in the Model on page 373.

Related topics

Basics

Details on Masking Simulink Blocks with TargetLink Custom Code (Type II) Blocks (📖 TargetLink Preparation and Simulation Guide)
Introduction to Matrix Signals (📖 TargetLink Preparation and Simulation Guide)

HowTos

How to Adapt a Column-Major Legacy Function Using C Code in a Custom Code Template..... 371
How to Adapt a Column-Major Legacy Function via Permute Dimensions Blocks in the Model..... 373

Examples

CUSTOM_ENHANCEMENT (📖 TargetLink Demo Models)

How to Adapt a Column-Major Legacy Function Using C Code in a Custom Code Template

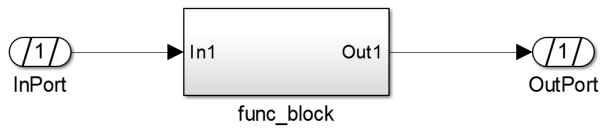
Objective

To adapt a column-major order legacy function to row-major order via C-Code provided in the Custom Code Template of a **Custom Code** (type II) block.

The C code provided in the custom code template calls the column-major function `func_col_order`. A temporary matrix `AT` is allocated to hold the transpose of input matrix `A`. The two nested loops write the transpose of `A` to `AT`. The function is then called with `AT` as its argument.

Precondition

- You read Principles of Row-Major and Column-Major Order (Storage of Matrices in Memory) (📖 TargetLink Preparation and Simulation Guide).
- You read Basics on Adapting Column-Major Functions for Use With TargetLink on page 369.
- You modeled the functionality of your legacy function as a single block (or subsystem) called **func_block** to enable MIL simulation:



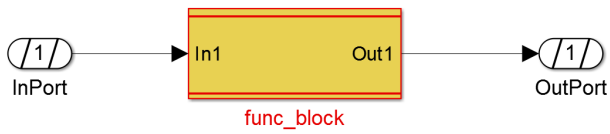
Restriction

> **Note**
>
> For matrix signals as inputs or outputs, TargetLink only supports the Custom Code (type II) block.

Method

To adapt a column-major legacy function using C code in a custom code template

1. Enhance func_block to a Custom Code (type II) block:



2. Open func_block's block dialog and specify the interface of your legacy function as required.
3. On the Code & Logging page, select Edit Custom Code to open the block's custom code template.
4. In the custom code template's fxp_output section, provide the C code that adapts the column-major order layout to the row-major layout. Make sure to also place the call to `func_col_order`.

   You can adapt the following pseudo code to your needs:

```
##############################################################################
# Enter output vector evaluation code for production code simulation below
##############################################################################
/* fxp_output_begin */
```

```
/* Input is A[ROWS][COLS].
* Define temporary storage for transposed matrix AT
 */
int AT[COLS][ROWS];
for (i=0;i<ROWS;i++)
    for(j=0;j<COLS;j++)
        AT[j][i] = A[i][j];
```

```
/* In C a function can only expect column-major order if the matrix is
 * passed as a pointer to linearized memory. */
y1 = func_col_major(&AT[0][0],ROWS,COLS);
```

```
/* fxp_output_end */
```

5. Save the custom code template and click OK to close func_block's block dialog.

Result

You adapted a column-major legacy function via C code in a custom code template.

The C code provided in the custom code template calls the column-major function `func_col_order`. A temporary matrix `AT` is allocated to hold the transpose of input matrix `A`. The two nested loops write the transpose of `A` to `AT`. The function is then called with `AT` as its argument.

Note

If the function's output is a matrix in column-major order, it also needs to be transposed to row-major.

Related topics

Basics

Basics on Adapting Column-Major Functions for Use With TargetLink..... 369
Details on Masking Simulink Blocks with TargetLink Custom Code (Type II) Blocks
(📖 TargetLink Preparation and Simulation Guide)
Principles of Row-Major and Column-Major Order (Storage of Matrices in Memory)
(📖 TargetLink Preparation and Simulation Guide)

Examples

CUSTOM_ENHANCEMENT (📖 TargetLink Demo Models)

References

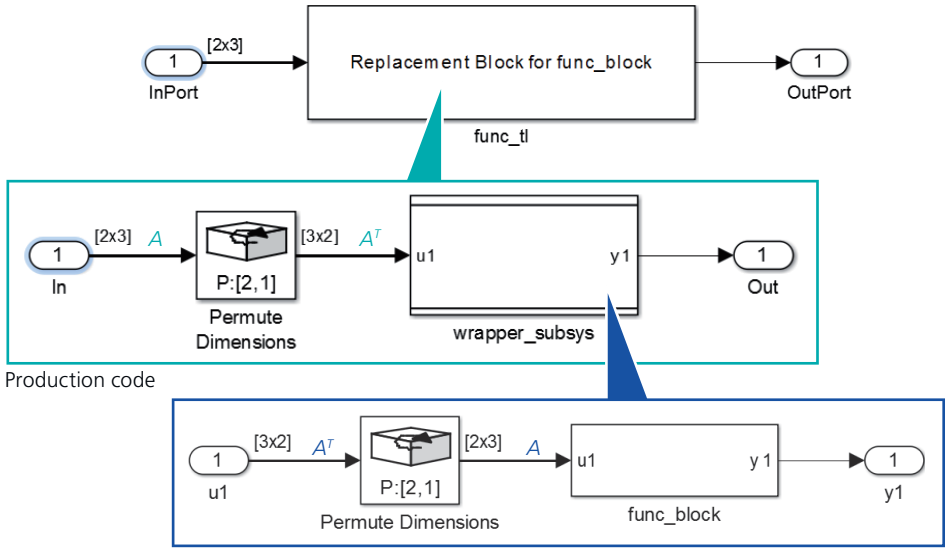
Common TargetLink Context Menu Options (📖 TargetLink Model Element Reference)

How to Adapt a Column-Major Legacy Function via Permute Dimensions Blocks in the Model

Objective

To adapt a column-major order legacy function to row-major order via Permute Dimensions blocks before a **Custom Code** (type II) block and under its mask.

Modeling overview



Model Component	Description
func_tl	A subsystem used as a replacement block for your legacy function called func_block. In production code it is implemented as the following external function that is specified in the Custom Code (type II) block called wrapper_subsys.

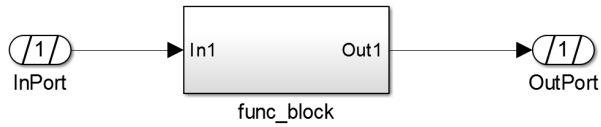
wrapper_subsys	A masked subsystem that uses TargetLink's Custom Code (type II) block.
	The Permute Dimensions block placed on the same level as wrapper_subsys transposes matrix A to A^T: i.e., from row-major order to column-major order. It is A^T that serves as input u1 of your custom code.
	The production code for wrapper_subsys is implemented as specified in the custom code template:
	`/* fxp_output_begin */`
	`y1 = func_col_major(&u1[0][0], IN_1_DIM_2_WIDTH, IN_1_DIM_1_WIDTH);`
	`/* fxp_output_end */`
	The Permute Dimensions block placed within wrapper_subsys is relevant only for MIL simulation mode and is ignored during code generation. It makes sure that in MIL simulation mode, A is used as the input.
	Note that the width macros reflect that A^T serves as the input to the function. `IN_1_DIM_2_WIDTH` evaluates to the number of A's rows or of A^T's columns, while `IN_1_DIM_1_WIDTH` evaluates to the number of A's columns or of A^T's rows.

func_block	Contains the functionality of one or more unsupported Simulink block(s) that represent your legacy function's functionality in MIL simulation mode.

Precondition

- You read Principles of Row-Major and Column-Major Order (Storage of Matrices in Memory) (📖 TargetLink Preparation and Simulation Guide).
- You read Basics on Adapting Column-Major Functions for Use With TargetLink on page 369.

- You modeled the functionality of your legacy function as a single block (or subsystem) called func_block to enable MIL simulation:



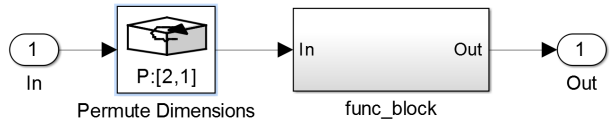
Restriction

> **Note**
>
> For matrix signals as inputs or outputs, TargetLink only supports the Custom Code (type II) block.

Method

To adapt a column-major legacy function via Permute Dimensions blocks in the model

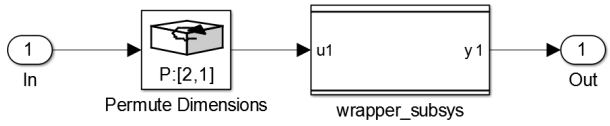
1. Place a Permute Dimensions block directly before func_block and connect it appropriately:



2. Select the Permute Dimensions block and func_block and create a subsystem from the selection.

   Rename the subsystem as `wrapper_subsys`.

3. Enhance wrapper_subsys to a Custom Code (type II) block:



4. Open wrapper_subsys's block dialog and specify the interface of your legacy function as required.
5. On the Code & Logging page, select Edit Custom Code to open the block's custom code template.
6. In the custom code template's fxp_output section, call your legacy function.

   You can adapt the following code to your needs:

```
##############################################################################
# Enter output vector evaluation code for production code simulation below
##############################################################################
/* fxp_output_begin */
y1 = func_col_major(&u1[0][0], IN_1_DIM_2_WIDTH, IN_1_DIM_1_WIDTH  );
/* fxp_output_end */
```

7. Save the custom code template and click OK to close func_block's block dialog.

8 Place a Permute Dimensions block directly before **wrapper_subsys** and connect it appropriately.

9 Select the Permute Dimensions block and **wrapper_subsys** and create a subsystem from the selection.

Rename the subsystem as `func_tl`.

Result

You adapted a column-major legacy function via Permute Dimensions blocks in a model:

- The Permute Dimensions block before the **Custom Code** (type II) block is required to transpose matrix A to A^T: i.e., from row-major order to column-major order. It is A^T that serves as input **u1** of your custom code.
- The Permute Dimensions block under the mask of the **Custom Code** (type II) block is required for MIL simulation only. It is ignored during code generation.

Related topics

Basics

Basics on Adapting Column-Major Functions for Use With TargetLink..... 369
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)
Principles of Row-Major and Column-Major Order (Storage of Matrices in Memory) (📖 TargetLink Preparation and Simulation Guide)

Examples

CUSTOM_ENHANCEMENT (📖 TargetLink Demo Models)

References

Common TargetLink Context Menu Options (📖 TargetLink Model Element Reference)

Including External Code via the Data Dictionary

Where to go from here

Information in this section

Basics on Including External Code via the Data Dictionary..... 377
How to Build and Link External Modules via File Specifications..... 378
How to Include External Code Files via DD Modules..... 379

Basics on Including External Code via the Data Dictionary

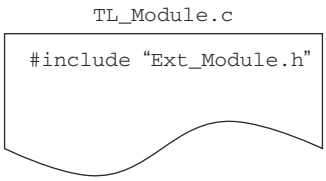
Introduction

TargetLink supports the integration of external code, which you must specify in the TargetLink Data Dictionary.

> **Note**
>
> External code files are not generated by TargetLink and are treated like a black box. Thus, the external function algorithm cannot be optimized. Optimization is possible only at the function interfaces.

Including external code files into modules generated by TargetLink

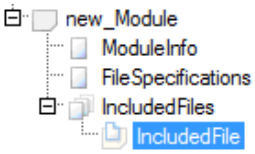
A module generated by TargetLink includes an external code file.



You can specify this `#include` via DD FileSpecification objects in the Data Dictionary's following subtree:

```
/Pool/Modules/[<ModuleGroup>/]*/<Module>/FileSpecifications
```

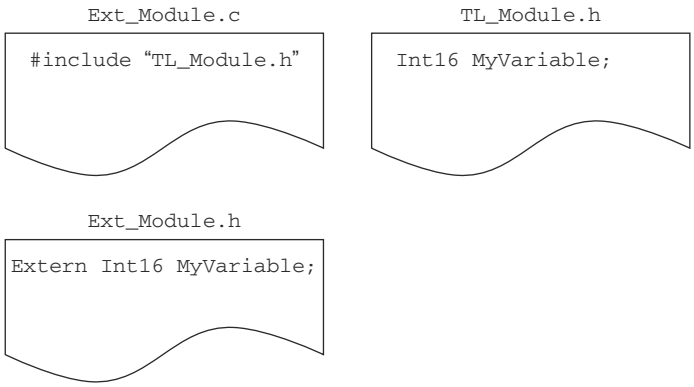
A separate IncludedFile object in the TargetLink Data Dictionary lets you specify the name of the file that is to be included.



Refer to How to Include External Code Files via DD Modules.

Generating a module to include in external code

You can generate code modules from the Data Dictionary that are independent of models. You can then include these modules in external code modules maintained by you.



How to Build and Link External Modules via File Specifications

Objective	By providing file specifications in DD Module objects, you can specify external modules to be compiled and linked during the build process.
Restriction	Linking external files via FileSpecification objects is not possible for building S-functions for Custom Code (type I) blocks.
Preconditions	An external header file must exist.
Optional information	Optionally, you can provide: ▪ An external source file ▪ Information about the path if the files are not in the working directory ▪ Information about the file extension
Method	**To build and link external modules via file specifications in DD module objects** **1** In the Data Dictionary Navigator, select the DD `/Pool/Modules` object. **2** Right-click Modules and select Create Module. **3** Change the DD Module object's name to match the name of your external code file. **4** Set the ModuleInfo.Responsibility property to `External`. **5** From the Module.FileSpecifications object's context menu, select Create FileSpecification. **6** Select the resulting FileSpecification object and set the CompileAndLinkCorrespondingSourceFile property to `on`.

7 Optionally, you can do the following:
   - Provide the path to the header/source file as the value of the **FilePath** property if the file is not in the working directory or on the path specified with the **IncludeFileSearchPath** or **SourceFileSearchPath** property of the DD `/Config/TargetLink` object.
   - Modify the **FileExtension** property if the user file does not comply with the usual file name extensions (`C/H`).
8 Use a code element contained in the external module in a code generation unit (CGU).

Result

You specified an external module that will be included in the build process.

How to Include External Code Files via DD Modules

Objective

To include external code files in generated production code without using an **Addfile** block.

Restriction

Linking external files via **FileSpecification** objects is not possible for building S-functions for **Custom Code** (type I) blocks.

Preconditions

You must have created at least the following two modules in the Data Dictionary:

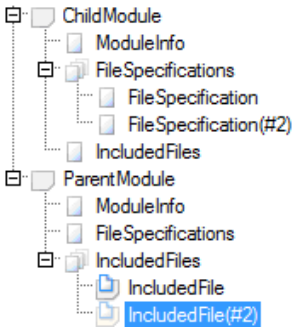
1. The module that represents the code file that is to be included (child module).
2. The module that represents the production code in which the code file is to be included (parent module).

Method

To include external code files via DD modules

1 Open the DD project file.

2 Specify the child module in the Pool area of the DD object tree as follows:
   - The **Responsibility** property (**ModuleInfo** object) must be set to **External**.
   - Create a **FileSpecification** object for each file type (`*.h` or `*.c`)



3 Specify the parent module in the **Pool** area of the DD object tree as follows:
   - Create an **IncludeFile** object for each code file that is to be included.
   - For its **ModuleRef** property, reference the child module.

4 Save the DD project file.

Result

During production code generation, the code file (child module) is included in the generated production code (parent module).

Related topics

Basics

Introduction to TargetLink's Custom Code Blocks (📖 TargetLink Preparation and Simulation Guide)

HowTos

How to Create Module Specifications in the Data Dictionary..... 391

References

Addfile Block (📖 TargetLink Model Element Reference)

Decomposing Models for Distributed Development

Introduction

Distributed function development teams are a *de facto* standard in today's software projects. There is always a tradeoff between keeping involved function developments and resources up to date, whereas it allows you to break down large work packages (decompose) or to appoint experts to specific development tasks. Large projects and large function development teams require access to many corporate resources, which is why you have to guarantee shared access to common repositories, such as the TargetLink Data Dictionary, for all team members. Shared repositories provide them with consistent and up-to-date information, thereby reducing conflicts.

The solution to the challenges of distributed development is model partitioning. TargetLink provides different ways to do this, e.g., 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. Finally, if you use component-based development with abstract interfaces, you can separate the overall functionality into smaller software modules that can be developed and tested independently.

Where to go from here

Information in this section

Mapping Code Generation Units to Code (Module Ownership)..... 382

DD-Based Code Generation..... 397

Decomposing Models Logically (Incremental Code Generation)..... 406

Decomposing Models into Model Parts (Model Referencing)..... 413

Defining Abstract Interfaces for Model-Components..... 447

Mapping Code Generation Units to Code (Module Ownership)

Where to go from here

Information in this section

Principles of Partitioning Models and Code 382

Basics on Modules and Module Ownership 388

How to Create Module Specifications in the Data Dictionary 391

How to Specify a Model-Based CGU as the Specific Owner of a DD Module Object 393

How to Specify a DD-Based CGU as the Specific Owner of a DD Module Object 395

Principles of Partitioning Models and Code

Introduction

Partitioning your model and code is closely connected to designing suitable parts and their interfaces.

Partitioning the model If you partition your model, it consists of different parts. These parts are usually connected according to Simulink semantics. Because they can be connected, they have interfaces.

When you partition your model, make sure that each part of the model is only loosely coupled with other parts. You should also make sure that none of the model parts scatter or tangle concerns.

Partitioning the code If you partition your model by using code generation units (CGUs), your code is partitioned into separate code modules. This is the case because the code of each CGU forms a root step function that is generated into an individual code module.

Note

Define the interface of each model-based CGU

The interface of model-based CGUs can have the following elements:

- The CGU name.
- Variables with global scope and a suitable data type.
- Parameters of a suitable data type.

If you partition your model by using CGUs, make sure the interface of each CGU is well-defined:

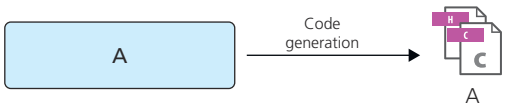
- Specify the interface variables, the data types, and the modules they belong to.
- Specify the CGU that is responsible for a module containing interface variables or data types.

This avoids problems caused by undefined responsibility during code generation.

Basic case with one CGU

Consider a model that contains a single model-based CGU ( ).

If you make no further specifications, TargetLink generates the production code of the CGU named **A** into a code module **A** ( ) in the file system. Refer to the following illustration:



Therefore, a mapping exists between model and code: The algorithms modeled in the CGU are mapped to the production code in the module in the file system.

Defining the interfaces

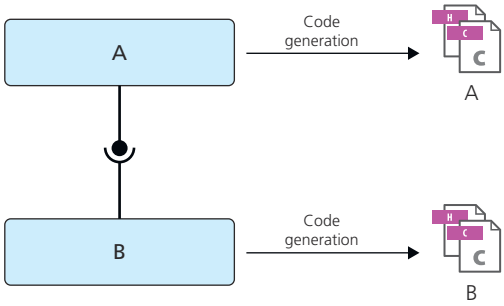
Note

Simplified explanations

To simplify the explanations in this topic, the following assumptions are made:

- Communication takes place via the global variable called `MyInterfaceVar`.
- The variable's data type is not explicitly considered.

The interface of a CGU ( ) is usually provided by model elements that result in the definition of interface variables with a global scope. Refer to the following illustration:



Without further specification, TargetLink generates a definition of `MyInterfaceVar` in the production code of both `A` and `B`:

CGU A	CGU B
Module A	**Module B**

`GLOBAL Int16 MyInterfaceVar;`
`void A(void)`
`{`
`  /* TargetLink outport: A/AOut */`
`  MyInterfaceVar = 1;`
`}`
	`GLOBAL Int16 MyInterfaceVar;`
	`void B(void)`
	`{`
	`  /* SLLocal: Default storage class for local variables | Width: 16 */`
	`  Int16 MyGain;`
	
	`  /* Gain: B/Gain */`
	`  MyGain = MyInterfaceVar;`
	`}`

In this case, the interface is not well defined, because the same variable is defined twice, once in each module.

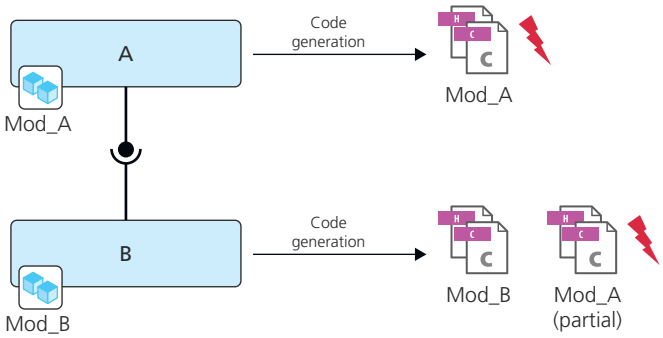
Note

Avoid duplicate variable definitions

To avoid duplicate variable definitions, specify which variable belongs to which module.

Specifying modules for cohesiveness

By making a module specification ( ) at each CGU, you can increase the cohesiveness of its code. In the following example, CGU `A` provides the interface variable `MyInterfaceVar`. The variable is defined in module `Mod_A`:



CGU A	CGU B	
Production Code	**Production Code**	
Module Mod_A	**Module Mod_B**	**Module Mod_A (Partial)**

`GLOBAL Int16 MyInterfaceVar;`
`void A(void)`
`{`
`  /* TargetLink outport: A/AOut */`
`  MyInterfaceVar = 1;`
`}`
	`void B(void)`
	`{`
	`  /* SLLocal: Default storage class for local variables | Width: 16 */`
	`  Int16 MyGain;`
	
	`  /* Gain: B/Gain */`
	`  MyGain = MyInterfaceVar;`
	`}`
		`GLOBAL Int16 MyInterfaceVar;`

The interface variable `MyInterfaceVar` is defined in code module `Mod_A` and used in code module `Mod_B`. Therefore, a partial code module `Mod_A` is generated during code generation for CGU `B`. The partial code module contains the definition of `MyInterfaceVar`.

In this case, the situation is unclear because you get two code modules in the file system that have the *same* name but *different* content. Because the conflicting modules are both production code, compilation errors might occur if the application results from the production code of CGU `A` *and* CGU `B`.

Note

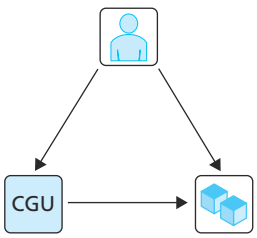
Specify module ownership as precisely as possible

To avoid generating code modules that share the same name but have different content, make sure that each module is owned by exactly one CGU. The CGU that owns a module is then responsible for generating the production code version of the module.

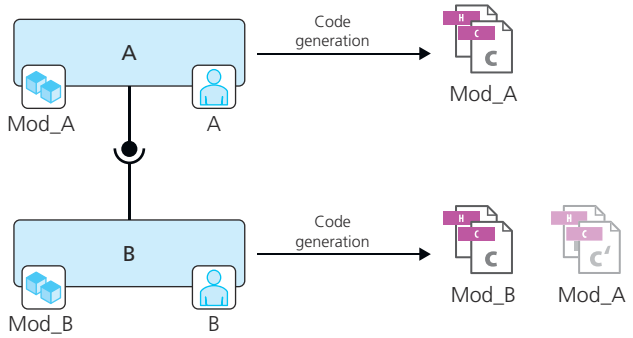
Specifying owners responsible for code generation

To solve the problem of code modules with identical names but different content, you can specify an owner ( ).

You specify the owner of a code module by establishing a connection between the module specification and the CGU that is responsible for generating the production code of the module:



Specifying the owner of each code module solves the problem of code modules with identical name but different content. Whenever a CGU needs code that is contained in a module owned by another CGU, TargetLink creates a partial stub code version ( ) of that module:



CGU A		CGU B	
Production Code	**Stub Code**	**Production Code**	**Stub Code**
Module Mod_A	**-**	**Module Mod_B**	**Module Mod_A (Partial)**

`GLOBAL Int16 MyInterfaceVar;`
`void A(void)`
`{`
`  /* TargetLink outport: A/AOut */`
`  MyInterfaceVar = 1;`
`}`
	-	`void B(void)`
		`{`
		`  /* SLLocal: Default storage class for local variables | Width: 16 */`
		`  Int16 MyGain;`
		
		`  /* Gain: B/Gain */`
		`  MyGain = MyInterfaceVar;`
		`}`
			`GLOBAL Int16 MyInterfaceVar;`

When you build an application that results from the code of both CGUs, the production code variant of `Mod_A` is used. When you build an application that results only from the code of CGU `B`, the stub code variant of `Mod_A` is used.

However, the code in module `Mod_B` still requires some of the code of module `Mod_A`. This means that both modules are coupled.

Note

Coupling lowers modularity

The tighter the coupling, the lower the modularity. To ensure loose coupling, you can further separate concerns by separately generating the code that is used by several modules into dedicated modules.

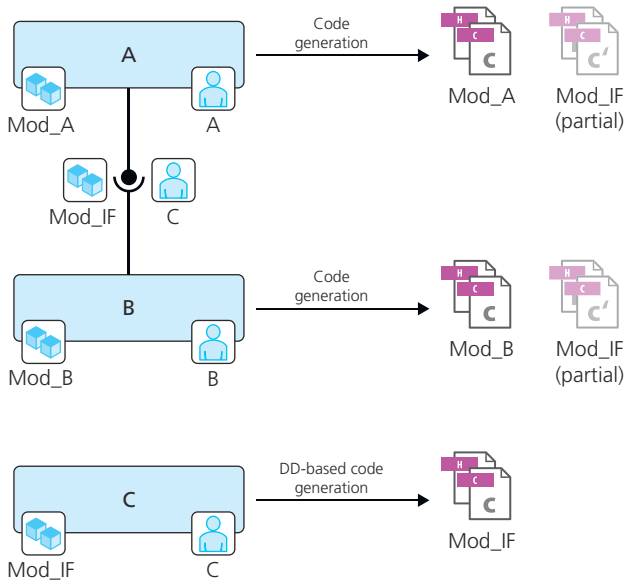
Separately generating code elements into dedicated modules

You can increase the modularity of your code if you generate the code that is used by different CGUs separately and into dedicated modules.

This is usually the case for the following code elements:

- Parameter variables
- Interface variables
- Data types

The following illustration shows how to generate the interface variable `MyInterfaceVar` into a dedicated code module called `Mod_IF` that is owned by a DD-based CGU called `C`:



CGU A		CGU B		CGU C	
Production Code	**Stub Code**	**Production Code**	**Stub Code**	**Production Code**	**Stub Code**
Module Mod_A	**Module Mod_IF**	**Module Mod_B**	**Module Mod_IF**	**Module Mod_IF**	**-**

`void A(void)`
`{`
`  /* TargetLink outport: A/AOut */`
`  MyInterfaceVar = 1;`
`}`
	`GLOBAL Int16 MyInterfaceVar;`	`void B(void)`
		`{`
		`  /* SLLocal: Default storage class for local variables | Width: 16 */`
		`  Int16 MyGain;`
		
		`  /* Gain: B/Gain */`
		`  MyGain = MyInterfaceVar;`
		`}`
			`GLOBAL Int16 MyInterfaceVar;`	`GLOBAL Int16 MyInterfaceVar;`	-

Modeling modular components

If you partition your model, it is best to partition it into separate modular components. Therefore, make sure the following criteria are met:

Criterium	Description
Cohesiveness	The component keeps its program elements together.
Loose coupling	The component's concerns neither are scattered nor tangled. Coupling components potentially lowers cohesion, unless the interfaces are well-defined.

Criterium	Description
Well-defined interfaces	It is clearly defined what the component requires and provides. Generally, this means: ▪ All outputs that are modified by exactly one component ( ) belong to this component (,). ▪ All inputs that are modified by exactly one component ( ) belong to this component (,). When developing in a bottom-up development approach, treat all the inputs that your component has as belonging to someone else. Usually, this will be one of the following: ▪ A model component higher in the model hierarchy that provides the required variables and data types. ▪ An independent component specified via a DD-based CGU that provides the required variables and data types.
Glue code	The code that is required only to couple the components should be generated by an independent component. This works well if you know what your interfaces are. This is usually the case when developing in a top-down development approach.

Special considerations for functions

TargetLink can generate functions that can be called by different callers. To generate their definitions into the correct modules, special considerations are required:

Function Kind	Instructions
Table functions	Reducing Code Size via Table Function Reuse (TargetLink Preparation and Simulation Guide)
External functions	Overview of Methods for Embedding External Code on page 358 How to Include External Code Files via DD Modules on page 379
Function reuse	Reusing C Functions and Variables for Identical Model Parts (Function Reuse) on page 297

Related topics

Basics

Basics on Modules and Module Ownership..... 388
Defining the Function Interface..... 241

Basics on Modules and Module Ownership

Module-oriented code generation

You can specify to generate code elements into specific modules. You do this by providing a module specification ( ) via DD **Module** objects.

You can reference DD **Module** objects at **Function** blocks and at the following DD objects:

DD Object
AccessFunction
Block[1]
DataVariant
FunctionClass
Typedef **Tip** To facilitate working with data types in distributed development scenarios, TargetLink provides the **UsePlatformTypeAbstractionLayer** (TargetLink Model Element Reference) Code Generator option. This option lets you change the mapping to platform-specific types in a single file per CGU rather than multiple files if necessary.
Variable
VariableClass

[1] For specifying look-up tables.

During code generation, TargetLink generates code elements resulting from DD objects that have a module specification ( ) into a code module ( ).

Module contents

You can influence the contents of the code module ( ) created from the DD **Module** object the object's **ModuleInfo.CodeGenerationBasis** property:

Value	Code Module Content
`DDBased`	Model-independent code elements using this module specification. Possible for the definitions of global variables and data types that can be generated from the DD alone. You must not use this module specification for Model-dependent code elements.
`ModelBased`	Model-dependent code elements using this module specification. Only the code elements required by the CGU are generated into the code module. All other code elements associated with this module specification are ignored.
`ModelAndDDBased`	As with `ModelBased`. Additionally, all the model-independent code elements using this module specification are generated into the module.

General module owners

You can specify a general owner for a DD **Module** object via the object's **ModuleInfo.Responsibilty** property.

This property determines who is responsible for the production code version of the code module ( ).

If the ModuleInfo.Responsibilty property of a DD Module object is set to `Default`, the general owner depends on the property value of the ModuleInfo.CodeGenerationBasis property:

CodeGenerationBasis	General Owner
`ModelBased`	Model
`ModelAndDDBased`	Model
`DDBased`[1]	Data Dictionary

[1] Refer to DD-Based Code Generation on page 397.

Stub code versions of code modules TargetLink generates a stub code version of a code module ( ) whenever the CGU used to generate code requires the module but does not own it.

Accordingly, when generating code for a model-based CGU that requires a DD-based module, TargetLink will generate a stub code version of the DD-based module.

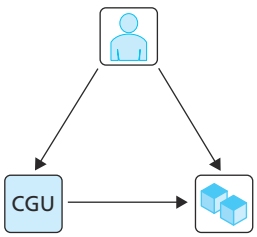
Note

Partitioning a model into CGUs requires specific module Owners

When you partition the model into different CGUs, define specific module owners.

Specific module owner

As explained in Principles of Partitioning Models and Code on page 382, you must refine general module ownership to specific ownership ( ) if you want to partition a model into different CGUs. You do this by specifying a DD ModuleOwnership object that establishes a connection between a CGU ( ) and one or more DD Module objects ( ):



The following table shows possible specific owners depending on the module's general ownership:

Value of ModuleInfo.Responsibilty Property	Specific Owner
`DataDictionary`	DD CodeGenerationUnit object[1]
`External`	None[3]

Value of **ModuleInfo.Responsibilty** Property	Specific Owner
`Internal`	None[2]
`Model`	Model-based CGU

[1] Refer to DD-Based Code Generation on page 397.
[2] Reserved for TargetLink. You must not create modules whose **Responsibility** is property set to `Internal`.
[3] Used for external code files. You are responsible for the deployment and the contents of these files. Refer to Including External Code via the Data Dictionary on page 376.

Note

Each module can have at most one specific owner. The relationship is `1:1`. Each owner can own several modules. The relationship is `1:n`. For example, if you specified a specific owner for a DD **ModuleGroup** object, each of the DD **Module** objects contained in this group is owned by the group's owner. You can no longer specify a specific owner for an individual DD **Module** object.
For a modeling example, refer to the **CG_FROM_DD** demo model.

Related topics

Basics

CG_FROM_DD (TargetLink Demo Models)
DD-Based Code Generation 397
Including External Code via the Data Dictionary 376
Principles of Partitioning Models and Code 382
Specifying the Location of Artifacts Generated or Used by TargetLink 71

HowTos

How to Create Module Specifications in the Data Dictionary 391
How to Specify a DD-Based CGU as the Specific Owner of a DD Module Object 395
How to Specify a Model-Based CGU as the Specific Owner of a DD Module Object 393

References

Function Block (TargetLink Model Element Reference)

How to Create Module Specifications in the Data Dictionary

Objective

To create a module specification ( ) that instructs TargetLink to generate code elements into a specific code module ( ).

Precondition

The DD objects or Function blocks that specify the code elements you want to generate into the code module must exist.

Method

To create a module specification in the Data Dictionary

1. In the Data Dictionary Manager, select the `/Pool/Modules` object.
2. From the context menu, select Create Module.
   TargetLink creates a new DD Module object.
3. Rename and specify the DD Module object as required.

   Note

   If you want to use this DD Module object to generate code from the Data Dictionary, you must set its ModuleInfo.CodeGenerationBasis property to `DDBased`.

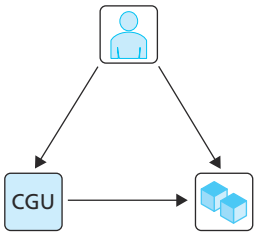
4. Reference the DD Module object at the DD objects or Function blocks whose code elements are to be generated into the code module.

Result

You created a module specification ( ) in the Data Dictionary.

Next step

If you want to partition your model into different model-based code generation units ( ), you must specify a specific owner ( ) of the DD Module object:



Refer to How to Specify a Model-Based CGU as the Specific Owner of a DD Module Object on page 393 or How to Specify a DD-Based CGU as the Specific Owner of a DD Module Object on page 395.

Related topics

Basics

Basics on Modules and Module Ownership..... 388
Including External Code via the Data Dictionary..... 376
Principles of Partitioning Models and Code..... 382

HowTos

How to Specify a DD-Based CGU as the Specific Owner of a DD Module Object..... 395
How to Specify a Model-Based CGU as the Specific Owner of a DD Module Object..... 393

References

Function Block (TargetLink Model Element Reference)

How to Specify a Model-Based CGU as the Specific Owner of a DD Module Object

Objective

To specify a model-based code generation unit (CGU) ( ) as the specific owner ( ) of a DD Module object.

Precondition

You created a module specification.

Workflow

This workflow consists of the following parts:

- Part 1 - To create a DD ModuleOwnership object and reference DD Module or ModuleGroup objects
- Part 2 - To specify a model-based CGU as the specific owner of DD Module or ModuleGroup objects

Part 1

To create a DD ModuleOwnership object and reference DD Module or ModuleGroup objects

1 In the Data Dictionary Manager, select the `/Pool/ModuleOwners` object.

> **Tip**
>
> If your Data Dictionary does not contain this object, create it via the `/Pool` object's context menu.

2 From the context menu of the `/Pool/ModuleOwners` object, select Create ModuleOwnership.

3 Rename the ModuleOwnership object as required.

4 Reference the DD Module or ModuleGroup objects in the Property Value List of the ModuleOwnership object's ModuleRefs or ModuleGroupRefs child objects.

Interim result

You created a DD ModuleOwnership object and referenced one or more DD Module and/or ModuleGroup objects.

Next, you specify the model-based CGU that is to own the DD ModuleModuleGroup objects.

Part 2

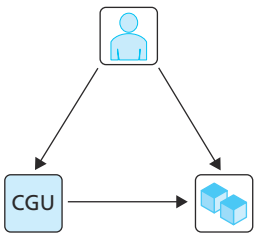
To specify a model-based CGU as the specific owner of DD Module or ModuleGroup objects

1 In the Data Dictionary Manager, select the DD ModuleOwnership object.

2 In its Property Value List, specify the name of the model-based CGU via the SystemName property.

Result

You specified a model-based CGU as the specific owner of DD Module or ModuleGroup objects:



When generating code for the owning CGU, TargetLink creates production code versions of the code modules that are owned by the CGU. When generating code for another CGU that requires these modules but does not own them, TargetLink creates stub code versions of these modules.

Related topics

Basics

Basics on Modules and Module Ownership..... 388
Principles of Partitioning Models and Code..... 382

HowTos

How to Create Module Specifications in the Data Dictionary..... 391

How to Specify a DD-Based CGU as the Specific Owner of a DD Module Object

Objective

To specify a DD-based code generation unit (CGU) ( ) as the specific owner ( ) of a DD Module object.

Preconditions

The following preconditions must be fulfilled:

- You created a module specification.
- You created a DD CodeGenerationUnit object.

Workflow

This workflow consists of the following parts:

- Part 1 - To create a DD ModuleOwnership object and reference DD Module or ModuleGroup objects
- Part 2 - To specify a DD CodeGenerationUnit object as the specific owner of DD Module or ModuleGroup objects

Part 1

To create a DD ModuleOwnership object and reference DD Module or ModuleGroup objects

1. In the Data Dictionary Manager, select the `/Pool/ModuleOwners` object.

   Tip

   If your Data Dictionary does not contain this object, create it via the `/Pool` object's context menu.

2. From the context menu of the `/Pool/ModuleOwners` object, select Create ModuleOwnership.
3. Rename the ModuleOwnership object as required.
4. Reference the DD Module or ModuleGroup objects in the Property Value List of the ModuleOwnership object's ModuleRefs or ModuleGroupRefs child objects.

Interim result

You created a DD ModuleOwnership object and referenced one or more DD Module and/or ModuleGroup objects.

Next, you specify the DD CodeGenerationUnit object that is to own the DD ModuleModuleGroup objects.

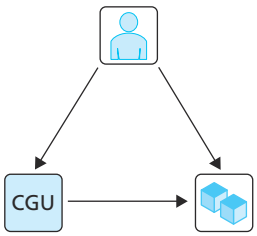
Part 2

To specify a DD CodeGenerationUnit object as the specific owner of DD Module or ModuleGroup objects

1. In the Data Dictionary Manager, select the DD ModuleOwnership object.
2. In its Property Value List, reference the CodeGenerationUnit object via the CodeGenerationUnitRef property.

Result

You specified a DD CodeGenerationUnit object as the specific owner of DD Module or ModuleGroup objects:



When generating code for the owning CGU, TargetLink creates production code versions of the code modules that are owned by the CGU. When generating code for another CGU that requires these modules but does not own them, TargetLink creates stub code versions of these modules.

Related topics

Basics

Basics on Modules and Module Ownership..... 388
Principles of Partitioning Models and Code..... 382

HowTos

How to Create a DD CodeGenerationUnit Object for DD-Based Code Generation..... 400
How to Create Module Specifications in the Data Dictionary..... 391

DD-Based Code Generation

Where to go from here

Information in this section

Basics on Generating Code from the Data Dictionary.....397

How to Create a DD CodeGenerationUnit Object for DD-Based Code Generation.....400

Example of Specifying Calibratable Variables Used by Several Model-Based CGUs.....401

Example of Specifying InterfaceVariables Used by Several Model-Based CGUs.....403

Basics on Generating Code from the Data Dictionary

Introduction

TargetLink lets you generate the following code elements directly from the Data Dictionary:

- Definitions of global variables.
- Definitions of data types.
- Definition of variables for look-up tables.

To generate such code elements, you need DD **CodeGenerationUnit** objects.

Creating DD CodeGenerationUnit objects

You can create DD **CodeGenerationUnit** objects in the following subtree of the Data Dictionary:

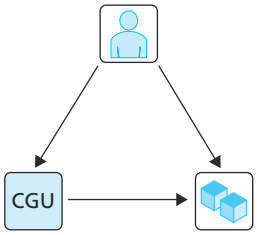
`/Pool/CodeGenerationUnits`

DD **CodeGenerationUnit** objects can have the **ModuleRefs** and **ModuleGroupRefs** child objects, whose properties let you reference one or more DD **Module** or **ModuleGroup** objects.

Specifying DD-based modules

The DD **Module** objects you reference at DD **CodeGenerationUnit** objects must be specified to be independent of the model. You do this by setting the **CodeGenerationBasis** property of the module's **ModuleInfo** child object to `DDBased`.

If the module ( ) contains code elements that are required by more than one DD-based CGU ( ), you have to specify a specific owner ( ) via a DD **ModuleOwners** object.



This prevents the generation of different production code versions of the same module by introducing stub code and lets you generate globally used modules. Refer to Principles of Partitioning Models and Code on page 382.

Generating restart and term functions

If you generate code for variables that are to be initialized in a restart function, TargetLink generates this function into a module ( ) that is called the CGU's root module. By default, the root module has the same name than the DD CodeGenerationUnit object to whom it belongs.

Changing the root module name via the name template You can specify the DD CodeGenerationUnit.NameTemplate property. If the name that you specify matches the name of another module generated by TargetLink, TargetLink will merge these modules.

Changing the root module name via reference You can reference a DD-based Module object via the CodeGenerationUnit.CodeGenerationUnitRootModuleRef property. TargetLink names the root module according to this module's ModuleInfo.NameTemplate property.

Starting DD-based code generation

You can start DD-based code generation as follows:

- From the context menu of DD CodeGenerationUnit objects
- From the context menu of DD Module and ModuleGroup objects
- From the Code Generator option set group box of the Code Generation page of the TargetLink Main Dialog
- Via the `tl_generate_code` API function

Use cases for globally used modules

The following table shows several use cases and modeling examples for global modules:

Use Case	Description	Modeling Example
Interface variables and data types	By generating interface variables and their data types into a dedicated module, you can separate the interface code from the code of the model components sharing the interface.	CG_FROM_DD demo model

Use Case	Description	Modeling Example
Parameter sharing	By generating parameters into a dedicated module, they can be used by different model components. When using data variants for DD Variable objects or page switching, consider the following information: ▪ Basics on Variant Coding and Conditional Code Compilation on page 562 ▪ Example of Changing Parameter Values on the ECU on page 104	▪ MODEL_REFERENCING demo model ▪ MODULAR_DEVELOPMENT demo model ▪ CG_FROM_DD demo model
Generating A2L files for variables generated from the Data Dictionary	You can separate the variables used by a table function from the function's definition. This lets you generate an A2L for the table's variables without generating code for the table function.	CG_FROM_DD demo model

Related topics

Basics

Basics on Modules and Module Ownership..... 388
Basics on Using Templates..... 203
CG_FROM_DD (📖 TargetLink Demo Models)
Details on Function Class Templates..... 220
Generating Unique Names via Name Macros..... 78
Including External Code via the Data Dictionary..... 376
MODEL_REFERENCING (📖 TargetLink Demo Models)
MODULAR_DEVELOPMENT (📖 TargetLink Demo Models)
Principles of Partitioning Models and Code..... 382

HowTos

How to Create a DD CodeGenerationUnit Object for DD-Based Code Generation..... 400
How to Specify a DD-Based CGU as the Specific Owner of a DD Module Object..... 395
How to Specify a Model-Based CGU as the Specific Owner of a DD Module Object..... 393

References

tl_generate_code (📖 TargetLink API Reference)

How to Create a DD CodeGenerationUnit Object for DD-Based Code Generation

Objective

To create a DD CodeGenerationUnit object to generate global variables or data types directly from the Data Dictionary.

Precondition

You created a module specification.

Method

To create a DD CodeGenerationUnit Object for DD-based code generation

1. In the Data Dictionary Manager, select the `/Pool/CodeGenerationUnits` object.
2. From its context menu, select Create CodeGenerationUnit.
3. Rename and specify the CodeGenerationUnit object as required.
4. From its context menu, select Create ModuleRefs or Create ModuleGroupRefs.
5. Select the ModuleRefs/ModuleGroupRefs child object.
6. In its Property Value List, reference one or more DD Module or ModuleGroup objects.

Note

The DD Module objects you reference must have their ModuleInfo.CodeGenerationBasis property to `DDBased`.

Result

You created a DD CodeGenerationUnit objects.

You can now generate code for the DD Module objects it references.

Note

A DD CodeGenerationUnit object does not automatically become a module's specific owner by referencing it.
Refer to How to Specify a DD-Based CGU as the Specific Owner of a DD Module Object on page 395.

Next step

If you create further DD CodeGenerationUnit objects you should think about module ownership. Refer to Basics on Modules and Module Ownership on page 388.

Related topics

Basics

Basics on Generating Code from the Data Dictionary..... 397
Basics on Modules and Module Ownership..... 388
Principles of Partitioning Models and Code..... 382

HowTos

How to Create Module Specifications in the Data Dictionary..... 391
How to Specify a DD-Based CGU as the Specific Owner of a DD Module Object..... 395

Example of Specifying Calibratable Variables Used by Several Model-Based CGUs

Introduction

The CG_FROM_DD demo model contains a DD CodeGenerationUnit object called `Parameters_CGU`.

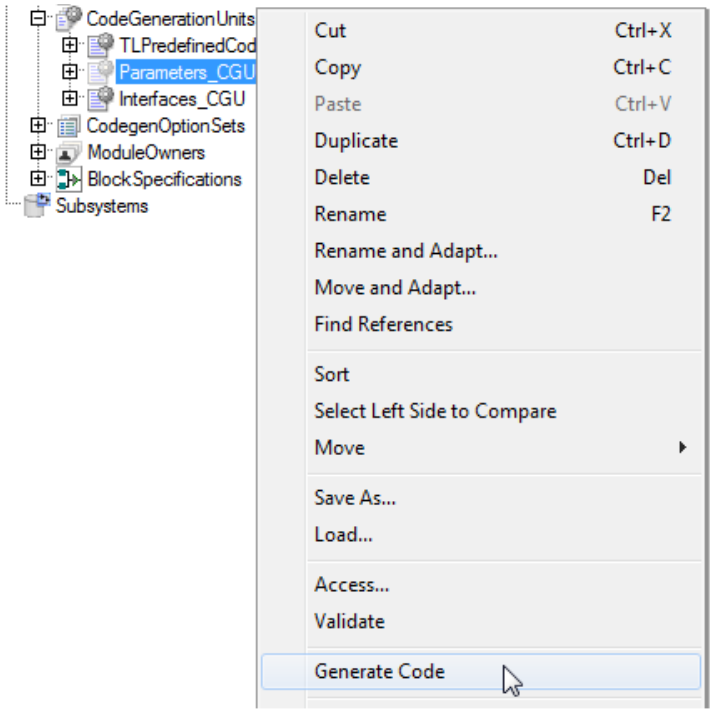
This DD-based code generation unit (CGU) is used to generate the definitions of calibration parameters that are used by different model-based CGUs contained in the model.

Module specifications

Via its ModuleGroupRefs child object, the DD-based CGU references a DD ModuleGroup object called `Common` that contains the DD Module called `Calibratables`. This object provides the module specification for the gain parameters specified by the DD Variable objects called `myGainA` and `myGainB`.

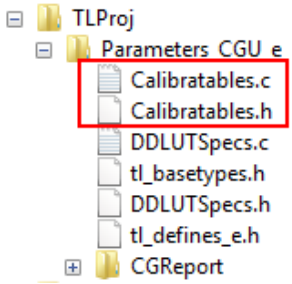
Starting DD-based code generation

You can start the DD-based code generation via the CGU's context menu.



Generated code files

Production code files are located in the folder for production code files defined via the DD ProjectFolder and FolderStructure objects..



The code module called`Calibratables` contains the declarations and definitions of the gain parameters:

Header File	Source File
`extern CAL Int16 myGainA;` `extern CAL Int16 myGainB;`	`CAL Int16 myGainA = 26 /* 0.8 */;` `CAL Int16 myGainB =  48 /* 1.5 */;`

`tl_defines_e.h` and `tl_basetypes.h` provide the required data types.

The code module called `DDLutSpec` stems from the module specification provided by the DD Module called `DDLUTSpecs` that also is contained in the DD ModuleGroup object called `Common`.

Related topics

Basics

Basics on Generating Code from the Data Dictionary..... 397
CG_FROM_DD (TargetLink Demo Models)
Mapping Code Generation Units to Code (Module Ownership)..... 382
Principles of Partitioning Models and Code..... 382

HowTos

How to Create Module Specifications in the Data Dictionary..... 391

Example of Specifying InterfaceVariables Used by Several Model-Based CGUs

Introduction

The CG_FROM_DD demo model contains a DD CodeGenerationUnit object called `Interfaces_CGU`.

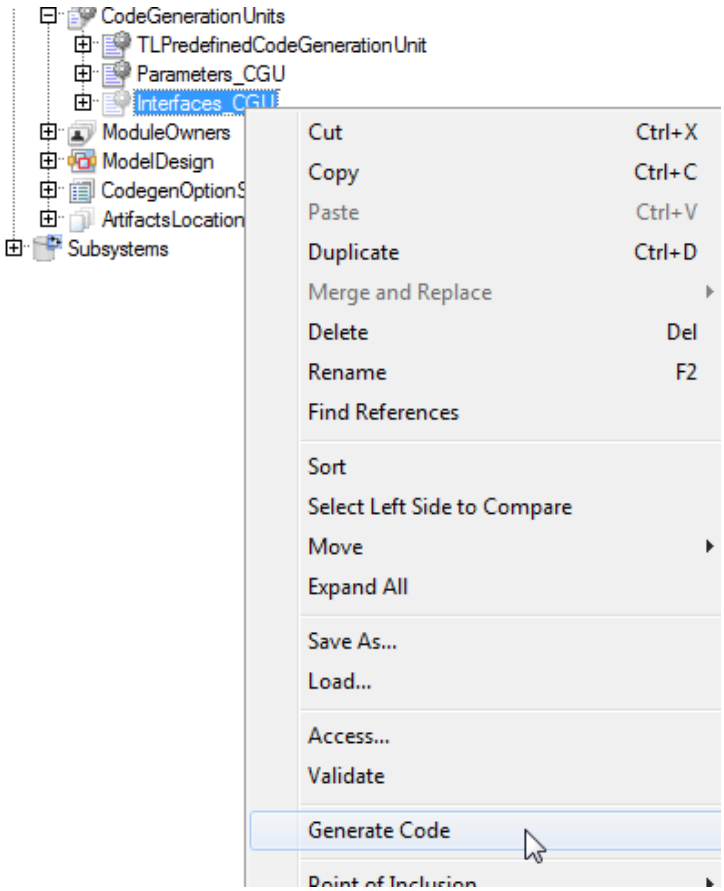
This DD-based code generation unit (CGU) is used to generate the definitions of interface variables that are used by different model-based CGUs contained in the model.

Module specifications

Via its ModuleRefs child object, the DD-based CGU references a two DD Module objects called `ifaceSubSysC` and `ifaceSubSysD`. These objects provide the module specification for the interface variables specified by the DD Variable objects called `myPortCout` and `myPortDout`.

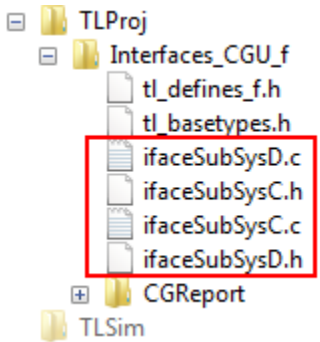
Starting DD-based code generation

You can start the DD-based code generation via the CGU's context menu.



Generated code files

Production code files are located in the folder for production code files defined via the DD ProjectFolder and FolderStructure objects..



The code module called`ifaceSubSysC` contains the declarations and definitions of the gain parameters:

Header File	Source File
`extern DISP Int16 myPortCout;`	`DISP Int16 myPortCout;`

The code module called`ifaceSubSysD` contains the declarations and definitions of the gain parameters:

Header File	Source File
`extern DISP Int16 myPortDout;`	`DISP Int16 myPortDout;`

`tl_defines_f.h` and `tl_basetypes.h` provide the required data types.

Related topics

Basics

Basics on Generating Code from the Data Dictionary 397
CG_FROM_DD (📖 TargetLink Demo Models)
Mapping Code Generation Units to Code (Module Ownership) 382
Principles of Partitioning Models and Code 382

HowTos

How to Create Module Specifications in the Data Dictionary 391

Decomposing Models Logically (Incremental Code Generation)

Where to go from here

Information in this section

Basics on Incremental Code Generation 406
How to Incrementally Generate Code for Function Subsystems 408
How to Extract Subsystems to a Test Frame in a Separate Model 411

Basics on Incremental Code Generation

Defining incremental code generation

Incremental code generation means generating production code separately for specific subsystems. The subsystems must be configured for incremental code generation and must contain a TargetLink **Function** block with the **Enable Incremental Code Generation** checkbox selected on the **Incremental** page. When you build the simulation application for the root-level TargetLink subsystem, the production code that is generated incrementally for the nested subsystems (external functions) is automatically linked to the same executable.

Advantages of incremental code generation

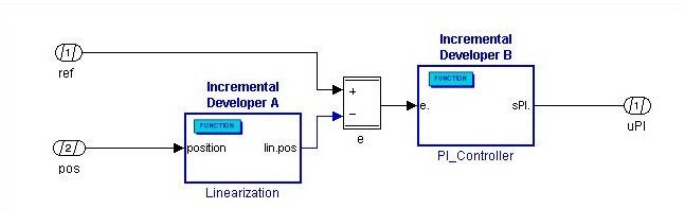
In a team of developers, each member can work on a subsystem individually and perform incremental code generation for it. TargetLink performs the necessary consistency checks when building the overall simulation application from all the parts. The root level TargetLink subsystem and the incrementally generated subsystems form a hierarchical tree. There are no limitations concerning subsystem hierarchy.

As a result, incremental code generation can significantly speed up the code generation for large models. Unchanged subsystems or subsystems which have passed the quality assurance process do not need to be generated again.

Example

The following examples show a typical application scenario for incremental code generation:

Suppose you are working on the `Linearization` subsystem, which is part of the root-level TargetLink subsystem `controller`.



When you finish modeling, you do not want to process the root-level TargetLink subsystem, but only the `Linearization` subsystem. For more information, refer to How to Incrementally Generate Code for Function Subsystems on page 408.

Suppose your colleague has also finished his or her work on the `PI_Controller` subsystem , which belongs to the same root-level TargetLink subsystem. In this case, incrementally generated production code for the subsystem already exists, which also means that you can use the code for your simulation application.

When you start the code generation process for the root-level TargetLink subsystem, the `Linearization` and `PI_Controller` subsystems are treated like external subsystems. Only calls to the respective functions `Linearization`() and `PI_Controller`() are generated into the code and the related header files are included. When the final simulation application is built, all production code, files including the incrementally generated ones, are compiled and linked to the same executable.

Note

In the TargetLink Data Dictionary, the hierarchy of incrementally generated subsystems is defined in the `/<Application>/Config` node. The Code Generator updates this node each time you generate new code.

Performing consistency checks

TargetLink performs several consistency checks when building a simulation application for incrementally generated subsystems. The build process is aborted if there are inconsistencies. Observe the following points:

- When target-specific code is generated, the processor/compiler combinations used for each subsystem must be compatible.
- The base type definitions must be compatible, for example, the byte order and the alignment.
- The root functions of the subsystems configured for incremental code generation must be called from the parent subsystem.
- The interface variables between the incrementally generated subsystems must be consistent.

Using data stores across code generation units

TargetLink **Data Store Memory** blocks can reside outside the code generation unit for which you want to generate code and that contains **Data Store Read** and **Data Store Write** blocks.

For more information, refer to How to Create a Global Data Store via Data Store Memory Blocks (📖 TargetLink Preparation and Simulation Guide).

Related topics

HowTos

How to Create a Global Data Store via a Simulink.Signal Object (📖 TargetLink Preparation and Simulation Guide)
How to Create a Global Data Store via Data Store Memory Blocks (📖 TargetLink Preparation and Simulation Guide)
How to Extract Subsystems to a Test Frame in a Separate Model..... 411
How to Incrementally Generate Code for Function Subsystems..... 408

References

Function Block (📖 TargetLink Model Element Reference)

How to Incrementally Generate Code for Function Subsystems

Objective

To allow for a more tailored code generation, you can incrementally generate code for a function subsystem by specifying it as a code generation unit (CGU).

Restrictions

There are restrictions when incrementally generating code for specific function systems:

- Incremental code generation is supported only for single-rate subsystems. There must not be any function-call connections into the subsystem.
- The name macro `$N` must not be used for function names, inputs or outputs. During code generation, the `$N` of the incremental function and of the parent subsystem are replaced by different subsystem names, so the function interface no longer matches.

Precondition

You created the following:

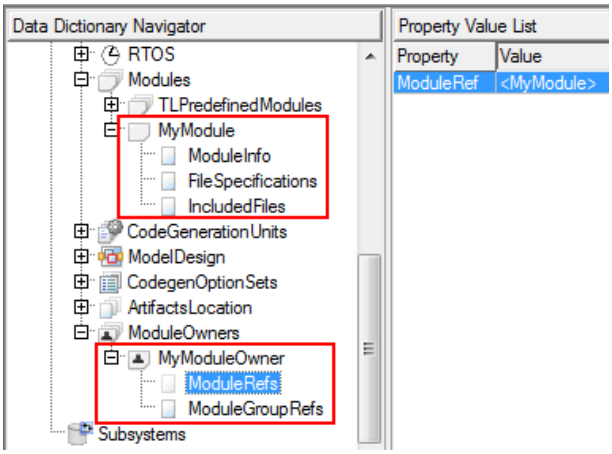
- A DD **Module** object called `MyModule`.
- DD **ModuleOwnership** object called `MyModuleOwner`.
- A function system called `MyFunctionSystem`

For more information, refer to Principles of Partitioning Models and Code on page 382 and Basics on Modules and Module Ownership on page 388.

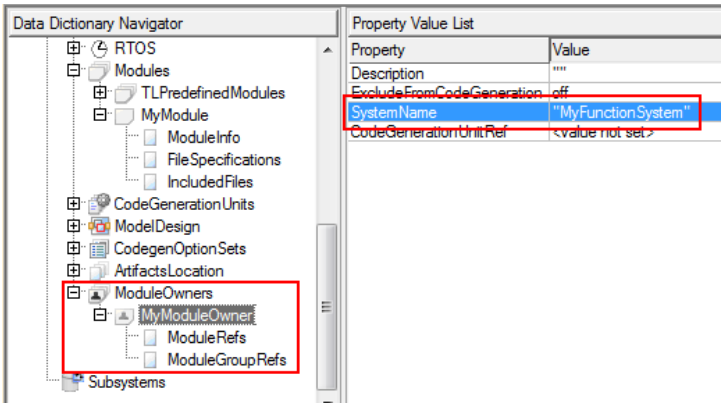
Method

To incrementally generate code for function systems

1 In the Data Dictionary Manager, reference the Module object called `MyModule` at the ModuleOwnership object called `MyModuleOwner`:

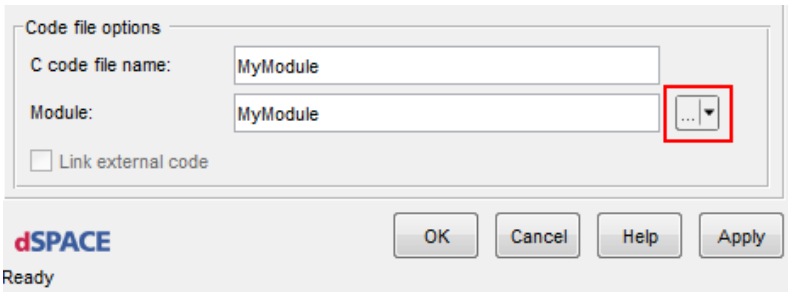


2 Select the MyModuleOwner object and set its SystemName property to `MyFunctionSystem`:

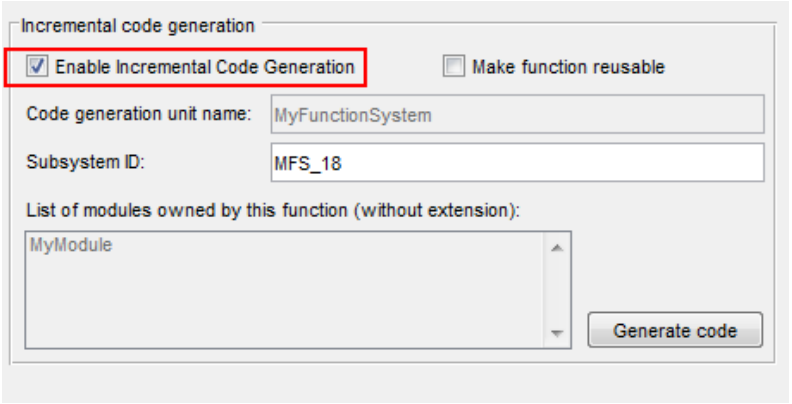


3 In the model, open the function system and open the Function block's dialog.

4 On the Options page, click the Module edit field's Browse button and select `MyModule` from the Data Dictionary:



5 On the Incremental page, select Enable incremental code generation and enter a unique Subsystem ID:



6 Click the Apply button and close the dialog.

7 Open the TargetLink Main Dialog block, select the CGU, and click the Generate Code button.

Result

Production code is generated for the nested subsystem.

Tip

If the production code contains a Restart, Init or Term function, carry out the following step.

On the Options page of the Function block, select the relevant Force checkbox.

When production code is generated for the parent subsystems, these flags are evaluated to determine if a Restart, Init or Term function has to be called for the child.

Tip

You can also start incremental code generation using the following API function:

```
tl_generate_code(...,
'FcnSubsystem','<model>/<subsys>/<fcnsubsys>');
```

To generate code for all subsystems configured for incremental code generation (from the innermost subsystem to the outermost one), set the `IncludeSubItems` property to on:

```
tl_generate_code(...,
'IncludeSubItems','on');
```

Related topics

Basics

Basics on Using Subsystems 226
Centrally Specifying and Synchronizing Function System Signatures 245
Decomposing Models into Model Parts (Model Referencing) 413
Decomposing Models Logically (Incremental Code Generation) 406
Details on the Expansion of Name Macros 80

References

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

How to Extract Subsystems to a Test Frame in a Separate Model

Objective

To perform software unit tests with the incrementally generated subsystem, you can extract it to a test frame in a separate model containing a ready-to-use simulation frame.

Method

To extract a subsystem to a test frame in a separate model

1. Open the dialog of the corresponding Function block and select the Incremental page.
2. In the Frame generation and subsystem extraction group box, specify the model or the full subsystem path in the destination model.
3. Select or clear the Supplement model with stimuli and measurement block checkbox according to your needs.
4. Click the Extract subsystem button.

Result

You can now test exactly the same production code as in the main model. There is no need to generate code again in the test model.

Related topics

Basics

Decomposing Models Logically (Incremental Code Generation)..... 406

HowTos

How to Incrementally Generate Code for Function Subsystems..... 408

References

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

Decomposing Models into Model Parts (Model Referencing)

Where to go from here

Information in this section

Introduction to Model Referencing.....413
Building Models with Model References from Scratch.....419
Building Models with Model References by Adapting and Converting Existing Models.....422
Generating Production Code in Combination with Model Referencing.....424
Example of Adapting Existing Models for Model Referencing.....430
Disabling and Restoring Model References.....442

Introduction to Model Referencing

Introduction

Model referencing lets you include models in other models by using Model blocks that represent references to other models. This is useful when it is possible to break down large models into smaller parts that can be implemented separately.

You have to know the basic concepts and the workflow of model referencing in TargetLink.

Where to go from here

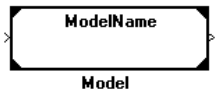
Information in this section

Basics on Model Referencing.....413
Details on Simulation and Logging of Referenced Models.....416

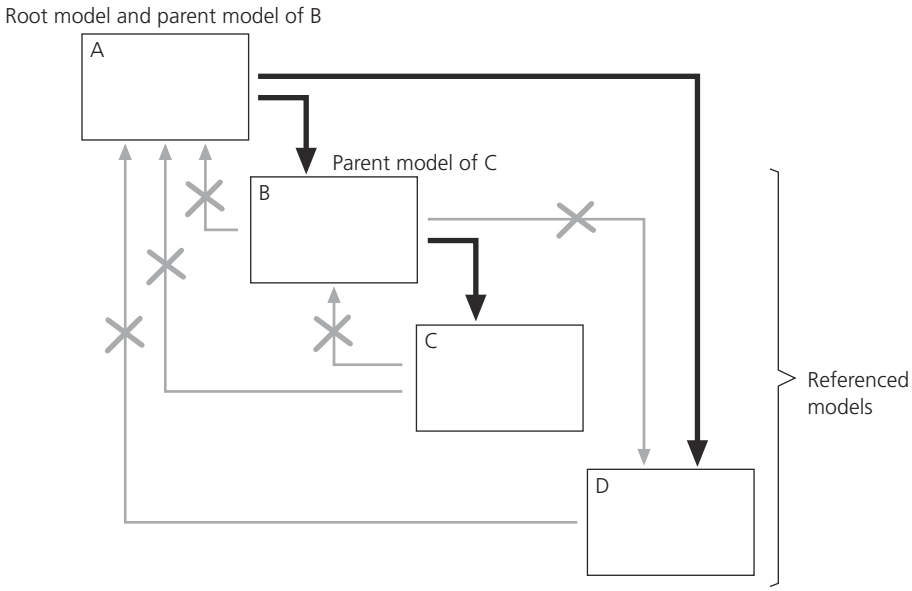
Basics on Model Referencing

Model referencing

Model referencing is useful when it is possible to break down a large model into smaller parts that can be implemented as separate models. You can include these models in other models via the Simulink Model block.



References to models can be nested, so that multiple parent models can be contained in one root model. The root model usually contains a **TargetLink Main Dialog** block and at least one TargetLink subsystem:



It is not possible for a Model block to refer back to the model that it references.

Benefits of model referencing

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.

Simulation and logging

There are differences in simulation and logging between TargetLink subsystems with and without references to other models. Refer to Details on Simulation and Logging of Referenced Models on page 416.

TargetLink Data Dictionary for model referencing

Referenced models are not associated with a DD project file. Therefore, no DD project file is loaded automatically when a referenced model is opened. If a referenced model is opened, the currently loaded DD project file remains unchanged. If the DD project file is not the desired one, you have to load it manually. If a DD project file is not loaded, TargetLink asks you which DD project file to open.

Tip

If you are an advanced user working in batch mode, the DD project file specified via the **Preferences Editor** is opened. You are not asked for a DD project file. It is opened only if no other DD project file is loaded yet.

Control flow

You can control the execution of referenced models via an additional control flow. To do this, you must place one of the following Simulink blocks on a referenced model's root level:

- An **Enable** block, which means you are implementing an enabled referenced model (i.e., data-based control flow).
- A **Trigger** block, which means you are implementing a triggered referenced model (i.e., event-based control flow).

Production code generation

In combination with model referencing, production code is generated at various stages of the development process and for different purposes. In contrast to code generation for TargetLink subsystems or subsystems configured for incremental code generation, code generation in a model referencing hierarchy has to be performed in the right order, i.e., code has to be generated for a referenced model before it can be generated for the referencing system. Refer to Generating Production Code in Combination with Model Referencing on page 424.

Managing models with references

To manage models containing model references, you can use the **TargetLink Main Dialog** block or the API.

Via the context menu of the **Code generation units** tree, you can access various context menu options that are helpful for model referencing. The available context menu options depend on the kind of code generation unit (CGU) you selected. For example, if you want to display nested CGUs, click the **Inspect referenced models** option. For more information on context menu options, refer to Context menu options of the code generation units.

Using data stores across code generation units

Data stores can reside outside the code generation unit for which you want to generate code and that contains **Data Store Read** and **Data Store Write** blocks.

For more information, refer to How to Create a Global Data Store via a Simulink.Signal Object (📖 TargetLink Preparation and Simulation Guide).

Limitations

For limitations on model referencing, refer to Component-Based Development Limitations (📖 TargetLink Limitation Reference).

Related topics

Basics

Component-Based Development (📖 TargetLink Demo Models)

HowTos

How to Create a Global Data Store via a Simulink.Signal Object (📖 TargetLink Preparation and Simulation Guide)
How to Create a Global Data Store via Data Store Memory Blocks (📖 TargetLink Preparation and Simulation Guide)

References

TargetLink Main Dialog Block (📖 TargetLink Model Element Reference)
tl_refmodel_to_subsystem (📖 TargetLink API Reference)
tl_sim (📖 TargetLink API Reference)
tl_subsystem_to_refmodel (📖 TargetLink API Reference)

Details on Simulation and Logging of Referenced Models

Introduction

There are some points to consider when you simulate a root model containing a TargetLink subsystem with references to other models.

Note

The referenced models are always simulated in the same simulation mode as the TargetLink subsystem that references them.

MIL simulation mode

If a TargetLink subsystem is simulated in MIL simulation mode, it is executed on a block basis. Simulink simulation mechanisms are used to execute the blocks in the referenced model, i.e., you can select one of the following Simulink simulation modes (not to be confused with TargetLink simulation modes):

- Normal mode (up to MATLAB R2007b): Simulink interprets the referenced model as if it were an atomic subsystem implemented directly within the parent model, and executes it.

- Accelerator mode: Simulink creates an S-function for the referenced model (called model S-function or referenced model's simulation target in the following), and runs it during simulation. When you start the simulation of the referenced model in the Accelerator simulation mode, the model S-function is built automatically.

For more information, refer to the Simulink documentation *About Referenced Model Simulation Modes*.

SIL and PIL simulation modes

If a TargetLink subsystem is simulated in SIL/PIL simulation mode, the production code generated for the TargetLink subsystem and all the referenced models in the model referencing hierarchy is executed.

Incremental loading for logging

One important feature of model referencing is incremental loading, which means that the referenced models are not opened and loaded until they are needed. For logging, TargetLink has to open the referenced models that are to be logged. If you want to log signals in a referenced model that is to be simulated in MIL simulation mode, the model is opened automatically and then closed afterwards.

Logging options for model referencing

TargetLink lets you specify whether to log signals in referenced models. The following model logging options are available in the Code generation units tree of the TargetLink Main Dialog for referenced models:

- Enable in all referenced models

  Signals in all referenced models are logged according to signal-specific logging options (set in the TargetLink block dialogs) and the global logging option (set in TargetLink Main Dialog).
- Enable in selected models

  Signals in selected referenced models are logged according to the signal-specific and global logging options.
- Disable in all referenced models

  No signals in any of the referenced models are logged, regardless of the signal-specific and global logging options.

In the Code generation units tree of the TargetLink Main Dialog, you can select multiple referenced models and set the logging options for them simultaneously.

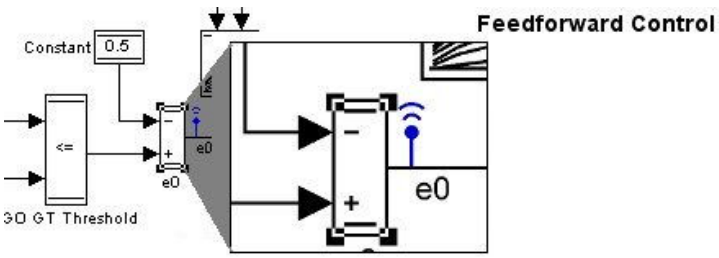
Note

- The model logging options for referenced models, specified in the TargetLink Main Dialog, do not apply in the SIL or PIL simulation mode. In these simulation modes, all signals specified for logging in the TargetLink block dialogs are logged. The referenced models do not need to be opened, because the production code is computed. Depending on the model logging option for referenced models in these simulation modes, more signals can be logged and plotted than in the MIL simulation mode.
- For signals in referenced models, the signal-specific logging option Min / Max values and the global logging option log min/max values are not supported in MIL simulation mode. The behavior is the same as if you select None in a TargetLink block dialog or `Do not log anything` in the TargetLink Main Dialog.
- For signals in referenced models that are not logged during MIL simulation, no overflow detection is performed.

Note

For referenced models, exclusive logging of minimum and maximum values in MIL simulation mode is not supported. However, when you log signal histories, minimum and maximum values are logged.

If signals in a referenced model are to be logged, TargetLink checks if any block outputs or Stateflow objects in the model are selected for logging at the start of the simulation. If this is the case, the signal-specific and global logging options are taken into account and relevant signals are marked with a Simulink test point.

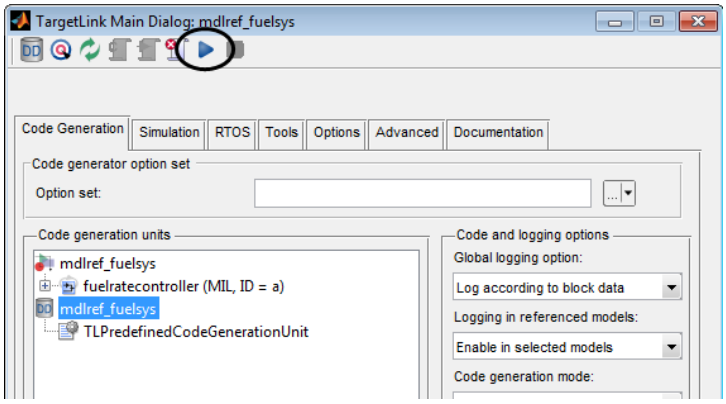


In addition, the model is saved and the model S-function (referenced model's simulation target) is built. The test points are removed for signals that provide test points, but should not be logged. These steps are executed only if necessary, i.e., TargetLink checks if the referenced model has changed since the model S-function was built the last time. If that is the case, the S-function is built again. Otherwise, it is not. To perform the above check, TargetLink uses a model-specific MAT file named `<refmodel>_tlds_logdata.mat`. These steps are performed for all referenced models.

If this file does not exist, for example because it was deleted, these steps are repeated permanently.

Starting a simulation in MIL mode

Referenced models can be logged in MIL mode only if the start of the simulation is performed via TargetLink, for example, via the **Start simulation** button of the **TargetLink Main Dialog** or a plot window, or via the **`tl_sim`** API function which is syntax-identical to the Simulink **`sim`** command.



Using TargetLink mechanisms to start the simulation is necessary to ensure correct logging in referenced models, because the S-functions for the referenced models (which have test points for logging) must be built before the simulation starts. Setting the test points during initialization is too late, because they would not take effect until the next simulation.

Building Models with Model References from Scratch

Introduction

If you want to work with models that have model references, you can either use existing models and adapt them for model referencing or you can build them from scratch.

When you build models with model references from scratch, you must build models that are suitable for model referencing and create models that reference them.

There are some points you have to consider.

Where to go from here

Information in this section

How to Build a Referenceable Model from Scratch..... 420

How to Create Models with Model References..... 421

How to Build a Referenceable Model from Scratch

Restrictions

To be suitable for model referencing, the model must fulfill all of the following conditions:

- It must not contain RTOS blocks.
- If it contains functional parts of an AUTOSAR runnable (i.e., *not* a complete runnable), the subsystem must not contain AUTOSAR blocks.
- It must not contain a **TargetLink Main Dialog** block or TargetLink subsystems.
- If it constitutes a function-call-triggered subsystem, its trigger port's **States when enabling** option must be set to **Reset** or **Held**.
- The model's interfaces must be completely defined, with TargetLink **InPort** and **OutPort** blocks whose width must be explicitly specified, either by bus objects or values other than `-1`.
- The name macros `$B`, `$M` or `$N` must not be used in the following edit fields on the **Options** page of the **Function** block:
  - **Step function name**
  - **Restart function name**
  - **Init function name**
  - **Term function name**
  - **C Code file name**

Method

To build a referenceable model from scratch

1. Build a model using blocks from the TargetLink block library following the requirements stated above.
2. Set the *Simulink* configuration parameters (**Simulation – Configuration Parameters**) required for model referencing as follows:

Location	Setting	Value
Data Import/Export – Save to workspace or file	Output checkbox	Cleared
Optimization – Signals and Parameters	Inline parameters checkbox	Selected
Diagnostics – Connectivity – Buses	Element name mismatch list	Error
	Mux blocks used to create bus signals list	Error

3. Insert a **Function** block at the root level, open its block dialog, and make the following settings:

Location	Setting	Value
Incremental page	Enable Incremental Code Generation checkbox	Selected
	Subsystem ID edit field	String consisting of alphanumeric characters and underscores

Location	Setting	Value
Options page	Make function reusable checkbox	Cleared
	C code file name edit field[1)]	String
Arguments page	Scaling invariant checkbox[1)]	Cleared

[1)] Not applicable for AUTOSAR.

4 Prepare the system for use with TargetLink using the API function `tl_prepare_system('System', <modelName>)`.

Result

You built a model suitable for model referencing. The model can now be referenced by another model. For more information, refer to How to Create Models with Model References on page 421.

Related topics

Basics

Partitioning Model and Code for Classic AUTOSAR (📖 TargetLink Classic AUTOSAR Modeling Guide)

How to Create Models with Model References

Objective

If you want to include a model in another model, you must create a model reference. Each model can have multiple model references, which are realized via Model blocks.

For model referencing with TargetLink, you can create references to other models on any hierarchy level inside TargetLink subsystems, inside subsystems configured for incremental code generation, and inside any model that is suitable for model referencing. For more information, refer to How to Build a Referenceable Model from Scratch on page 420.

Preconditions

The following preconditions must be fulfilled before you can perform the subsequent instructions.

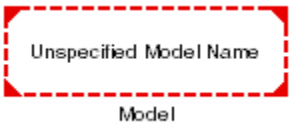
- The model you want to reference must meet the following requirements before you can perform the subsequent instructions:
  - It must be suitable for model referencing. For more information, refer to How to Build a Referenceable Model from Scratch on page 420.
  - It must be on the MATLAB search path.

Method

To create a model reference

1 Open the model in which you want to create a model reference.

2 From Simulink's Ports & Subsystems block library, drag a Model block to the position in your model where you want to create the model reference.



As long as you do not specify the name of the referenced model, the Model block is shown with red dotted lines.

3 Open the new Model block's parameter dialog and enter the name of the referenced model in the Model name edit field.

Result

You created a model reference to another model.

Tip

If you want to open a referenced model, double-click the Model block that references it.

Building Models with Model References by Adapting and Converting Existing Models

Introduction

If you want to work with models that have model references, you can build them from scratch or you can use existing models and adapt them for model referencing.

When you build a model with model references by adapting and converting an existing model, you add a Function block to the subsystem that you want to reference, make appropriate settings, and convert it into a model.

There are certain steps you have to consider.

How to Create Models with Model References by Adapting Existing Models

Restrictions

There are certain subsystems that cannot be converted to referenced models:

- TargetLink subsystems
- Reused subsystems
- Subsystems with an invariant scaling
- Subsystems containing RTOS blocks
- Subsystems modeling functional parts of runnables containing AUTOSAR blocks

The name macros `$B`, `$M` or `$N` must not be used in the following edit fields on the Options page of the Function block:

- Step function name
- Restart function name
- Init function name
- Term function name
- C Code file name

Precondition

The TargetLink subsystem contains at least one subsystem.

Method

To create a model with model references by adapting an existing model

1. Open a subsystem in the TargetLink subsystem that is to be converted into a referenced model.
2. Add a Function block in which the Enable Incremental Code Generation checkbox is selected.
3. Open its block dialog and and select the Incremental page.

   Select the Enable Incremental Code Generation checkbox.
4. Enter a subsystem ID in the Subsystem ID edit field.
5. Switch to the Options page.

   Make sure that none of the edit fields contain any of the forbidden name macros.
6. Define the interfaces completely using TargetLink InPort and OutPort blocks whose width must be explicitly specified, either by bus objects or values other than `-1`.
7. Repeat steps 1 to 6 for each subsystem that you want to convert into a referenced model.
8. Open the TargetLink Main Dialog to convert the subsystems into referenced models.
9. Select and right-click the subsystems to be converted in the Code generation units tree.
10. From the context menu, select Create reference.

11 In the **Converting subsystem(s) to model(s)** dialog that opens, click **Specify all names**.

12 In the **Name for '<Model>'** field, enter the names for the referenced models one after the other.

Result

The subsystems are converted into models and replaced by Model blocks referencing the newly generated models. The Simulink configuration parameters needed for model referencing are set automatically.

Generating Production Code in Combination with Model Referencing

Introduction

In combination with model referencing, production code is generated at various stages of the development process and for different purposes. There are some points you have to note.

Where to go from here

Information in this section

Basics of Generating Production Code in Combination with Model Referencing..... 424

How to Generate Code in Combination with Model Referencing..... 426

Basics of Generating Production Code in Combination with Model Referencing

Incremental code generation

TargetLink treats referenced models as subsystems configured for incremental code generation, i.e. all the restrictions that apply to subsystems configured for incremental code generation apply to referenced models as well. For more information, refer to Code Generation Limitations (📖 TargetLink Limitation Reference).

From the **TargetLink Main Dialog**, you can generate code for:

- TargetLink subsystems
- Subsystems configured for incremental code generation
- Referenced models

You can specify if production code is to be generated only for selected systems or for nested systems as well. For TargetLink subsystems contained in the root model, you can also build a simulation application host or target.

Tip

Via the context menu of the **Code generation units** tree, you can access various context menu options that are helpful for model referencing. The available context menu options depend on the kind of code generation unit (CGU) you selected. For example, if you want to display nested CGUs, click the **Inspect referenced models** option. For more information on context menu options, refer to Context menu options of the code generation units.

Order of code generation

In contrast to code generation for a TargetLink subsystem or a subsystem configured for incremental code generation without model references, the order in which code is generated for referencing systems and referenced models in a model referencing hierarchy is essential. During code generation for a referenced model, a subsystem object containing the interface description of the referenced model is generated in the Subsystem area of the Data Dictionary. Using this interface description, code can be generated for the referencing system without opening and searching the referenced model.

In a model referencing hierarchy, the code for a system (TargetLink subsystem, subsystem configured for incremental code generation, or referenced model) referencing another model can be generated only if the production code for the referenced model was already generated. Thus, the code must be generated recursively, starting with the bottommost referenced model.

If you select the **Include sub-items for code generation** checkbox, TargetLink lets you generate code for the complete model referencing hierarchy in one step and ensures the correct order.

If you generate code for specific systems in the model referencing hierarchy, you have to ensure the correct order yourself.

Related topics

HowTos

How to Generate Code in Combination with Model Referencing..... 426

How to Generate Code in Combination with Model Referencing

Objective

If you want to generate code in combination with model referencing, you can select whether to generate it for a single referenced model, a part of the model referencing hierarchy, or the complete hierarchy.

Code Generator options

When code is generated for referenced models, the Code Generator options specified in the root model's **TargetLink Main Dialog** block are used.

Preconditions

The following preconditions must be fulfilled before you can perform the subsequent instructions:

- The root model is open.
- In a model referencing hierarchy, code for a single referenced model on a higher level can be generated only if the code for referenced models on lower levels has already been generated.

Possible methods

Referenced models can be nested, i.e., referenced models can contain Model blocks that reference other models or subsystems configured for incremental code generation. You can generate:

- Code for a specific referenced model. You can use this method, for example, to create an interface description for the referenced model before it is exported for distributed development, or to generate code for the referenced model after the function development is finished and before it is reintegrated into the integrator model. Refer to Method 1.
- Code for a hierarchy of systems. Refer to Method 2.
- Code for all TargetLink subsystems in the root model. You can use this method to generate code for the overall model, or to generate glue code that holds code generated for the referenced models together. Refer to Method 3.

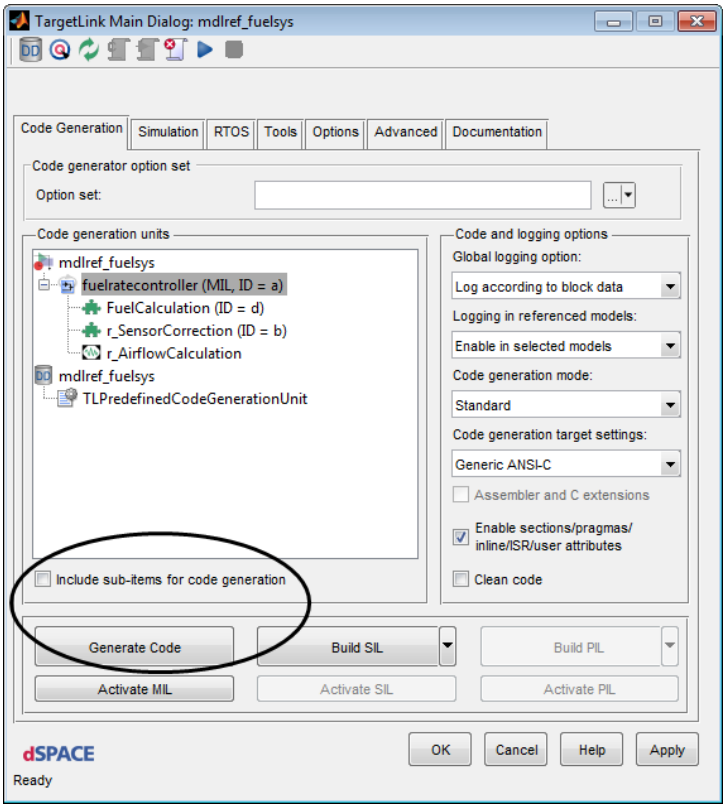
Method 1

To generate code for a referenced model

1. Open the **TargetLink Main Dialog**.
2. In the **Code generation units** tree, select the referenced model that you want to generate code for.

Tip

Via the context menu of the **Code generation units** tree, you can access various context menu options that are helpful for model referencing. The available context menu options depend on the kind of code generation unit (CGU) you selected. For example, if you want to display nested CGUs, click the **Inspect referenced models** option. For more information on context menu options, refer to Context menu options of the code generation units.



3 Click Generate Code.

Method 2

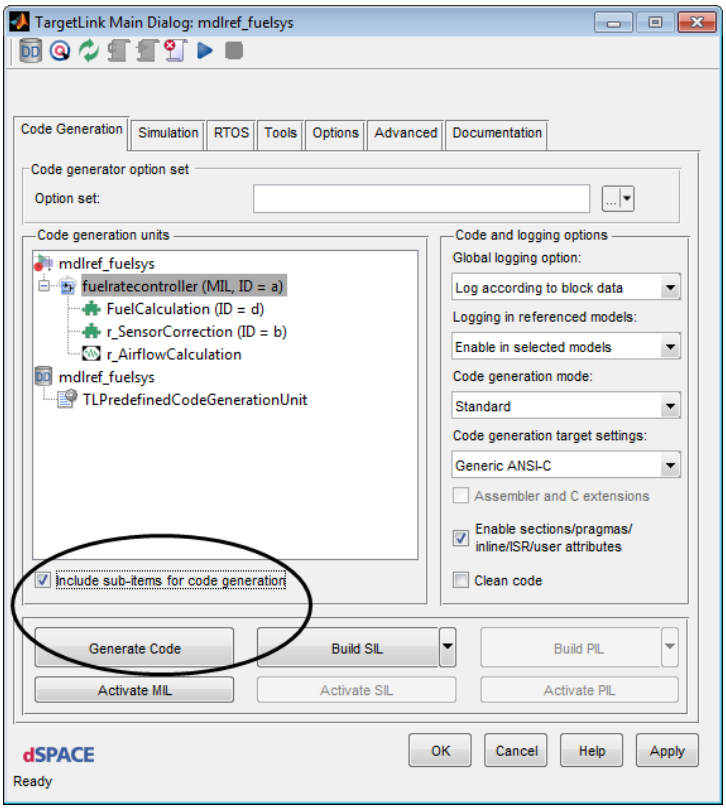
To generate code for a hierarchy of systems

1 Open the TargetLink Main Dialog.

2 In the Code generation units tree, select the system you want to generate production code for (including all nested systems that code is to be generated for).

3 Select Include sub-items for code generation.

TargetLink ensures that the code for the referenced models is generated in the correct order.



4 Click Generate Code.

Method 3

To generate code for all TargetLink subsystems in the root model

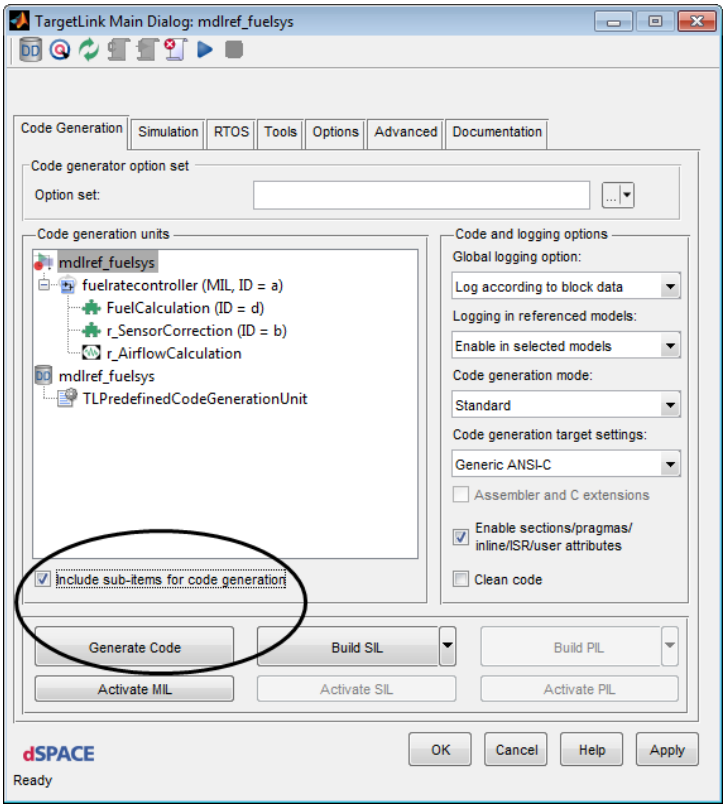
1 Open the TargetLink Main Dialog.

2 In the Code generation units tree, select the root model.

3 Select Include sub-items for code generation checkbox if you want to generate code for nested systems as well

TargetLink ensures that the code for the referenced models is generated in the correct order.

Clear this option if you want to generate code only for the TargetLink subsystems. Keep in mind that in this case, code for the nested referenced models must have been generated before.



4 Click Generate Code.

Result

Production code is generated.

By default, i.e., if you have not specified a C code file name in the respective Function block, the module generated for a referenced model is named <ModelName>.

Related topics

Basics

Basics of Generating Production Code in Combination with Model Referencing..... 424

References

tl_generate_code (📖 TargetLink API Reference)

Example of Adapting Existing Models for Model Referencing

Where to go from here

Information in this section

Example Overview of Adapting Existing Models for Model Referencing..... 430

Example of Preparing a Subsystem for Conversion into a Referenced Model..... 431

Example of Defining DD Variable Objects for Mergeable Variables..... 432

Example of Defining Data Dictionary Module Objects..... 435

Example of Defining the Module Ownership of Globally Defined Variables..... 436

Example of Importing Code Generator Options into the Data Dictionary..... 439

Example of Creating a Model Reference from a Subsystem..... 440

Example Overview of Adapting Existing Models for Model Referencing

Introduction

Suppose you have a complex model that you want to implement by using distributed development. To do this, you can adapt it and convert its subsystems into referenced models that can be handled separately. There are various points to note when adapting an existing model for model referencing.

The following examples give you step-by-step instructions on how to convert the FUELSYS demo model into a model with references as shown in the MODEL_REFERENCING demo model. The single examples are based on each other. Therefore, you have to carry them out one after the other.

Step	Documentation
Prepare a subsystem for conversion into a referenced model.	Example of Preparing a Subsystem for Conversion into a Referenced Model on page 431
Define DD **Variable** objects for mergeable variables.	Example of Defining DD Variable Objects for Mergeable Variables on page 432
Define DD **Module** objects.	Example of Defining Data Dictionary Module Objects on page 435
Define module ownership of globally defined variables.	Example of Defining the Module Ownership of Globally Defined Variables on page 436

Step	Documentation
Import Code Generator options into the Data Dictionary.	Example of Importing Code Generator Options into the Data Dictionary on page 439
Create a model reference from a subsystem.	Example of Creating a Model Reference from a Subsystem on page 440

Related topics

Basics

FUELSYS (TargetLink Demo Models)
MODEL_REFERENCING (TargetLink Demo Models)

Example of Preparing a Subsystem for Conversion into a Referenced Model

Introduction

TargetLink lets you convert a subsystem into a referenced model and automatically replace it with the **Model** block. However, you have to prepare the subsystem accordingly.

This example uses the FUELSYS demo model and is part of a workflow for adapting existing models for model referencing. Refer to Example Overview of Adapting Existing Models for Model Referencing on page 430.

Suppose you want to replace the **Airflow calculation** subsystem in the FUELSYS demo model with a **Model** block referencing a model created from this subsystem.

Method

To prepare the ./fuelratecontroller/Airflow calculation subsystem for conversion into a referenced model

1. Open the `fuelsys` model.
2. Open the **fuelratecontroller** subsystem.
3. Open the **Airflow calculation** subsystem.
4. Specify a **Dimension** other than `-1` at the TargetLink **InPort** and **OutPort** blocks.
5. Open the **Function** block.
6. On the **Incremental** page , select **Enable Incremental Code Generation** and enter `b` in the **Subsystem ID** edit field.
7. On the **Options** page, replace the $S_$B string in the names of the restart and term functions with the name of the step function, e.g., change RESTART_$S_$B to RESTART_IntakeAirflowEstimation.

Result

You prepared the **fuelratecontroller** subsystem for conversion into a referenced model.

Next steps

After you prepared the subsystem for conversion into a referenced model, you have to ensure that merged variables specified in the model can still be used after the subsystem is converted into a referenced model. Refer to Example of Defining DD Variable Objects for Mergeable Variables on page 432.

Related topics

Basics

FUELSYS (TargetLink Demo Models)

Examples

Example Overview of Adapting Existing Models for Model Referencing..... 430

Example of Defining DD Variable Objects for Mergeable Variables

Introduction

TargetLink lets you specify identical variables that you want to be generated for different blocks within one TargetLink subsystem as mergeable. If the mergeable variables are used in different code generation units, you must specify these variables in the Data Dictionary.

This example uses the FUELSYS demo model and is part of a workflow for adapting existing models for model referencing. Refer to Example Overview of Adapting Existing Models for Model Referencing on page 430.

To ensure that the merged variables specified in the FUELSYS demo model can still be used after the **Airflow calculation** subsystem is converted into a referenced model, you must specify them in the Data Dictionary.

Points to note with merged variables

You have to consider the following two kinds of mergeable variables:

1. Mergeable variables used at the interfaces of the referenced system and in the referencing model.

   In this case, TargetLink performs a real merge operation: Differences in the variable definitions are intercepted during code generation.
2. Mergeable variables used in the referenced system which are used in the referencing system at the same time. Typically, these variables are parameter variables.

In this case, TargetLink does not perform a merge operation between variables used in the referenced system and the referencing system. If the variables differ with regard to their properties at the inner and outer locations, this is not intercepted during code generation. You must ensure identical properties. To avoid multiple definitions of these variables, one in the module generated for the referenced model and one in the module generated for the referencing system, the Module property must be set at the DD variable objects to explicitly specify in which module the variables are to be defined. Without additional specifications the module is generated for the referenced model and for the referencing system. During the code generation for the referencing system, the module that was generated for the referenced model is overwritten and vice versa. For this reason, one system must be specified to be the owner of this module. The module for which an owner is specified is generated only when production code is generated for this system.

Merged variables in the fuelsys demo model

In the fuelratecontroller TargetLink subsystem in the fuelsys demo model, mergeable variables are specified for outports and inports of subsystems, which are directly connected. For example, for the outports of the Sensor correction and Fault Redundancy subsystem and the corresponding inports of the Airflow calculation system, mergeable variables are defined.

Furthermore, mergeable variables are specified for parameter variables. For example, the axis of the Prelookup block named Speed, contained in the Sensor correction and Fault Redundancy subsystem, is merged. The same applies to the row of the Look-Up Table (2D) block named Pumping Constant, contained in the Airflow calculation subsystem.

Workflow

This workflow consists of the following parts:

- Creating DD Variable objects for mergeable variables from a block dialog. Refer to Part 1.
- Referencing DD Variable objects for mergeable variables at a block dialog. Refer to Part 2.

Part 1

To create DD Variable objects from a block dialog

Note

The following instructions use the Look-Up Table (2D) and Prelookup blocks as an example to show how to define DD variable objects for merged parameter variables.

1. In the Sensor correction and Fault Redundancy subsystem, double-click the Prelookup block called Throttle Prelookup to open its block dialog.
2. In the block dialog, switch to the Axis page.
3. In the Name edit field, replace the $P name macro with the string contained in the Axis (breakpoint data) edit field, placed in the Axis values frame.

Copy the string from the **Name** field and paste it to the **Variable** edit field.

This prompts TargetLink to reference a DD **Variable** object of this name. Because the DD **Variable** object does not exist yet, it is highlighted in red.

Click **Apply**.

The variable object is created in the Data Dictionary and referenced at the block dialog.

4 In the Data Dictionary Manager, select the DD **Variable** object you created and adjust its **Width** property to match the width of the corresponding MATLAB workspace variable.

5 Adjust and repeat steps 1 to 4 for the blocks called **Speed Prelookup** and **Press Prelookup**.

Interim result

You created the following DD **Variable** objects and adjusted their **Width** property:

DD Variable Object	MATLAB Workspace Variable	Width
`axis_ThrotVect`	`ThrotVect`	17
`axis_SpeedVect`	`SpeedVect`	18
`axis_press`	`press`	19

Part 2

To reference DD Variable objects at a block dialog

1 In the **Airflow calculation** subsystem, open the **Look-Up Table (2D)** block called **Pumping Constant**.

2 Reference the DD **Variable** objects created in part Part 1 at the block dialog as shown in the following table:

Block Dialog Page	Edit Field	Variable
Axis#1	Variable	`axis_SpeedVect`
Axis#2	Variable	`axis_press`

3 Apply your changes and close the block dialog.

Result

You defined DD **Variable** objects for the merged interface and look-up table variables and referenced them at different blocks.

Next steps

You have to define the DD **Module** objects that specify the modules containing the variables in the **TargetLink Data Dictionary** to avoid double definitions of the variables. Refer to Example of Defining Data Dictionary Module Objects on page 435.

Related topics

Basics

FUELSYS (TargetLink Demo Models)

HowTos

How to Create New Data Dictionary Objects (TargetLink Data Dictionary Basic Concepts Guide)

Examples

Example Overview of Adapting Existing Models for Model Referencing..... 430

References

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

Example of Defining Data Dictionary Module Objects

Introduction

TargetLink lets you generate all variable definitions of a subsystem and calibratable parameters into separate modules. Therefore, you have to define DD **Module** objects that specify the modules. Refer to Principles of Partitioning Models and Code on page 382.

This example uses the FUELSYS demo model and is part of a workflow for adapting existing models for model referencing. Refer to Example Overview of Adapting Existing Models for Model Referencing on page 430.

Suppose you want to generate all merged interface variables of the code generation units that the TargetLink subsystem **fuelratecontroller** will contain as well as calibratable parameters into separate modules that are created when code is generated for the TargetLink subsystem and all its future code generation units, i.e., subsystems configured for incremental code generation and referenced models.

Method

To define DD Module objects

1. Open the Data Dictionary Manager.
2. Make sure that **View - Show unset properties** is selected.
3. In the **DD Navigator**, expand the DD object tree and right-click the **Modules** object.
4. From the context menu, select **Create Module**.

   A new DD **Module** object is created.

5 Enter **`AirflowCalculation`** as the name of the DD Module object.

6 Make sure that the NameTemplate in the ModuleInfo object is set to $D.

7 Set the CodeGenerationBasis property to ModelBased.

8 In the property value list, enter a Description, for example, **`Contains production code generated for system r_AirflowCalculation`**.

9 Repeat steps 4 - 8 for the TargetLink subsystem its other future code generation units according to the following table.

Module Object	NameTemplate	CodeGenerationBasis	Description
fuelratecontroller	fuelratecontroller	ModelBased	Contains production code generated for the TargetLink system fuelratecontroller.
SensorCorrection	$D	ModelBased	Contains production code generated for the future referenced model r_SensorCorrection.
FuelRateCalculation	$D	ModelBased	Contains production code generated for the future referenced model FuelCalculation.
CalParameters	$D	DDBased	Contains production code generated for calibratable parameters.

Result

You defined DD Module objects for the TargetLink subsystem fuelratecontroller and its other future Code generation units, i.e., subsystems configured for incremental code generation, referenced models, and calibratable parameters

Next steps

After you defined DD Module objects, you have to define the ownership of the modules. Refer to Example of Defining the Module Ownership of Globally Defined Variables on page 436.

Related topics

Basics

FUELSYS (📖 TargetLink Demo Models)
Principles of Partitioning Models and Code..... 382

Examples

Example Overview of Adapting Existing Models for Model Referencing..... 430

Example of Defining the Module Ownership of Globally Defined Variables

Introduction

TargetLink lets you define the module ownership for globally defined variables. This means that the module that contains globally defined variables is only

generated if production code is generated for the system that is the owner of the module. Refer to Principles of Partitioning Models and Code on page 382.

This example uses the FUELSYS demo model and is part of a workflow for adapting existing models for model referencing. Refer to Example Overview of Adapting Existing Models for Model Referencing on page 430.

Suppose you want to generate all merged interface variables of the code generation units that the TargetLink subsystem **fuelratecontroller** will contain and calibratable parameters into separate modules that are created when code is generated for the TargetLink subsystem and all its future Code generation units, i.e., subsystems configured for incremental code generation and referenced models.

Part 1

To define DD ModuleOwner objects

1. Open the Data Dictionary Manager.
2. In the DD Navigator, expand the DD object tree and right-click the Pool object.
3. Select Create ModuleOwners from the context menu.

   The DD ModuleOwners object is created.
4. Right-click the DD ModuleOwners object and select Create ModuleOwnerShip from the context menu.

   The new_ModuleOwnerShip object is created.
5. Enter the name **`AirflowCalculation`**.
6. In the property value list, enter a Description, for example, **`Specification of modules owned by the system r_AirflowCalculation`**.
7. Enter the SystemName, for example, **`r_AirflowCalculation`**.
8. In the ModuleRefs child object, click the ModuleRef Browse button and select the AirflowCalculation module.
9. Repeat steps 4 - 8 for the other module ownership objects according to the following table.

ModuleOwnerShip Object	SystemName	Description	ModuleRef
fuelratecontroller	fuelratecontroller	Specification of modules owned by the TargetLink subsystem fuelratecontroller.	fuelratecontroller
SensorCorrection	r_SensorCorrection	Specification of modules owned by the future referenced model r_SensorCorrection.	SensorCorrection
FuelCalculation	FuelCalculation	Specification of modules owned by the future referenced model FuelCalculation.	FuelRateCalculation

Part 2

To define the module ownership of the globally defined variables

1. Open the Data Dictionary Manager.
2. Select View - Show Object Explorer.
3. Make sure that View - Show unset properties and View - Object Explorer - Show all selected properties are selected.

4 In the **DD Navigator**, expand the DD object tree and select one of the DD **Variable** objects.

5 In the **Property Value List**, right-click the **ModuleRef** property and select **Show Property in Object Explorer** from the context menu.

6 In the **DD Navigator**, select the DD **Variables** object.

7 In the **Object Explorer**, select a DD **Variable** object you previously created, for example, the **EGO** variable object.

8 Click the **ModuleRef** Browse button and select a module object, for example, the **fuelratecontroller** module object.

You specified the module where the variables are defined.

Note

You can select multiple variable objects and set their **ModuleRef** properties for all the selected variable objects.

9 Repeat steps 7 - 8 for all other DD **Variable** objects according to the following table.

Variable Object	ModuleRef
est_air_flow	AirflowCalculation
feedback_correction	AirflowCalculation
axis_press	CalParameters
axis_SpeedVect	CalParameters
axis_ThrotVect	CalParameters
PressEst_z_table	CalParameters
pumpCon_z_table	CalParameters
SpeedEst_z_table	CalParameters
ThrotEst_z_table	CalParameters
fuel_rate	FuelRateCalculation
fail_O2	fuelratecontroller
fail_PRESS	fuelratecontroller
fail_SPEED	fuelratecontroller
fail_THROT	fuelratecontroller
fuel_mode	fuelratecontroller
press	fuelratecontroller
speed	fuelratecontroller
throttle	fuelratecontroller
tSim	fuelratecontroller
EGO_est	SensorCorrection
MAP_est	SensorCorrection
speed_est	SensorCorrection
throttle_est	SensorCorrection

Result

You defined the ownership of the modules containing the merged interface variables and the calibratable parameters.

Next steps

After you defined the ownership of the modules containing the merged variables and calibratable parameters, you have to import Code Generator options into the Data Dictionary. Refer to Example of Importing Code Generator Options into the Data Dictionary on page 439.

Related topics

Basics

FUELSYS (TargetLink Demo Models)
Principles of Partitioning Models and Code 382

Examples

Example Overview of Adapting Existing Models for Model Referencing 430

Example of Importing Code Generator Options into the Data Dictionary

Introduction

TargetLink lets you use the same Code Generator options for model-based and DD-based code generation units as well as for DD-generatable modules. Therefore, you have to import the Code Generator options that are saved at the model into the TargetLink Data Dictionary.

This example uses the FUELSYS demo model and is part of a workflow for adapting existing models for model referencing. Refer to Example Overview of Adapting Existing Models for Model Referencing on page 430.

Suppose you want to generate the DD-generatable module with the calibratable parameters you specified in the TargetLink Data Dictionary with the same Code Generator options as used for the model-based code generation units, i. e., TargetLink subsystems, subsystems configured for incremental code generation, and referenced models.

Method

To import the Code Generator options from the fuelsys model into the TargetLink Data Dictionary

1. In the MATLAB Command Window, type

   ```
   dsdd_import_optionset('OptionSetName',...
   'ImportedFromMdlref_fuelsys')
   ```

   The ImportedFromMdlref_fuelsys code generation option set object is created with the same Code Generator options as saved at the `fuelsys.mdl` model.

2 In the DD Navigator, select the **TLPredefinedCodeGenerationUnit** code generation unit object.

3 In the property value list, click the **CodegenerationOptionSetRef** Browse button and select the **ImportedFromMdlref_fuelsys** object.

The **ImportedFromMdlref_fuelsys** code generation option set object is referenced by the **TLPredefinedCodeGenerationUnit** code generation unit object.

Result

You imported the Code Generator options into the Data Dictionary and stored them as in the **ImportedFromMdlref_fuelsys** object.

Next steps

After you imported Code Generator options into the Data Dictionary, you can convert the **Airflow calculation** subsystem into a model and create a reference to it. Refer to Example of Creating a Model Reference from a Subsystem on page 440.

Related topics

Basics

Basics on Configuring the Code Generator for Production Code Generation..... 49
FUELSYS (📖 TargetLink Demo Models)

Examples

Example Overview of Adapting Existing Models for Model Referencing..... 430

References

dsdd_compare_optionset (📖 TargetLink API Reference)
dsdd_export_optionset (📖 TargetLink API Reference)
dsdd_import_optionset (📖 TargetLink API Reference)

Example of Creating a Model Reference from a Subsystem

Introduction

TargetLink lets you create a model reference from a subsystem configured for incremental code generation. The content of the subsystem is transfered into the referenced model and replaced with a **Model** block which holds the referenced model.

This example uses the FUELSYS demo model and is part of a workflow for adapting existing models for model referencing. Refer to Example Overview of Adapting Existing Models for Model Referencing on page 430.

Suppose you have prepared the existing Airflow calculation subsystem for model referencing, defined variable objects for merged variables in the Data Dictionary, and defined the ownership of the merged variables, you can convert the Airflow calculation subsystem into a model and create a reference to it.

Method

To create a model reference from a subsystem

1. Open the TargetLink Main Dialog.

   In the Code generation units tree, the systems hierarchy of the fuelsys model is displayed.
2. Expand the tree and right-click the Airflow calculation subsystem.
3. From the context menu, select Create reference.
4. In the Converting subsystem(s) to model(s) dialog that opens, click Specify all names.
5. In the Name for '<Model>' field, enter `r_AirflowCalculation`.

Result

The Airflow calculation subsystem is converted into a referenced model. The content of the subsystem is transfered into the referenced model and replaced with a Model block which holds the referenced model.

Next steps

To complete the conversion of the FUELSYS demo model into a model with references as shown in the MODEL_REFERENCING demo model, you have to perform the following steps:

- Prepare the Sensor correction and Faul Redundancy and Fuel Calculation subsystem for incremental code generation. Refer to Example of Preparing a Subsystem for Conversion into a Referenced Model on page 431.
- Create a model reference from the Sensor correction and Faul Redundancy subsystem. Refer to Example of Creating a Model Reference from a Subsystem on page 440

Related topics

Basics

FUELSYS (📖 TargetLink Demo Models)
MODEL_REFERENCING (📖 TargetLink Demo Models)

Examples

Example of Creating a Model Reference from a Subsystem..... 440
Example of Preparing a Subsystem for Conversion into a Referenced Model..... 431
Example Overview of Adapting Existing Models for Model Referencing..... 430

Disabling and Restoring Model References

Introduction

To develop the functions of a complex model, it is recommended not to work inside the referenced model, but rather to modify the function inside a subsystem.

To do so, you have to disable the model reference, and to restore it afterwards.

Where to go from here

Information in this section

How to Disable Model References..... 442

How to Restore Model References..... 445

How to Disable Model References

Objective

To develop the functions of a complex model, you are recommended not to work inside the referenced model, but rather modify the function inside a subsystem.

To work with a subsystem instead of a referenced model, you can disable the model reference in the parent model. The model reference is then replaced with a subsystem configured for incremental code generation that has the same contents as the referenced model.

Reasons for disabling model references

One reason for disabling a model reference is that during development, the function is modified frequently and its behavior has to be checked via simulation. This means that the simulation target (S-function) of the referenced system needed for the simulation must be rebuilt each time the referenced model is changed. For more information, refer to Basics on Model Referencing on page 413.

Another reason is that some TargetLink tools that are used during development, such as the Autoscaling tool, logging of minimum and maximum values, and overflow detection, do not fully work with referenced models.

Additionally, disabling a model reference lets you work with various versions of a function without the risk of losing its original state, which can be restored afterwards by restoring the model reference. For more information, refer to How to Restore Model References on page 445.

Precondition

The following precondition must be fulfilled before you can perform the subsequent instructions.

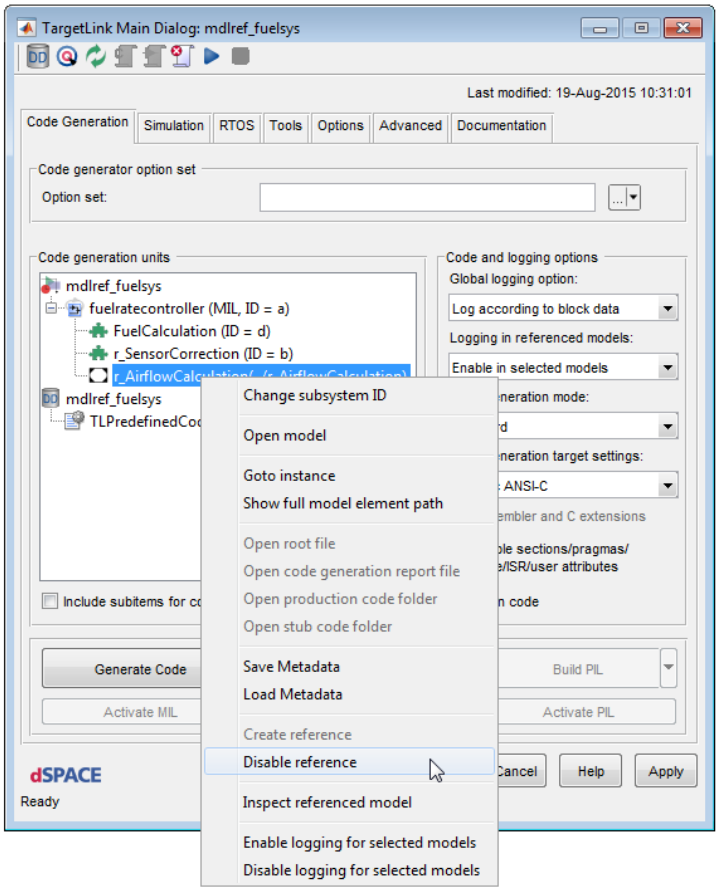
- The root model is open.

Method

To disable model references

1 Open the TargetLink Main Dialog.

2 In the Code generation units tree, select and right-click the referenced model whose reference you want to disable.

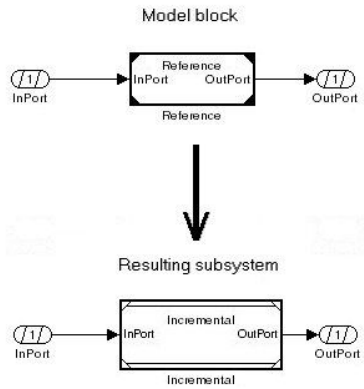


3 From the context menu, select Disable reference.

Result

The reference is disabled, i.e., the Model block that contains the reference to the model is replaced with a subsystem that has the same contents as the referenced model.

To indicate that the resulting subsystem was converted from a referenced model, its block is marked by additional lines that make it look similar to the Model block.



Note

When a referenced model is converted into a subsystem, the variables in its model workspace are not considered. They are not imported into the parent model's workspace.

Next steps

After you disabled the reference, you can model the function in the resulting subsystem. You can test and simulate the function.

After you completed function development, you have to restore the model reference. For more information, refer to How to Restore Model References on page 445.

Related topics

References

tl_refmodel_to_subsystem (TargetLink API Reference)

How to Restore Model References

Objective

After you completed function development, test and simulation, you have to restore the model reference.

Preconditions

The following preconditions must be fulfilled before you can perform the subsequent instructions.

- The model reference was disabled. For more information, refer to How to Disable Model References on page 442.
- The root model is open.

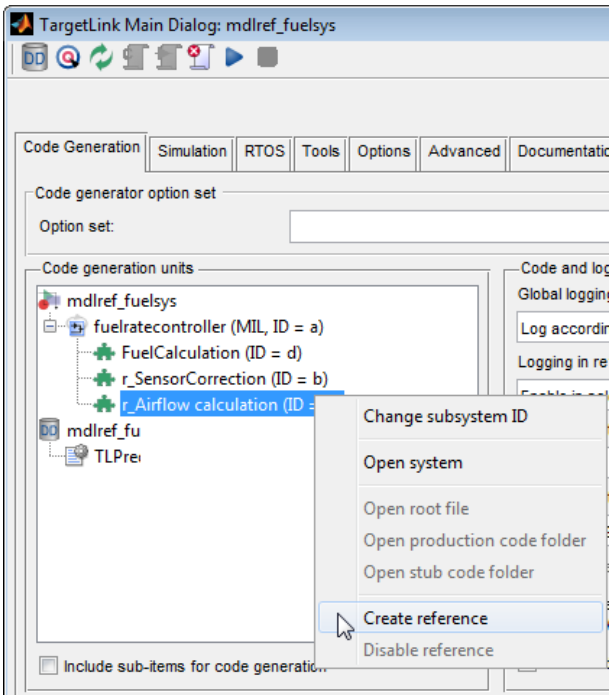
Method

To restore model references

1 Open the TargetLink Main Dialog.

2 In the Code generation units tree, select the subsystem that you want to restore the model reference for.

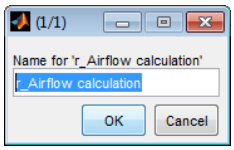
3 From the context menu, select Create reference.



4 In the Converting subsystem(s) to model(s) dialog that opens, click Use default names or Specify all names.

- If you select Use default names, the referenced model is named as the subsystem it is created from.

- If you select **Specify all names**, you can enter a name for each selected subsystem in the **Name for '<Model>'** field that is displayed.



5 You are asked if you want to use the existing model or update the existing model.
   - If you select **Update existing model**, the content of the referenced system is replaced with the content of the subsystem to be converted.
   - If you select **Use existing model**, the content of the subsystem to be converted is erased, and the content of the already existing (to be referenced) model is left unchanged. Thus, you can restore the original state.

Note

If the model does not exist, for example, because it was deleted after it was converted into a subsystem, a new model is created with the content of the subsystem.

Result

A model reference for the selected subsystem is restored. Depending on the selected **Use existing model** or **Update existing model** option, the referenced model remains unchanged or its content is updated with the content of the selected subsystem.

Tip

You can also use the **Create Reference** function to break down a complex model and convert its subsystems into referenced models. For more information, refer to How to Create Models with Model References by Adapting Existing Models on page 423.

Related topics

References

tl_subsystem_to_refmodel (TargetLink API Reference)

Defining Abstract Interfaces for Model-Components

Introduction

Support of component-based development allows you to separate the overall functionality into smaller software modules, called modular units which can be developed and tested independently. A separated modular unit can be tested with full test coverage more easily. Testing with full coverage is often not possible if the functionality is integrated in a larger context and therefore not all the test vectors can easily be applied directly at the modular unit. Thus the testability of modular units is generally higher. Later you only have to test the interaction between the modular units in the integration model for covering the overall functionality. Modular units also help you solve problems with long code generation times and high memory consumption during code generation for very large models, and can easily be reused without modification across different projects and in different contexts (integration model). This is an advantage if you want to change certain aspects of modular unit implementation: e.g., when you make a parameter of a modular unit a constant variable in one ECU project and a calibratable variable in another project. TargetLink provides abstract interfaces to support such changes without modifying the modular unit itself. You can map abstract interfaces to concrete implementations independently and after production code has been generated and tested for the modular unit (and should therefore not be changed).

Where to go from here

Information in this section

Basics on Component-Based Development Using Abstract Interfaces (RDI Objects) 447

Basics on Modeling Flexible Parameters Using RDI Objects 455

How to Model Flexible Parameters Using RDI Objects 456

How to Map RDI Objects to Specific Implementation Variables 460

Basics on Modeling Efficient Interfaces 462

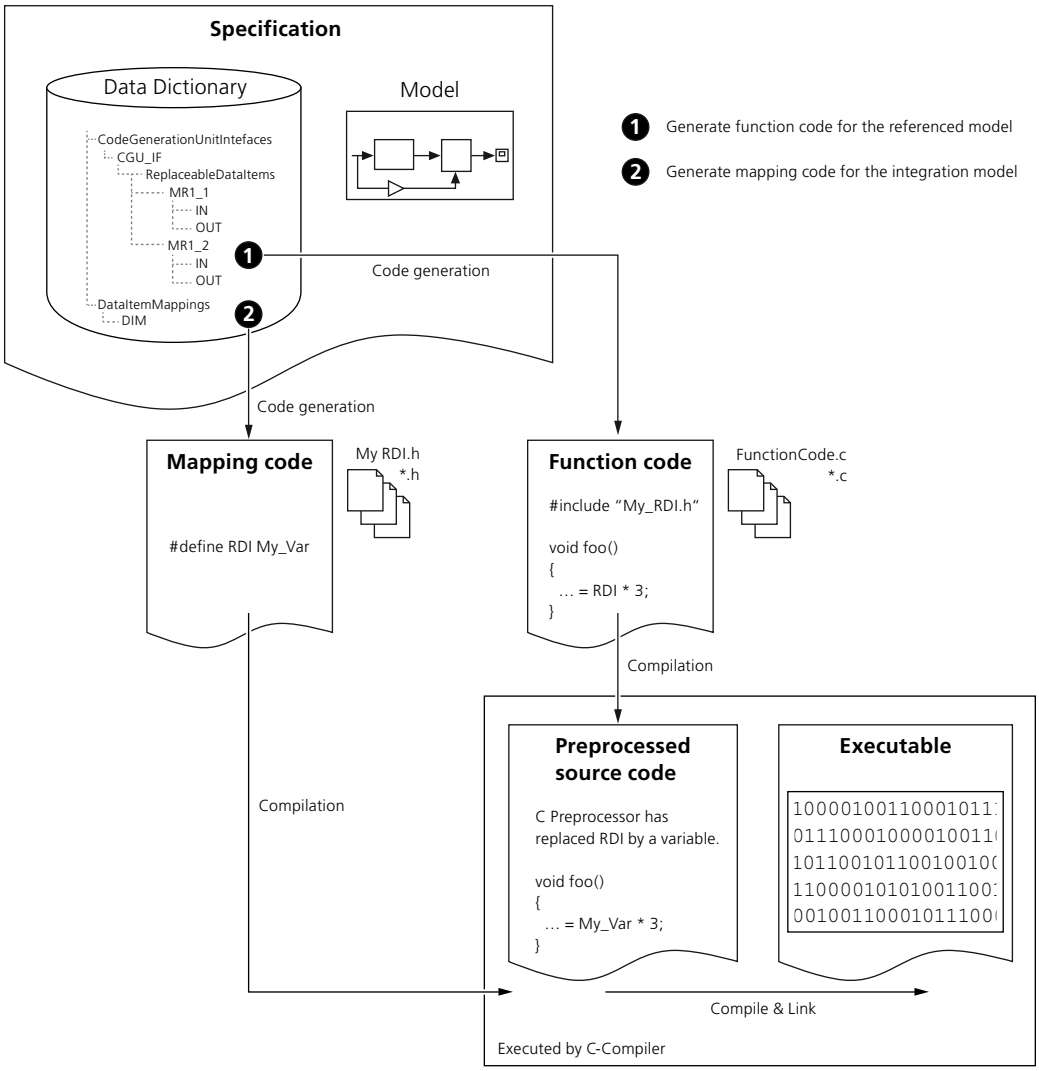
How to Model Efficient Interfaces Using RDI Objects 463

Basics on Component-Based Development Using Abstract Interfaces (RDI Objects)

Introduction

TargetLink can create abstract interfaces by using ReplaceableDataItem objects. Abstract interfaces allow you to map a project-specific, physical specification of an interface (made in the TargetLink Data Dictionary) to a logical interface. A

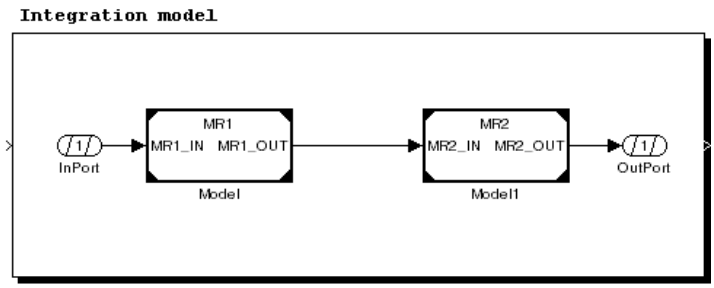
ReplaceableDataItem (RDI) object is a DD object that describes an abstract interface's basic properties such as the data type, scaling and width. RDI objects can be used at the borders of incrementally generated subsystems, referenced models, and TargetLink subsystems. They can be referenced in block dialogs the same way as DD variable objects. During code generation, RDI objects are generated as global macros that act as placeholders in the generated code. The code for mapping these global macros to implementation variables can then be generated later, either from an integration model or via DD-based code generation.



In this approach, the implementation is independently associated with the interface. Therefore, if the interface implementation context changes, you do not need to change the Simulink subsystem or the RDI object, and the generated production code for the modular unit can be reused without modification.

Integration model

To reuse a software module, called a modular unit below in different contexts, an integration model representing the surrounding system is used. The integration model contains existing modular units that were already tested, so code needs to be generated only for parts of the integration model. The mapping code is also generated from the integration model. The function code of the inner modular units does not have to be regenerated, so they do not need to be retested. For details on specifying function or mapping code, refer to How to Model Flexible Parameters Using RDI Objects on page 456 and to How to Model Efficient Interfaces Using RDI Objects on page 463.



Supported component-based interface types

For abstract interfaces using RDI objects, TargetLink supports two types called *flexible parameters* and *efficient interfaces*. Each interface type is intended for a specific use case as described below, and is specified in only a few simple steps. For flexible parameters, you have to specify at least an RDI and a corresponding DataItemMapping (DIM) object. For efficient interfaces, you have to specify only an RDI object. However, if you want to specify the mapping for efficient interfaces explicitly, you can use DIM objects anyway. Regardless of the interface type you use, you do not have to decide whether to implement an RDI object by, e.g., a global variable or a struct component (different ECU memory layout) immediately. You can decide this when integrating the modular unit in the integration model or when using DD-based code generation. Thus, you can generate the corresponding macro definition, i.e., the mapping code, without adapting the existing modular unit.

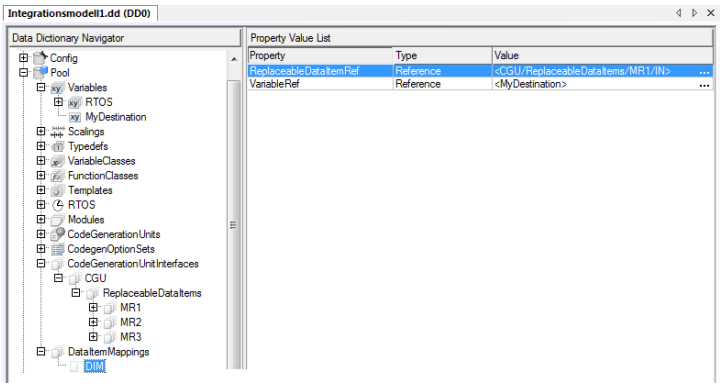
Note

- With distributed code generation, TargetLink might not have all the information required for code generation. If this happens, it will not be able to perform all the necessary checks. You are therefore responsible for ensuring that the generated code with the mapping functions correctly and also that the code can be compiled.
- RDI objects can be mapped only if all the variables involved have identical properties, for example, data type, scaling and width.

Flexible parameters

A flexible parameter is an abstract variable that lets you map arbitrary block parameters using RDI objects to specific implementations later. Typical use cases are modular units that are reused across several different projects with different

model parameters for each project. For this *explicit modeling*, you have to specify a RDI and, if you want to specify the mapping of the RDI object explicitly, a DataItemMapping (DIM) object in the TargetLink Data Dictionary. 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. You must specify the DIM object in the integration phase at the latest. You only have to specify it if the default implementation does not meet your requirements. The mapping of RDI objects to DIM objects must be unique and works only for **Pool** variables that are explicitly specified in the DD. This approach makes parameterization extremely flexible and ensures that the generated function code contains only one unique global macro for each RDI object (otherwise an RDI object might not be clearly mapped to a single implementation variable object for integration).



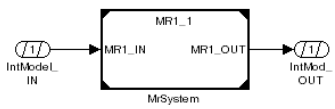
For details on creating flexible parameters, refer to How to Model Flexible Parameters Using RDI Objects on page 456.

Efficient interfaces

This type of interface lets you reduce the number of RAM variables that are usually required for the interface to referenced models and to subsystems configured for incremental code generation. It requires only little specification work in the TargetLink Data Dictionary, as only the inport and outport blocks must reference RDI objects. Specifying DIM objects is an optional not a mandatory task: i.e., you can specify them but you don't have to.

Example Additional RAM variables are usually created for the following model:

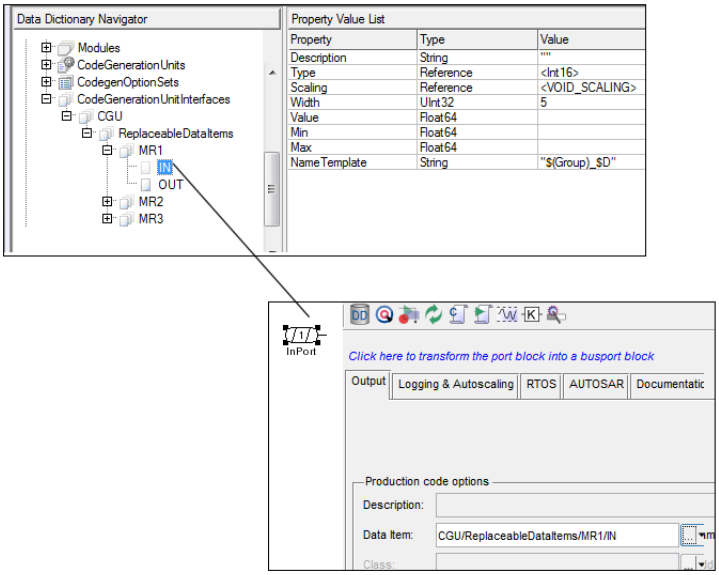
- **IntModel_IN**, **IntModel_OUT** for the integration model
- **MR1_IN** and **MR1_OUT** for the interface of the referenced model



Typically, before and after calling the referenced model, update assignments are performed as shown below.

```
MrSystem.c: (created when generating code for the referenced model)
   Int16 MR1_IN, MR1_OUT;
   void MrSystem()
   {
      … = MR1_IN;
      …
      MR1_OUT = …;
   }
TL_Subsystem.c: (created when generating code for the integration
model)
   #include "MrSystem.h"
   Int16 IntModel_IN, IntModel_OUT;
   foo()
   {
      MR1_IN = IntModel_IN;
      MrSystem();
      IntModel_OUT = MR1_OUT;
   }
```

To reduce the number of global variables and thus memory consumption, you can create MR1_IN and MR1_OUT as RDI objects and reference them at the respective inport and outport blocks inside the referenced model. This avoids RAM variables that are usually mandatory for the interface to referenced models and subsystems configured for incremental code generation. Under favorable conditions, the usual update assignments for the IntModel_IN and IntModel_OUT variables might even be eliminated by the compiler. However, if your variable mapping changes, you may need update assignments, which is why they are generated anyway. Thus, you can adapt the mapping code if required.

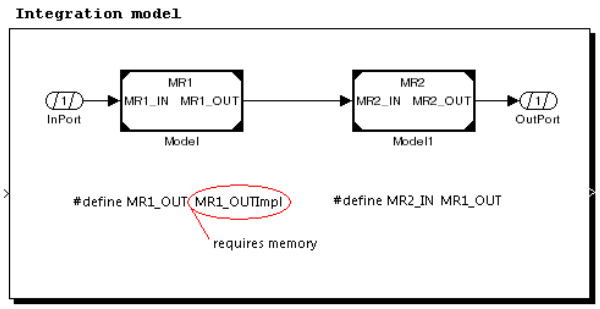


As a result, the generated code would look similar to the following:

```
MrSystem.c: (created when generating code for the referenced model)
   #include "tl_rdi_MrSystem.h"
   void MrSystem()
   {
      … = MR1_IN;
      ...
      MR1_OUT = …;
   }
tl_rdi_MrSystem.h: (created when generating code for the integration
model)
   #include "TL_Subsystem.h"
   #define MR1_IN IntModel_IN
   #define MR1_OUT IntModel_OUT
```

```
TL_Subsystem.c: (created when generating code for the integration
model)
   #include "MrSystem.h"
   #include "tl_rdi_MrSystem.h"
   Int16 IntModel_IN, IntModel_OUT;
   foo()
   {
      MR1_IN = IntModel_IN;
      MrSystem();
      IntModel_OUT = MR1_OUT;
   }
```

If you do not specify the mapping of the RDI objects via DIM objects explicitly, the Code Generator tries to automatically map inputs and outputs to existing global variables or components of a parameter structure (struct component) according to the modeled data flow (this is called implicit mapping). If no matching variables are found, new default implementation variables are created. Default implementation variables are indicated by their suffix *Impl* and can be further influenced by a template in the DD. In the model shown below, *MR1_OUTImpl* is the only variable that requires memory.



Details on RDI and DIM objects

As mentioned above, TargetLink provides two DD objects you can use to implement abstract interfaces. These are the ReplaceableDataItem (RDI) and DataItemMapping (DIM) objects.

ReplaceableDataItem objects You can group related RDI objects below the ReplaceableDataItemGroups object. Since RDI objects are always implemented as global macros, most properties of a variable class are not applicable, which is why RDI objects do not have a variable class property. This also implies that RDI objects themselves are not mergeable and therefore cannot be reused repeatedly in the model. Multiple RDI objects must also not be mapped to one implementation variable, i.e., an implementation variable must not be used by two RDI objects or by one RDI object and one block in the model, which would be the same as variable merging. Thus, in order to prevent unintentional repeated use of the implementation variable, you have to explicitly set the associated DD variable object to a variable class where the **Optimization** property is set to **MERGEABLE**. Since there can be only one global macro for each RDI object in the generated function code, you are not allowed to use name macros such as $B that cause new variables wherever an RDI object is referenced in a block. However, the **NameTemplate** property of an RDI object can contain certain name macros, for example, $D, $(Group) or $T. If you use other name macros, an error message is displayed during code generation.

If you use an RDI object with an empty **Value** property for a block variable that must be initialized, a warning occurs prompting you to set the **Value** property to obtain consistent simulation results.

Note

With distributed code generation, TargetLink might not have all the information required for code generation. If this happens, TargetLink will not be able to perform all the necessary checks. You are therefore responsible for ensuring that the generated code functions correctly and also that the code can be compiled. TargetLink does not check whether an RDI object used at a block output is mapped to a nonconstant variable. Finally, you must ensure that the implementation variable is initialized with the same value as the **Value** properties at the RDI object, and that the implementation variables are equal.

If you do not specify the mapping of the RDI objects via DIM objects explicitly and the Code Generator does not find a suitable variable, a default implementation variable is generated. The name of this implementation variable (`$(ReplaceableDataItem)Impl`) and the variable class are determined via a variable template. The name macro expands to the name of the RDI object to which the variable is mapped. By default, the default implementation variable is located in the module of the associated RDI object. Since TargetLink always generates an implementation variable (at least in stub code), your models can be compiled and simulated at any time. Consequently, you are not forced to map all the RDI objects to perform a simulation.

Limitation For the RDI objects themselves, TargetLink does not support struct types, struct components, bitfields, or pointers. If you assign one of these data types, an error message is displayed during code generation.

DataItemMapping objects You can group related DIM objects below the DataItemMappingGroups object. Specifying the mapping via a separate DIM object can be advantageous. An RDI object can be referenced by only one DIM object (via the **ReplaceableDataItemRef** property). Otherwise, an error

message is displayed. In addition, you cannot map an RDI object (via the **VariableRef** property of the DIM object) to a variable of struct or pointer type, but to global variables or components of global structures. If the **Scope** property of the associated variable class is not **global** or the **Storage** property is set to **static**, an error message is displayed. Nevertheless, a variable can be *static global*, if you use an access function template that references a global **FunctionClass**, e.g., GLOBAL_FCN. The **VariableRef** property supports code variants.

If the **ReplaceableDataItemRef** property and the **VariableRef** property are set at a DIM object, the following consistency checks apply: The **Width** and the scaling values (**LSB** and **Offset**) of the referenced objects must be identical, the **Value** and **BaseType** properties at the RDI object and the DD variable object must be identical, and the **Min/Max** values must lie within the specified **Min/Max** values of the implementation variable. The **Min/Max** values at the RDI object and at the implementation variable can be derived from the **Min/Max** values of a user-defined data type (UDT). If one or more of these requirements are not met, an error message is displayed.

Limitation RDI objects cannot be mapped (via the **VariableRef** property of the DiM object) to a variable of struct or pointer type, but to global variables or components of global structures.

Module-oriented code generation

Via the **ModuleRef** property of an RDI object, you can specify the module (C header file) containing the definitions of the global macros generated for the RDI objects. If you do not define a module, the default module **tl_replaceabledataitems** located in `/Pool/Modules/TLPredefinedModules` is used instead. You can adapt this default module directly. By default, the module name contains the $N name macro, for example, `tl_rdi_$N`, which generally ensures that the default module is unique for each code generation unit (CGU).

When you map an RDI object via a DIM object to a DD variable object, the Code Generator generates an implementation variable during code generation (except for extern and alias). If no **ModuleRef** property is specified for a DD variable object (either at the object itself or at its associated variable class) and if it is not used in the model, TargetLink creates the implementation variable in the same module as the RDI macro definition. If the DD variable object is used directly by the model, the rules for defining a variable in the production code apply.

Impact of ownership on code generation

To make good use of RDI objects, their definitions can be generated independently of the current modular unit. If the current modular unit is not the owner of the RDI default module, only stub code is generated for the module. Generating stub code is necessary because it allows you to compile and simulate a TargetLink subsystem. Referenced models and subsystems configured for incremental code generation are owned by the surrounding system which contains them. This makes simple integration scenarios possible. For example, you can integrate multiple modular units containing RDI objects in an integration model. During code generation for the modular units, the RDI objects are implemented as global macros and the default module is included. However,

when you start code generation for the integration model, production code is generated for the RDI default module containing the implementation variables you mapped and for the default implementation variables. You do not have to perform additional configuration for this. For more complex scenarios such as nested subsystems configured for incremental code generation or nested referenced models, you can use the configuration options provided by the component-based code generation feature. RDI macro definitions therefore can be generated purely from the Data Dictionary using the same means as for DD variable objects. For details on component-based code generation, refer to Basics on Generating Code from the Data Dictionary on page 397.

Related topics

Basics

REPLACEABLE_DATA_ITEMS (TargetLink Demo Models)

HowTos

How to Map RDI Objects to Specific Implementation Variables..... 460
How to Model Efficient Interfaces Using RDI Objects..... 463
How to Model Flexible Parameters Using RDI Objects..... 456

Basics on Modeling Flexible Parameters Using RDI Objects

Modular unit implementation

In different contexts you might want to change certain aspects of the modular unit implementation, e.g., when you make a parameter of a modular unit a constant variable in one ECU project and a calibratable variable in another project. You do not have to decide whether to implement an RDI object by, e.g., a global variable or a struct component (different ECU memory layout) immediately. You can decide this when integrating the modular unit in the integration model or when using DD-based code generation, i.e., independently and after production code has been generated and tested for the modular unit (and therefore will not be changed).

Mapping RDI objects to DIM objects

The mapping of RDI objects to DIM objects must be unique and works only for Pool variables that are specified in the DD explicitly. This approach makes parameterization extremely flexible and also ensures that the generated function code contains only one unique global macro for each RDI object (otherwise an RDI object could not be clearly mapped to a single implementation variable object for the integration context).

RDI objects can be mapped to either a global variable or a struct component via separate DIM objects at a later stage in the project process, even by another

developer or administrator. The entire process can therefore be divided into two process steps and a few preparation steps:

1. Creating Variable and RDI objects, refer to How to Model Flexible Parameters Using RDI Objects on page 456.
2. Mapping RDI objects to Variable objects via separate DIM objects when integrating the modular unit in an integration model, refer to How to Map RDI Objects to Specific Implementation Variables on page 460.

Related topics

HowTos

How to Model Flexible Parameters Using RDI Objects..... 456

How to Model Flexible Parameters Using RDI Objects

Objective

To map arbitrary block parameters to various implementations when integrating the function code in the integration model.

Demo model

The following instructions explain how to model flexible parameters based on the Sensor Correction and Fault Redundancy subsystem, which is part of the ReplaceableDataItems demo model located in `<DocumentsFolder>\dSPACE\TargetLink\<Version>\Demos\ReplaceableDataItems\ReplaceableDataItems.mdl`. The Throttle Prelookup block is used as an example showing the steps you have to perform.

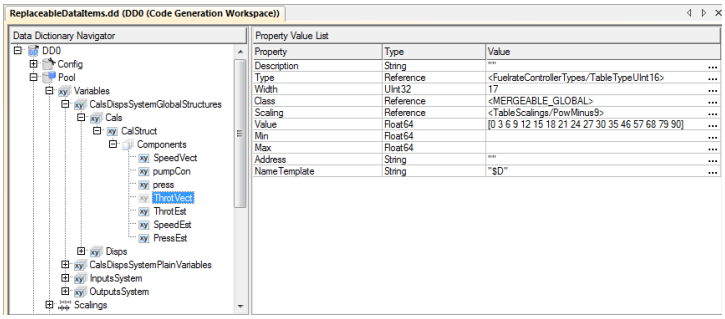
The ReplaceableDataItems demo model also shows how to implement a complete model via abstract interfaces. All abstract interface objects in the demo model are highlighted in blue.

Method

To model a flexible parameter

1. In the MATLAB Command Window, type **`tl_demos Replaceable_Data_Items`** to open the ReplaceableDataItems demo model.
2. In the Data Dictionary Manager, create global variables or struct components for all the parameters you want to assign to varying projects.

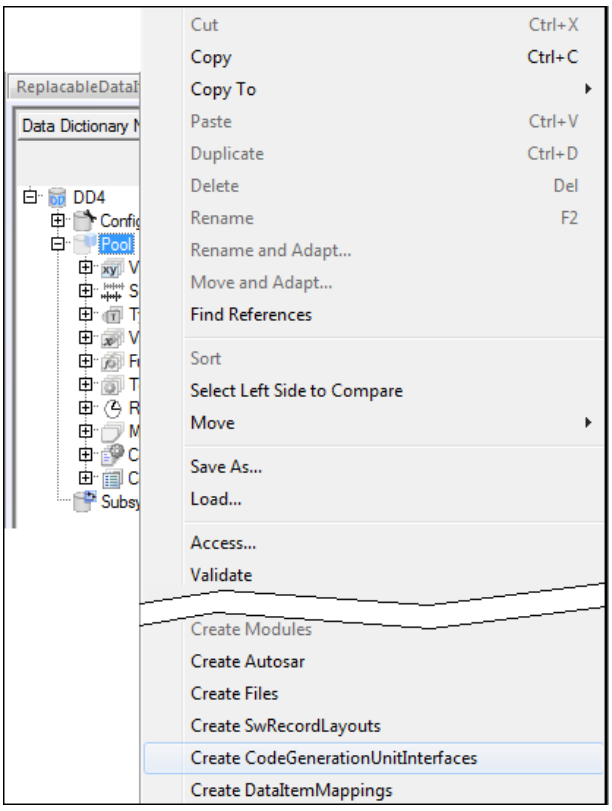
   If you are not familiar with how to create DD variable objects, refer to Example of Creating New Variables (TargetLink Data Dictionary Basic Concepts Guide).



Tip

If you have to create many DD objects such as Typedefs, Scalings and so on and these DD objects already exist in other projects, you can save existing objects or entire object groups as partial DDs. When you open a partial DD (Ctrl+O) in the current workspace of your *ReplaceableDataItem* model, its content is automatically inserted at the right place according to the metadata stored with the partial DD.

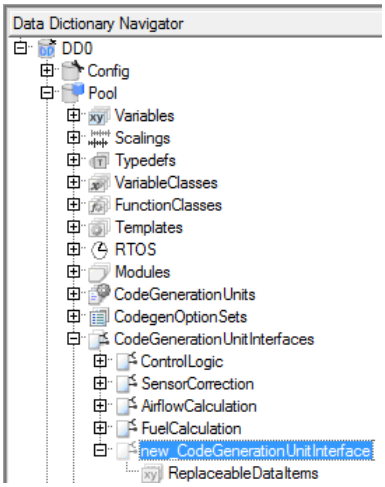
3 Right-click the Pool object and select Create CodeGenerationUnitInterfaces from the context menu.



A CodeGenerationUnitInterfaceGroup object named CodeGenerationUnitInterfaces is created.

4 Right-click CodeGenerationUnitInterfaces and select Create CodeGenerationUnitInterface from the context menu.

A CodeGenerationUnitInterface object named new_CodeGenerationUnitInterface and a ReplaceableDataItemGroup object named ReplaceableDataItems are created.



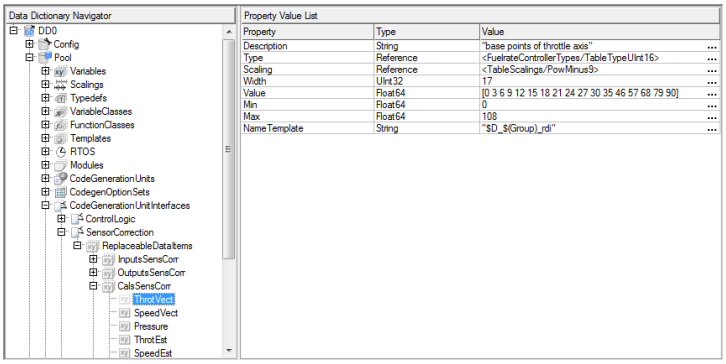
5 Name the new_CodeGenerationUnitInterface object you just created, for example, CGUInterface.

Tip

You can rename certain DD objects such as new_CodeGenerationUnitInterface. This might be useful if you want to adapt the DD object names to the model structure for better readability, or to group related DD objects.

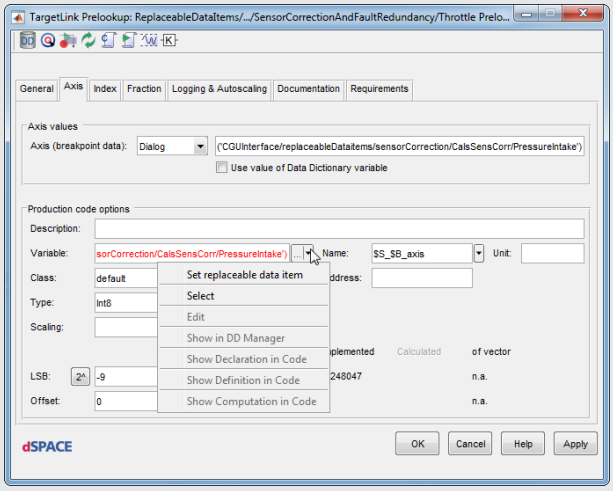
If you need to substructure your project more precisely, you can create group objects at certain DD objects.

6 Create substructures (Create ReplaceableDataItemGroup) and RDI objects (Create ReplaceableDataItem) as required, for example, as shown below.



Tip

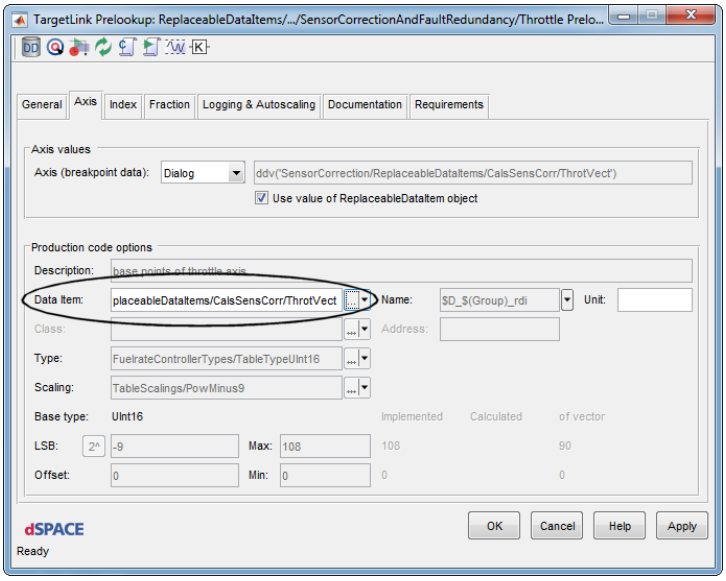
Alternatively, you can create RDI objects via the respective block dialog. To do so, enter an existing DD object path and the name of the object to be created in the Variable edit field. On the edit field, click the context menu button, select Set replaceable data item to create the RDI object and click Apply.



7 In the Sensor Correction and Fault Redundancy subsystem, double-click the Throttle Prelookup block and select the Axis page.

8 In the Production code options group box, click the Variable browse button and select the ThrotVect RDI object you created before.

The RDI object is set, which is why the Variable edit field is renamed to Data Item.



Result

You have created various DD objects and finally the RDI object ThrotVect. However, you have not mapped this RDI object to an implementation variable. This is done via DataItemMapping (DIM) objects and must be carried out later when you integrate the modular unit in the integration model. This approach makes parameterization extremely flexible as you can map each RDI object to different implementations.

Next Steps

You have several options at this point. You can create DataItemMapping (DIM) objects, or following the `ReplaceableDataItem` demo model, you can create other flexible parameters or efficient interfaces as shown by the example of the SensorCorrectionandFaultRedundancy subsystem. For details on modeling efficient interfaces to reduce the number of global RAM variables, refer to How to Model Efficient Interfaces Using RDI Objects on page 463. If you want to map an RDI object to an implementation variable, refer to How to Map RDI Objects to Specific Implementation Variables on page 460.

Related topics

Basics

Basics on Component-Based Development Using Abstract Interfaces (RDI Objects)..... 447

HowTos

Basics on Modeling Flexible Parameters Using RDI Objects..... 455
How to Map RDI Objects to Specific Implementation Variables..... 460
How to Model Efficient Interfaces Using RDI Objects..... 463

How to Map RDI Objects to Specific Implementation Variables

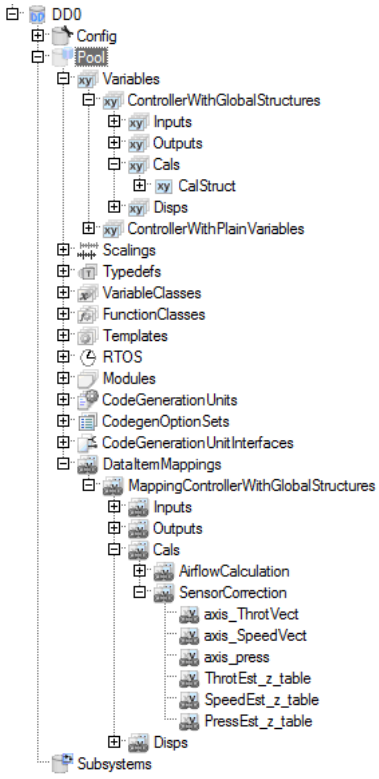
Objective

To map RDI objects to a specific implementation variable using a DataItemMapping (DIM) object.

Precondition

- You have opened the integration model and its associated DD file in the Data Dictionary Manager.
- You have created suitable DD variable objects and RDI objects. For details on how to create RDI objects, refer to How to Model Flexible Parameters Using RDI Objects on page 456.

If these preconditions are fulfilled, your DD could look similar to the following one.



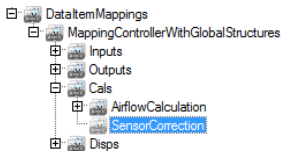
Method

To map an RDI object to a specific implementation variable

1 In the Data Dictionary Manager, right-click the Pool object and select Create DataItemMappings from the context menu.

A DataItemMappingGroup object named DataItemMappings is created.

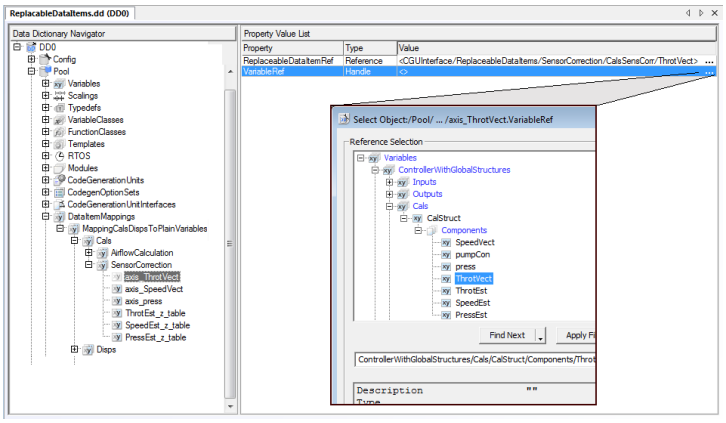
2 Create substructures using DataItemMappingGroup objects as required, for example, as shown below.



3 Right-click the SensorCorrection group object and select Create DataItemMapping from the context menu.

4 Name the DataItemMapping object you just created, for example, axis_ThrotVect.

5 In the **Property Value List**, click the browse buttons to reference the **ThrotVect** RDI and DD objects as shown below.



6 Click **OK** to close the **Select Object** dialog.

Result

You have mapped an RDI object to a specific implementation variable.

Related topics

Basics

Basics on Component-Based Development Using Abstract Interfaces (RDI Objects)..... 447

HowTos

How to Model Efficient Interfaces Using RDI Objects..... 463
How to Model Flexible Parameters Using RDI Objects..... 456

Basics on Modeling Efficient Interfaces

Additional RAM variables

Usually, additional RAM variables are generated for input and output variables at the interface between the integration model and referenced models or subsystems configured for incremental code generation. By referencing RDI objects at TargetLink InPort and OutPort blocks, you can reduce the number of variables used. Under favorable conditions, the usual update assignments for the RAM variables might even be eliminated by the compiler. However, if your variable mapping changes, you may need update assignments, which is why they are generated anyway. Thus, you can adapt the mapping code if required.

Related topics

HowTos

How to Model Efficient Interfaces Using RDI Objects..... 463

How to Model Efficient Interfaces Using RDI Objects

Objective

To model efficient interfaces you can reduce the number of the usually mandatory global RAM variables for modular units and thus memory consumption by using ReplaceableDataItem (RDI) objects.

Demo model

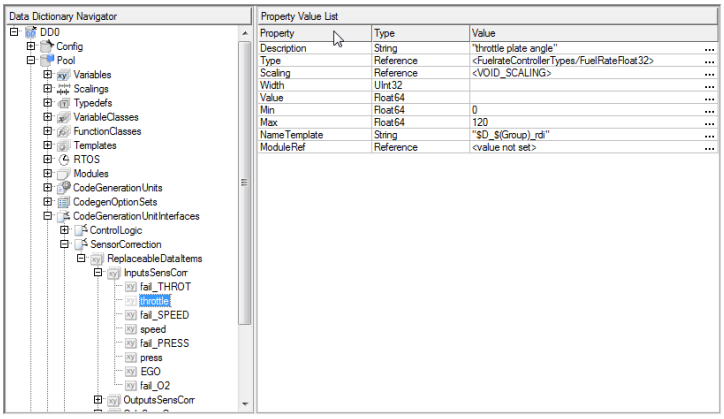
The following instructions explain how to model an efficient interface based on the Sensor Correction and Fault Redundancy subsystem, which is part of the ReplaceableDataItems demo model located in `<DocumentsFolder>\dSPACE\TargetLink\<Version>\Demos\ReplaceableDataItems\ReplaceableDataItems.mdl`. The Throttle Prelookup block is used as an example showing the steps you have to perform.

The ReplaceableDataItems demo mode also shows how to implement a complete model via abstract interfaces. All abstract interface objects in the demo model are highlighted in blue.

Method

To model an efficient interface

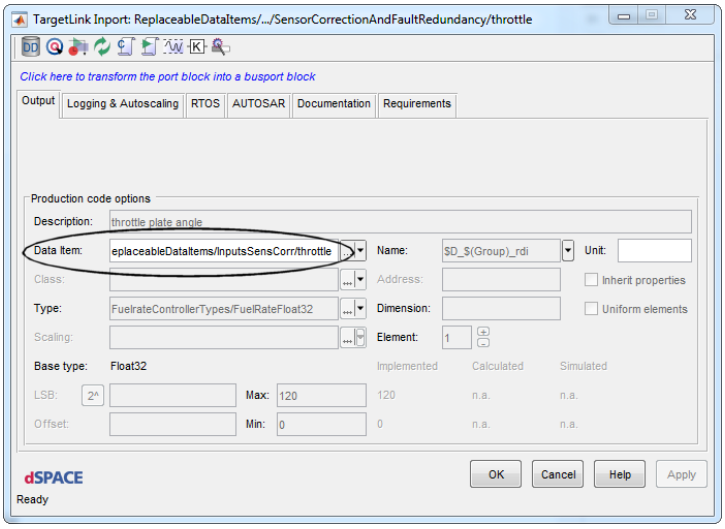
1. In the MATLAB Command Window, type **`tl_demos Replaceable_Data_Items`** to open the ReplaceableDataItems demo model.
2. In the Data Dictionary Manager, create RDI group objects to substructure your project more precisely and RDI objects for the inports and outports you want to reference.



3 In the SensorCorrectionandFaultRedundancy subsystem, double-click the throttle InPort block.

4 In the Production code options group box, click the Variable browse button, select the throttle RDI object you created before and click OK to close the dialog.

The RDI object is set, which is why the Variable edit field is renamed to Data Item.



Result

You have referenced an RDI object at a TargetLink InPort block. The Code Generator tries to automatically map inputs and outputs to existing global variables or components of a parameter structure (struct component) according to the modeled data flow (this is called implicit mapping). If no matching variables are found, new default implementation variables are created. If you want to specify an implementation variable explicitly, you can do this using DataItemMapping (DIM) objects.

Next steps

You have several options at this point. You can create further RDI objects and reference them at the respective inports and outports, or you can map RDI objects to specific implementation variables. This is done via DataItemMapping (DIM) objects and must be carried out only when you integrate the modular unit in the integration model. This approach makes parameterization extremely flexible as you can map each RDI object to different implementations. For details, refer to How to Map RDI Objects to Specific Implementation Variables on page 460.

Related topics

Basics

Basics on Component-Based Development Using Abstract Interfaces (RDI Objects)..... 447
Basics on Modeling Efficient Interfaces..... 462

HowTos

How to Map RDI Objects to Specific Implementation Variables..... 460
How to Model Flexible Parameters Using RDI Objects..... 456

Customizing the Representation of Buses in Production Code

Introduction

To work with buses and let TargetLink generate C structs for them in the production code.

Where to go from here

Information in this section

Mapping Buses and Subbuses to Implicit Structs in the Code..... 468

Mapping Entire Buses to Explicit Structs in the Code..... 479

Specifying Bus Elements Individually..... 491

Mapping Buses and Subbuses to Implicit Structs in the Code

Introduction

You can have TargetLink generate C structs for (sub)buses in the generated code. Hence, TargetLink implicitly derives the representation of bus signals in the generated code from the model specification.

Where to go from here

Information in this section

Basics on Implicit Structure Variables Resulting from Buses..... 468

How to Let TargetLink Create Implicit Structs for Buses..... 469

Examples of Implicit Structs for Nested Buses..... 471

Basics on Sort Sequences of Bus Elements in Implicit Structure Variables..... 475

How to Specify the Sort Criteria of Bus Elements..... 476

Basics on Implicit Structure Variables Resulting from Buses

Specification of production code options

There are production code options which are not applicable to structs resulting from buses, for example, the **Scaling** or **Unit** option. These options are therefore disabled for (sub)buses (nodes) in the bus port dialog. You can specify the following properties for implicit structs resulting from buses:

Property	Description
Description	Lets you enter a description of the struct.
Name	Lets you enter a name for the struct.
Class	Lets you specify one of the variable classes as defined in the current Data Dictionary project file. For more information, refer to Details on Specifying Variable Classes for Structs Representing Buses (📖 TargetLink Preparation and Simulation Guide).
Type	Lets you specify the data type of the variable in the generated C code. For implicit C structs, you can specify only a struct base type which does not have any components because the components are created by the Code Generator following the sort criteria you specified. Refer to How to Specify the Sort Criteria of Bus Elements on page 476. If you set 'CreateTypedef' to 'On', you can specify a synonym for the type. This renaming mechanism allows you to omit the `struct` keyword in the variable's definition. For details on data types, refer to Basics on TargetLink Base Types, Typedefs, and Header Files on page 128 and to How to Rename Base Types on page 136.

Related topics

Basics

Basics on the Representation of Buses in the Production Code (📖 TargetLink Preparation and Simulation Guide)
Details on Specifying Variable Classes for Structs Representing Buses (📖 TargetLink Preparation and Simulation Guide)

HowTos

How to Rename Base Types..... 136
How to Specify the Sort Criteria of Bus Elements..... 476

References

Bus Inport Block (📖 TargetLink Model Element Reference)
Bus Outport Block (📖 TargetLink Model Element Reference)

How to Let TargetLink Create Implicit Structs for Buses

Objective

To have TargetLink create implicit structure variables for buses and subbuses.

Suitable blocks

Creating implicit C structs for (sub)buses is possible with TargetLink's bus port blocks:

- Bus Inport
- Bus Outport

Precondition

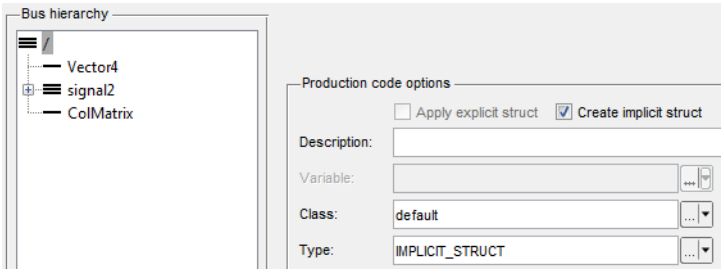
A model containing a bus connected to a bus port block is open.

Method

To let TargetLink create implicit structs for buses

1. Double-click the bus port block to open its dialog. Go to the Output page.
2. In the Bus hierarchy, select the (sub)bus you want TargetLink to generate a struct for.

3 In the **Production code options** group box, select the **Create implicit struct** checkbox.



The production code options become active and the **Type** parameter is automatically set to the empty (i.e, without components) **IMPLICIT_STRUCT** DD **Typedef** struct. However, you can also reference any other empty DD **Typedef** struct.

4 In the **Production code options** group box, specify the bus' production code options.

Production code options that are not applicable to structs resulting from buses, are disabled.

5 If the bus contains a subbus, you have to perform the above steps also for the subbus, if you want TargetLink to generate a struct for it: i.e., a substruct of the struct.

Result

During production code generation, TargetLink generates a (sub)struct for the (sub)bus.

TargetLink uses the same structured type, if the generated implicit structs have the same number of components, the same component names, and the same component types, or the same empty DD **Typedef** struct is used and the buses match. This applies only to structs created from bus signals within the following boundaries:

- TargetLink subsystem
- Subsystem configured for incremental code generation
- External subsystem
- Reused subsystem

For example, if you want TargetLink to generate an implicit struct from a bus residing at the TargetLink subsystem, and from another bus residing at an subsystem for incremental code generation, TargetLink does not use the same structured type for the generated structs if the default **IMPLICIT_STRUCT** type is used. For details, refer to the following sections on using explicit structs, or use a DD **Typedef** struct.

Related topics

HowTos

How to Specify the Sort Criteria of Bus Elements..... 476

Examples

Examples of Implicit Structs for Nested Buses..... 471

References

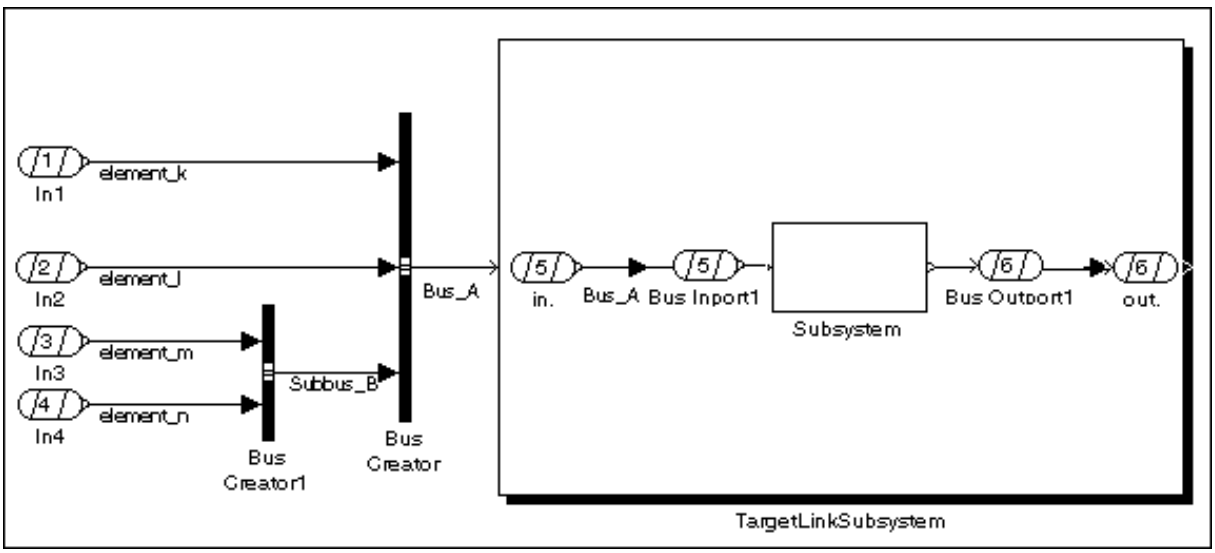
Bus Inport Block (📖 TargetLink Model Element Reference)
Bus Outport Block (📖 TargetLink Model Element Reference)

Examples of Implicit Structs for Nested Buses

Mapping nested buses to the code

TargetLink supports any level of nested buses. There are different ways of mapping nested buses to C structs in the production code, depending on the setting of the **Create implicit struct** checkbox and the nesting depth. The **Apply explicit struct** checkbox must always be cleared.

The following example model contains the `Bus_A` bus and the `Subbus_B` subbus.

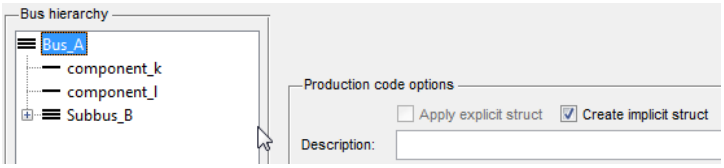


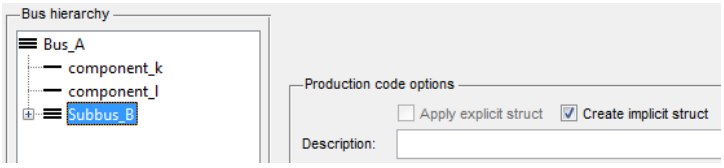
Mapping the bus as is to implicit structs

Suppose you want TargetLink to generate the following variables for the bus and its bus elements:

- Struct variables for `Bus_A` and `Subbus_B`
- 16-bit integer variables for `element_k` and `element_l`
- 32-bit integer variables for `element_m` and `element_n`

Production code settings The illustrations below show the required settings of the Create implicit struct checkbox for `Bus_A` and `Subbus_B` you have to specify on the Output page of the Bus Inport dialog. The checkbox is selected for both buses.





The following table shows the production code settings of the bus elements you have to specify:

Bus Element	Type
`element_k`	Int16
`element_l`	Int16
`element_m`	Int32
`element_n`	Int32

Resulting production code The Code Generator generates the following definition for the struct's type:

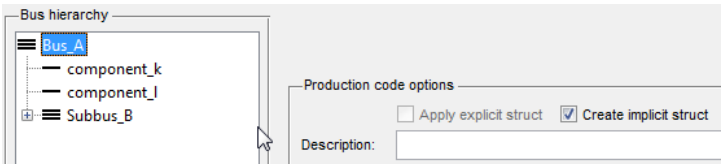
```
struct BSS_IP_Sa1_Bus_Inport_tp {
   Int32 element_m;
   Int32 element_n;
}; /* bus inport struct */
struct BS_IP_Sa1_Bus_Inport_tp {
   Int16 element_k;
   Int16 element_l;
   struct BSS_IP_Sa1_Bus_Inport_tp Subbus_B;
}; /* bus inport struct */
struct BS_IP_Sa1_Bus_Inport_tp Bus_A;
```

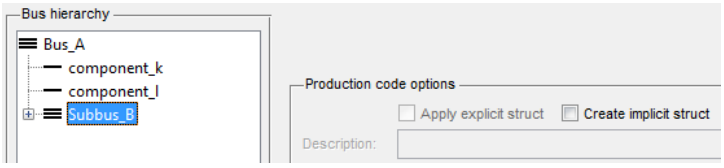
Mapping only the bus to an implicit struct

Suppose you want TargetLink to generate the following variables for the bus and its bus elements:

- Struct variable for `Bus_A`
- 16-bit integer variables for `element_k` and `element_l`
- 32-bit integer variables for `element_m` and `element_n`

Production code settings The illustrations below show the required settings of the Create implicit struct checkbox for `Bus_A` and `Subbus_B` you have to specify on the Output page of the Bus Inport dialog. The checkbox is cleared for `Subbus_B`.





The following table shows the production code settings of the bus elements you have to specify:

Bus Element	Type
`element_k`	Int16
`element_l`	Int16
`element_m`	Int32
`element_n`	Int32

Resulting production code The Code Generator generates the following definition for the struct's type:

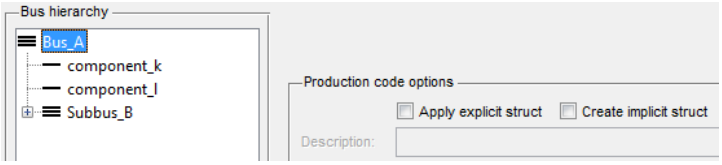
```
struct BS_IP_Sa1_Bus_Inport_tp {
   Int32 element_m;
   Int32 element_n;
   Int16 element_k;
   Int16 element_l;
}; /* bus inport struct */
struct BS_IP_Sa1_Bus_Inport_tp Bus_A
```

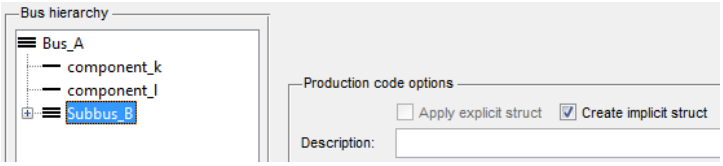
Mapping only the subbus to an implicit struct

Suppose you want TargetLink to generate the following variables for the bus nodes and bus elements:

- Struct variable for Subbus_B
- 16-bit integer variables for element_k and element_l
- 32-bit integer variables for element_m and element_n

Production code settings The illustrations below show the required settings of the Create implicit struct checkbox for Bus_A and Subbus_B you have to specify on the Output page of the Bus Inport dialog. The checkbox is selected only for Subbus_B.





The following table shows the production code settings of the bus elements you have to specify:

Bus Element	Type
element_k	Int16
element_l	Int16
element_m	Int32
element_n	Int32

Resulting production code The Code Generator generates the following definition for the struct's type:

```
Int16 element_k;
Int16 element_l;
struct BS_IP_Sa1_Bus_Inport_tp {
   Int32 component_m;
   Int32 component_n;
}; /* bus inport struct */
struct BS_IP_Sa1_Bus_Inport_tp Subbus_B;
```

Separate variables for the leaf bus elements

If the Create implicit struct checkbox is cleared for Bus_A and Subbus_B, the Code Generator does not create any implicit structs but creates separate variables for all four leaf bus elements.

Refer to Specifying Bus Elements Individually on page 491.

Related topics

HowTos

How to Let TargetLink Create Implicit Structs for Buses..... 469

Basics on Sort Sequences of Bus Elements in Implicit Structure Variables

Sort sequence for implicitly generated structs

The components' sort sequence for implicitly generated structs is specified in the same way as for dynamic struct components. For details, refer to Example of Collecting Gain Parameters via Dynamic Components on page 161.

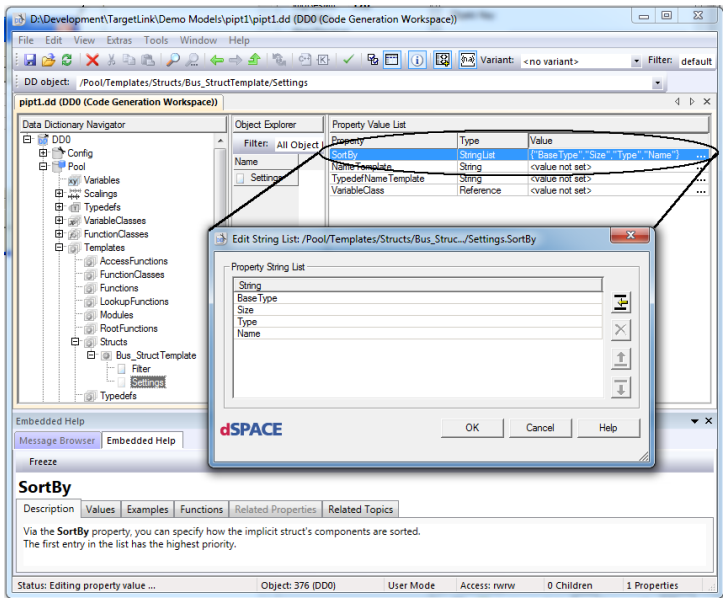
Sorting rules

You can specify the sorting rules the Code Generator follows when mapping bus elements to implicit structure components in the production code. You can sort the struct components by the following criteria:

- Size
- Base Type
- Type
- Name

Sort criteria in the TargetLink Data Dictionary

You can specify the sort criteria in the TargetLink Data Dictionary Manager.



For details, refer to How to Specify the Sort Criteria of Bus Elements on page 476.

Related topics

HowTos

How to Specify the Sort Criteria of Bus Elements..... 476

Examples

Example of Collecting Gain Parameters via Dynamic Components..... 161

How to Specify the Sort Criteria of Bus Elements

Objective

To specify the order of bus elements.

Suitable blocks

Specifying the order of bus elements is possible with TargetLink's bus port blocks:

- Bus Inport
- Bus Outport

Preconditions

- The desired DD project file (<name>.dd) is open.
- The appropriate bus port dialog is open.

Method

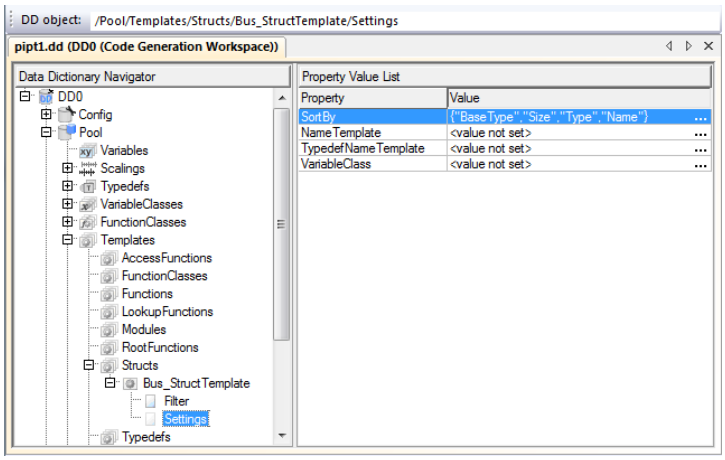
To specify the sort criteria of bus elements

1. In the /Pool/Templates area of the Data Dictionary Navigator, right-click Structs to open the context menu.
2. Select Create StructTemplate.

   A new subnode for the new structure template appears.
3. In the new subnode, enter the name for the new structure template, for example, enter `Bus_StructTemplate`.
4. Expand the new struct template and change to the Filter subnode.
5. In the property value list, click the StructSpec browse button to open the Edit String dialog.
6. In the String value edit field, enter **`BusInportStruct`** and click OK.

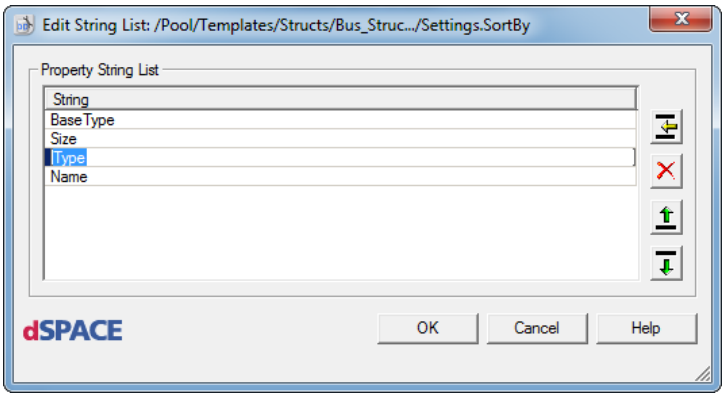
   For details on possible values for structs resulting from buses, refer to Bus-Related StructSpec Strings.

7 In the Data Dictionary Navigator, switch to the **Settings** child object of the struct template.



8 In the **property value list**, click the **SortBy** browse button.

The **String Array** dialog opens for you to enter and arrange the sort criteria of the struct components.

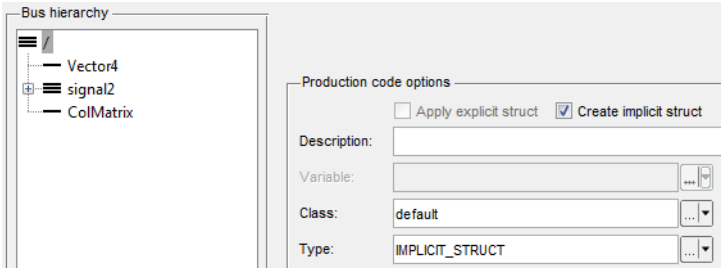


9 Enter the desired sort criteria and arrange them using the up and down buttons. For details, refer to SortBy.

10 Click **OK** to close the dialog.

11 Change to the **Output** page of the **BusInport** dialog and select a bus in the **bus hierarchy**.

12 Make sure that the **Create implicit struct** checkbox is selected for the bus.



Result

TargetLink generates a struct during code generation and arranges the bus elements according to the sort criteria you specified.

Related topics

Basics

Basics on Sort Sequences of Bus Elements in Implicit Structure Variables..... 475

References

Bus Inport Block (📖 TargetLink Model Element Reference)
Bus Outport Block (📖 TargetLink Model Element Reference)

Mapping Entire Buses to Explicit Structs in the Code

Introduction

You can create a predefined struct in the TargetLink Data Dictionary (DD), and assign a complete bus to this DD struct. This is especially helpful if the bus consists of many bus elements, and/or the type or variable is used several times in a model or across models.

Where to go from here

Information in this section

Basics on the Compatibility of Buses and Predefined Structs..... 479

How to Map Entire Buses to Predefined Structure Variables..... 483

How to Map Entire Buses to Predefined Structure Type Definitions..... 485

Example of Mapping Bus Elements to a Custom C Struct..... 487

Basics on the Compatibility of Buses and Predefined Structs

Introduction

To inform you about the compatibility requirements when applying predefined struct variables and type definitions (explicit structs). Predefinition is possible in the **TargetLink Data Dictionary** by using DD **Typedef** objects whose **BaseType** is set to **Struct**, or DD **Variable** objects referencing such a (structured) DD **Typedef** object. When you reference a structured **Typedef**, all its struct components must be specified explicitly, which is why TargetLink does not provide an IMPLICIT_STRUCT base type here. However, for DD **Variable** objects, TargetLink supports struct types without struct components.

Compatibility rules and requirements

Compatibility rules and requirements are:

- If a DD **Typedef** object whose **BaseType** is set to **Struct** or a DD **Variable** object referencing a structured **Typedef**, is assigned to a bus signal, TargetLink checks at code generation time whether the first bus element is compatible with the first struct component. Then it checks whether the second bus element is compatible with the second struct component, and so on. Compatibility is achieved if a component's signal dimension matches the signal dimension of the counterpart bus element, for example, scalar to scalar, or vector to vector.

  For details on the compatibility of signal dimensions, refer to Dimension reduction (📖 TargetLink Preparation and Simulation Guide).

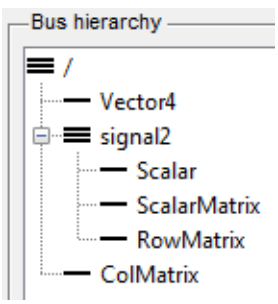
The number of struct components must match the number of bus elements.

Note

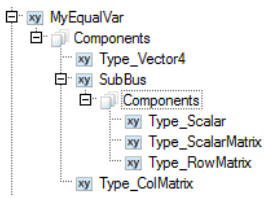
The `sldatatypestr` Simulink data type must be specified for the bus. Otherwise, TargetLink cannot perform the compatibility checks mentioned above.

- The internal hierarchy of structs is not taken into account. Only the elements/components are considered in the order of their occurrence in the bus or struct, respectively.
- The structure of the struct variable created for the bus elements is determined by the structure of the referenced **Typedef** object. The name and data type properties provided by the structured typedef are assigned to the associated bus elements. If the data type also provides constraints (**Constraints** object), such as constrained limits (**Max**, **Min**), fixed-point scaling parameters or a **Unit**, then these properties are taken into account. If no fixed-point scaling is specified, TargetLink uses the default scaling and a warning message occurs.
- If **Type** is a struct (**BaseType=Struct**), you can select a unique variable class for all the struct type's components via the **Comp. class** browse button.
- The following requirement applies only to Stateflow data: If the signal's data type is `Simulink.Bus`, then the signal must be assigned to either a completely specified DD **Variable** object or to a DD **Type** object that is a completely specified struct type. In the latter case, you can use the **Comp. class** parameter to set the data type for all the leaf struct components. An error message occurs, if neither a structured DD **Variable** object nor a structured DD **Type** object is specified. If the signal's data type is not `Simulink.Bus`, it must not be assigned to a structured DD **Variable** object or a structured DD **Type** object. If such a signal is assigned to a DD object nonetheless, an error message occurs.

Checking compatibility Suppose a bus consists of five bus elements as shown in the table below. The bus element names reflect the elements' signal dimensions, for example, vector [4].

Bus Signal	⇒ Bus Elements
	1. Vector [4] 2. Scalar 3. Scalar matrix [1x1] 4. Row matrix [1xn] 5. Column matrix [mx1]

Suppose a DD **Variable** object (BaseType=Struct) consists of five components as shown in the table below. The component names reflect the components' signal dimensions, for example, vector [4].

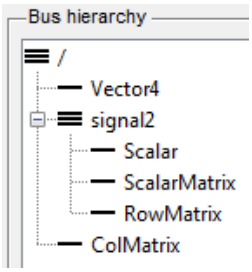
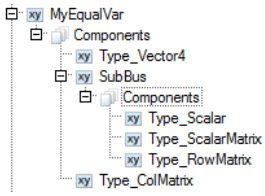
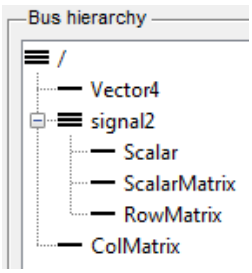
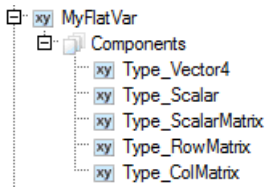
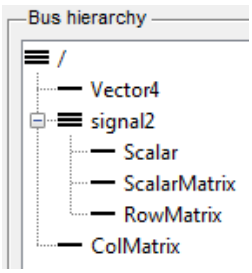
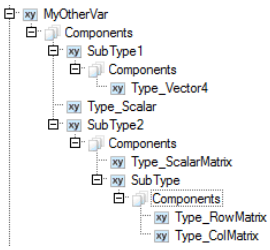
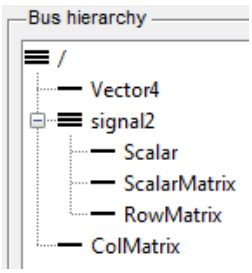
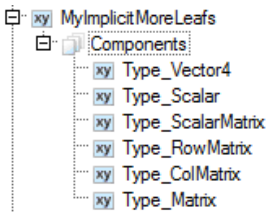
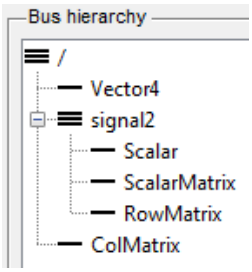
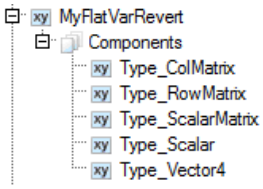
DD Variable Object	⇒ Struct Components
	1. Vector [4] 2. Scalar 3. Scalar matrix [1x1] 4. Row matrix [1xn] 5. Column matrix [mx1]

The bus and the DD **Variable** object are compatible, because each bus element/struct component pair is compatible (one-to-one correspondence).

Bus Elements	Struct Components
1. Vector [4] 2. Scalar 3. Scalar matrix [1x1] 4. Row matrix [1xn] 5. Column matrix [mx1]	1. Vector [4] 2. Scalar 3. Scalar matrix [1x1] 4. Row matrix [1xn] 5. Column matrix [mx1]

Compatibility examples

The following table shows examples of matching buses and DD **Variable** objects (BaseType=Struct). The bus element/struct component names reflect the elements'/components' signal dimensions, for example, vector [4].

Compatible?	Bus Signal	DD Variable Object
YES Bus exactly matches struct.		
YES Leaf bus elements match struct components (number and order). Internal structure is irrelevant.		
YES Leaf bus elements match struct components (number and order). Internal structure is irrelevant.		
NO Different number of leaf bus elements versus struct components, e.g., 5 versus 6		
NO Different signal dimensions, e.g., vector [4] versus column matrix		

Related topics

Basics

Basics on Code Generation for MATLAB® Code (TargetLink Code Generation Guide for MATLAB® Code in Simulink® Models)
Basics on the Representation of Buses in the Production Code (TargetLink Preparation and Simulation Guide)

HowTos

How to Map Entire Buses to Predefined Structure Type Definitions..... 485
How to Map Entire Buses to Predefined Structure Variables..... 483

How to Map Entire Buses to Predefined Structure Variables

Objective

To map buses to DD Variable objects based on a one-to-one correspondence between bus elements (in the model) and struct components (in the code).

Suitable blocks

Mapping entire buses to predefined structure variables is possible with these TargetLink blocks:

- TargetLink bus port blocks:
  Bus Inport, Bus Outport
- TargetLink bus-capable blocks.

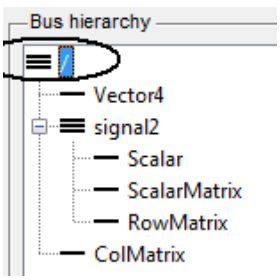
Precondition

The C struct is defined as a DD Variable object in the TargetLink Data Dictionary. It matches the number and the order of the bus elements.

Method

To map an entire bus to a predefined structure variable

1. In the model, double-click the bus port block to open its dialog. Go to the Output page.
2. In the Bus hierarchy, select the bus.



3 In the Production code options, select the Apply explicit struct checkbox.

Note

This option is available only for the bus root.

4 For the Variable parameter of the bus, reference the DD Variable object.

All other production code options are now set as specified by the DD Variable object and are read-only.

If the DD Variable object is not compatible with the bus, an error message is displayed in the block dialog.

5 Click OK to close the block dialog.

Result

You mapped the entire bus to a predefined structure variable. After code generation, you can find the definition of the explicit structure variable and the definition of the user-defined data type in the generated production code, for example:

Type definition

```
struct SubType_Equal_tag {
   UInt16 Type_Scalar;
   Int32 Type_ScalarMatrix[1][1];
   Float32 Type_RowMatrix[1][4];
};
typedef struct Type_Equal_Struct_tag {
   Int8 Type_Vector4[4];
   struct SubType_Equal_tag SubBus;
   Int16 Type_ColMatrix[4][1];
} Type_Equal_Struct;
```

Variable definition

```
Type_Equal_Struct MyEqualVar;
```

Related topics

Basics

Basics on the Compatibility of Buses and Predefined Structs..... 479

HowTos

How to Create New Data Dictionary Objects (TargetLink Data Dictionary Basic Concepts Guide)

References

Bus Inport Block (TargetLink Model Element Reference)
Bus Outport Block (TargetLink Model Element Reference)
Constant Block (TargetLink Model Element Reference)
Data Store Memory Block (TargetLink Model Element Reference)
Merge Block (TargetLink Model Element Reference)
Multiport Switch Block (TargetLink Model Element Reference)
Switch Block (TargetLink Model Element Reference)
Unit Delay Block (TargetLink Model Element Reference)

How to Map Entire Buses to Predefined Structure Type Definitions

Objective

To map buses to DD **Typedef** objects based on a one-to-one correspondence between bus elements (in the model) and struct components (in the code).

Suitable blocks

Mapping entire buses to predefined structure type definitions is possible with these TargetLink blocks:

- TargetLink bus port blocks:
  Bus Inport, **Bus Outport**
- TargetLink bus-capable blocks

Precondition

The type definition is defined as a DD **Typedef** object whose **BaseType** property is set to **Struct**.

- The DD **Typedef** object matches the number and the order of the bus elements.
- For integer data types, each **Typedef** struct component references a DD **Typedef** object whose **Constraints** property references a DD **Scaling** object.

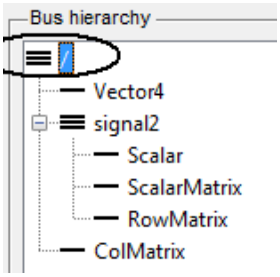
Method

To map an entire bus to a predefined structure type definition

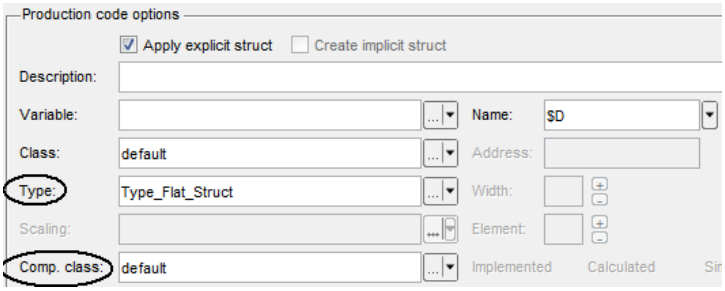
1. In the model, double-click the bus port or bus-capable block to open its dialog. Go to the Output page.
2. If the block is a bus port block, select the bus in the Bus hierarchy and select the Apply explicit struct checkbox in the Production code options group box.

   Note

   The Apply explicit struct option is available only for the bus.

3. For the Type parameter of the bus, reference the DD Typedef object whose BaseType property is set to Struct.

   The Comp. class parameter is displayed below the Type parameter.

   If the DD Typedef object is not compatible with the bus, an error message is displayed in the block dialog.
4. Specify the Comp. class parameter.

   This class is assigned to all bus elements.
5. Click OK to close the block dialog.

Result

You mapped the entire bus to a predefined type definition structure. After code generation, you can find the definition of the structure variable and the definition of the user-defined data type in the generated production code, for example:

Type definition

```
struct SubType_Equal_tag {
   UInt16 Type_Scalar;
   Int32 Type_ScalarMatrix[1][1];
   Float32 Type_RowMatrix[1][4];
};
typedef struct Type_Equal_Struct_tag {
   Int8 Type_Vector4[4];
   struct SubType_Equal_tag SubBus;
   Int16 Type_ColMatrix[4][1];
} Type_Equal_Struct;
```

Variable definition

```
Type_Equal_Struct DD_Sa1_InPort;
```

Related topics

Basics

Basics on the Compatibility of Buses and Predefined Structs..... 479

HowTos

How to Create New Data Dictionary Objects (📖 TargetLink Data Dictionary Basic Concepts Guide)

References

Bus Inport Block (📖 TargetLink Model Element Reference)
Bus Outport Block (📖 TargetLink Model Element Reference)
Constant Block (📖 TargetLink Model Element Reference)
Custom Code Block (📖 TargetLink Model Element Reference)
Data Store Memory Block (📖 TargetLink Model Element Reference)
Delay Block (📖 TargetLink Model Element Reference)
Merge Block (📖 TargetLink Model Element Reference)
Multiport Switch Block (📖 TargetLink Model Element Reference)
Switch Block (📖 TargetLink Model Element Reference)
Unit Delay Block (📖 TargetLink Model Element Reference)

Example of Mapping Bus Elements to a Custom C Struct

Introduction

Suppose you have external code, and this code contains a C struct which is specified as pointer-to-struct.

External code

The external code comprises the `AddConst.h` and `AddConst.c` modules:

AddConst.h

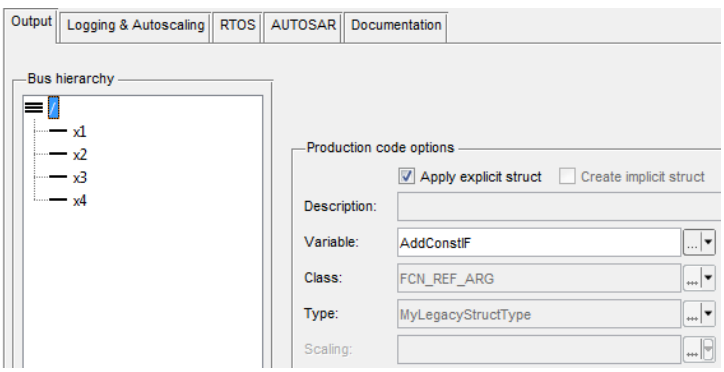
```
#ifndef ADDCONST_H
#define ADDCONST_H

/**
* definition of the struct type
*/
typedef struct MyLegacyStructType_tag {
   signed char Sig1;        /* Int8*/
   signed char Sig2[2];     /* Int8 */
   signed short int Sig3;   /* Int16 */
   signed char Sig4;        /* Int8 */
} MyLegacyStructType;


/**
* declaration of the function
*/
signed short int AddConstFcn(MyLegacyStructType * pMyStruct);

#endif
```

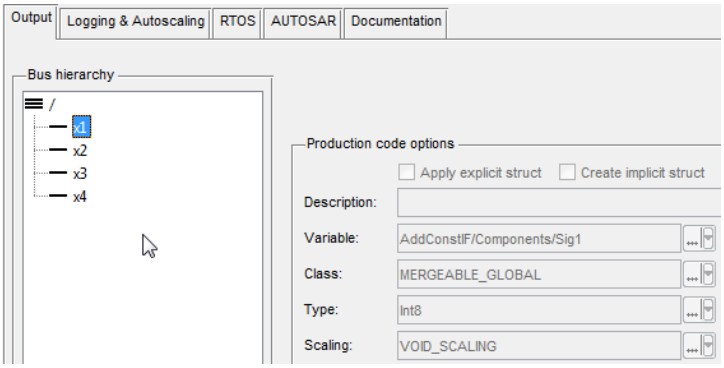
AddConst.c

```
#include "AddConst.h"

/**
 * definition of the function
*/
signed short int AddConstFcn(MyLegacyStructType * pMyStruct)
{
    return pMyStruct->Sig1 +
    pMyStruct->Sig2[0] +
    pMyStruct->Sig2[1] +
    pMyStruct->Sig3 +
    pMyStruct->Sig4;
}
```

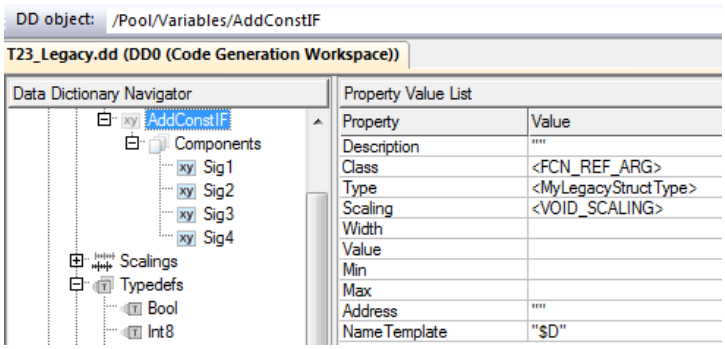
Model/DD specification

Model A bus (four leaf bus elements) is fed to the Root TargetLink subsystem. The bus is processed in the MyF subsystem. The bus and its leaf bus elements (e.g., x1) are specified as follows:

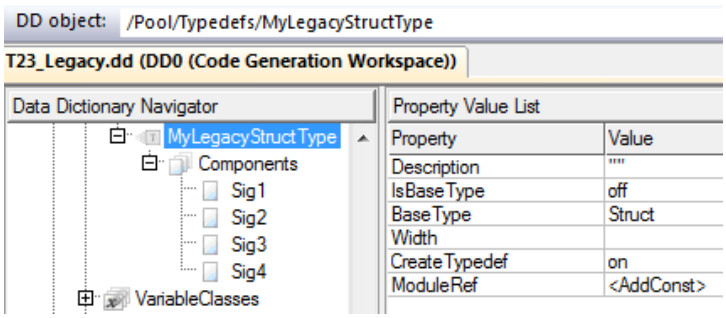




Data Dictionary The `AddConstIF` DD Variable object is specified as follows:



The `MyLegacyStructType` DD Typedef object is specified as follows:



Generated code

TargetLink would generate the following code:

Root.h

```
...
#include "AddConst.h"
...
```

Root.c

```
…
void Root(MyLegacyStructType * AddConstIF)
{
   /* call of function: Root/MyF */
   AddConstFcn(AddConstIF);
}
…
```

Related topics

References

Bus Inport Block (📖 TargetLink Model Element Reference)
Bus Outport Block (📖 TargetLink Model Element Reference)

Specifying Bus Elements Individually

Introduction

If none of the previously described alternatives are applicable, you can manually specify the implementation variables of individual bus elements.

When using individual specification, you have full control over the generated code. It is especially useful if you need to map bus elements to members of structures that do not match the internal structure of the bus. In this case you can still map the bus elements individually to the structure's components.

Where to go from here

Information in this section

How to Create Separate Variables for Leaf Bus Elements..... 491

Details on Mapping Bus Elements to Function Arguments..... 492

Example of Leaf Bus Elements Specified Individually..... 493

How to Create Separate Variables for Leaf Bus Elements

Objective

To create a separate variable for each leaf bus element.

Suitable blocks

Creating a separate variable for each leaf bus element is possible with TargetLink's bus port blocks:

- Bus Inport
- Bus Outport

Precondition

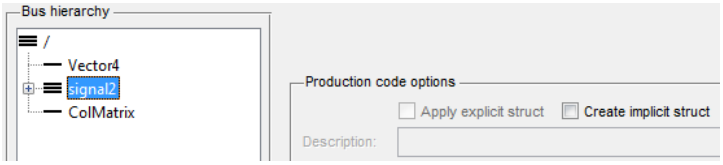
A model containing a bus connected to a bus port block is open.

Method

To create separate variables for bus elements

1. In your model, double-click the bus port block to open its dialog. Go to the Output page.
2. In the bus hierarchy, select the bus or subbus whose leaf bus elements you want TargetLink to generate separate variables for.

The substructure of the bus is displayed.



3 Make sure that the **Create implicit struct** checkbox (and also the **Apply explicit struct** checkbox) is cleared for the selected (sub)bus and for any parent node including the top root.

   The bus' production code options are disabled.

4 In the **bus hierarchy**, select a leaf bus element.

5 In the **Production code options** group box, specify the leaf bus element's production code options, or reference a DD **Variable** object.

6 Repeat steps 4 and 5 for the remaining leaf bus elements.

Note

If the bus contains a subbus and you want TargetLink to create separate variables for its leaf bus elements, you have to perform the above steps also for the subbus. Refer to Example of Leaf Bus Elements Specified Individually on page 493.

Result

You have specified the leaf bus elements individually.

Related topics

Examples

Example of Leaf Bus Elements Specified Individually..... 493

References

Bus Inport Block (📖 TargetLink Model Element Reference)
Bus Outport Block (📖 TargetLink Model Element Reference)

Details on Mapping Bus Elements to Function Arguments

Order of function arguments

If you use the **Function** block to generate code for a subsystem in a separate C function, you can specify the order of the function arguments. Refer to the **Arguments** page. If the elements of a bus result in several function arguments, you can specify their positions in the argument list only blockwise. The sequence in that block is the same as the sequence in the bus hierarchy.

Related topics

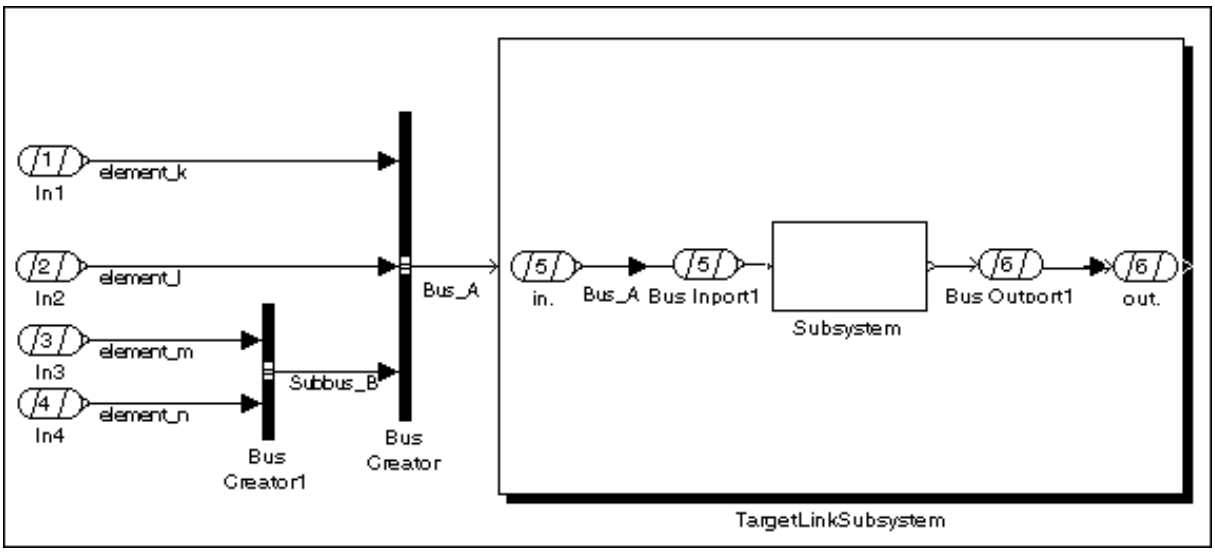
References

Function Block (📖 TargetLink Model Element Reference)

Example of Leaf Bus Elements Specified Individually

Example model

Consider the following example model containing the `Bus_A` bus and the `Subbus_B` subbus.

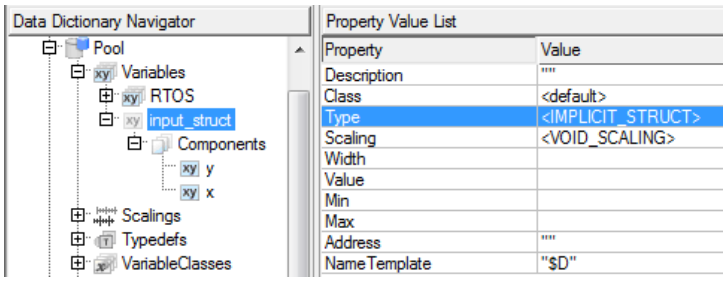


Production code settings

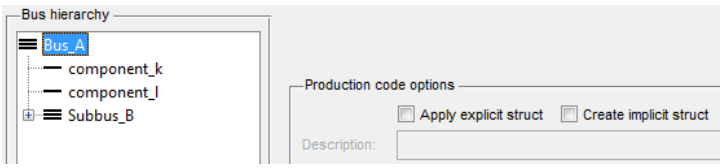
Suppose you want TargetLink to generate the following variables for the leaf bus elements:

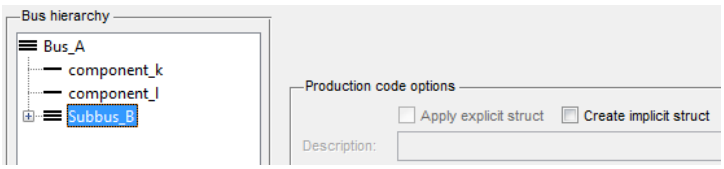
- `element_k`: `x` component (16-bit integer variable) of `input_struct`
- `element_l`: 16-bit integer variable
- `element_m`: `y` component (32-bit integer variable) of `input_struct`
- `element_n`: 32-bit integer variable

The following illustration shows the input_struct variable object in the TargetLink Data Dictionary. Its Type references IMPLICIT_STRUCT.

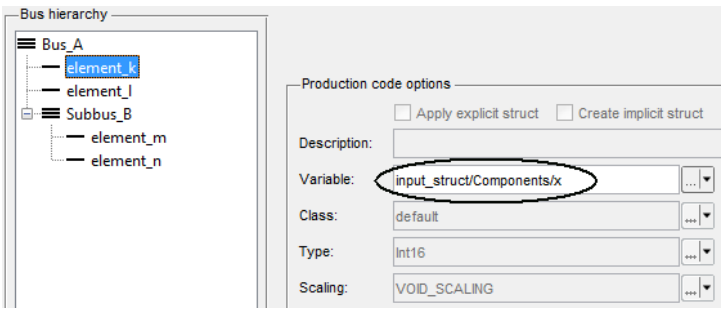


The following illustration shows the settings of the Apply explicit struct and Create implicit struct checkboxes for Bus_A and Subbus_B which you can specify on the Output page of the BusInport dialog. These checkboxes are cleared.





The following illustration shows the settings of element_k on the Output page of the BusInport dialog. It references the x struct component. Analogously, element_m references the y struct component.



The following table shows the production code settings of all the bus elements you have to specify:

Bus Element	Variable	Type
`element_k`	`input_struct/Components/x`	(int16)
`element_l`	—	int16

Bus Element	Variable	Type
element_m	input_struct/Components/y	(int32)
element_n	—	int32

Resulting production code

With the above settings, the Code Generator generates the **x** and **y** variables in the production code as components of **input_struct**:

```
struct tag_input_struct {
   Int32 y;
   Int16 x;
}; /* struct type created for variable: input_struct */
```

Related topics

HowTos

How to Create Separate Variables for Leaf Bus Elements..... 491

References

Bus Inport Block (📖 TargetLink Model Element Reference)
Bus Outport Block (📖 TargetLink Model Element Reference)

Optimizing the Production Code

Introduction

After you build your model, either from scratch or by converting a Simulink subsystem into a TargetLink subsystem, you can optimize the generated production code.

Where to go from here

Information in this section

Introduction to Optimizing Production Code 498
Optimizing Variables 504
Optimizing Functions 524
Optimizing Block-Specific Code 534
Optimizing Target-Specific Code 536
Optimizing Control Flows 543
Optimizing via Range Propagation 551
Optimizing Bitfields for Booleans and Stateflow States 555

Introduction to Optimizing Production Code

Introduction

TargetLink provides various methods to improve the production code's execution time, stack consumption and code size. These methods are used by the Code Generator when it generates and compiles the production code for the current TargetLink subsystem. The methods are implemented as algorithms that analyze the data and control flow and use code patterns especially adapted to your target.

Where to go from here

Information in this section

Basics on Optimizing the Production Code..... 498

How to Enable Code Optimization..... 502

Basics on Optimizing the Production Code

Defining optimization

Optimization is a process by which the efficiency and performance of generated code is improved, for example, by eliminating unnecessary computations. These improvements mainly aim at minimizing runtime and memory consumption. Thus, already highly efficient code becomes even better and faster due to optimization efforts.

Optimizations are basically applied to:

- Variables
- Functions
- Function interfaces

Basic views for optimization

Depending on whether you look at optimization from the perspective of a design engineer or a software developer, two basic views can be distinguished:

- Model view (block diagrams, charts)
- Code view (generated C code)

Optimizations take place for both views. The conditions for these optimizations may differ depending on the respective view.

Optimization methods

TargetLink provides useful optimization methods like the elimination or moving of code to reduce for example the production code's execution time, stack consumption and code size. These methods are used by the Code Generator when it generates and compiles the production code for the current TargetLink

subsystem. The methods are implemented as algorithms that analyze the data and control flow and use code patterns especially adapted to your target.

Usually TargetLink performs optimizations that try to reduce RAM / stack consumption, ROM size, and execution time all at once. There may, however, be situations where the optimization of one or two of these factors has negative effects on the other factors. The decrease of execution time may, for example, result in a higher RAM or stack consumption. Such a trade-off is always influenced by the context surrounding the pattern to be optimized. It is, for example, important whether the optimizable pattern is situated inside or outside a loop. In some situations a trade-off is hard to evaluate, which is shown by the following example:

```
b = f(a1, a2);
for (i = 0; i < 10; ++i) {
....
if (cond) {
   d = g(b, c);
.... }
 .... }
```

If TargetLink cannot determine that `cond` is always false or that it is true for only one iteration of the loop, the `if` branch may be executed more than once. The elimination of the intermediate variable `b` may result in multiple calls to `f()`, which can cause a much higher execution time.

Specifying optimization methods

Many optimization methods are performed internally without you having a chance of specifying them. There are, however, also optimization methods which you can specify. You can do so by either enabling or disabling Code Generator options provided by TargetLink or by defining **Optimization** property values for variable classes or function classes in the **TargetLink Data Dictionary**. The Code Generator options as well as the **Optimization** properties both enable you to determine which optimization methods are to be performed and which not. You can thus directly influence the optimization process.

The interaction between the Code Generator options and the Optimization properties depends on the respective use case. In some cases a Code Generator option may overrule the setting of an **Optimization** property, in other cases they may influence each other. For detailed information on each available Code Generator option and its interaction with **Optimization** properties, refer to Code Generator Options.

Accessing optimization methods

The optimization methods provided by TargetLink can be accessed in different ways:

- Via the **Advanced** page of the **TargetLink Main Dialog**

  By clicking **All Options** on the **Advanced** page you get access to Code Generator options. The options listed below **Optimization** allow you to specify optimization methods performed by TargetLink.
- Via **Optimization** properties in the **TargetLink Data Dictionary**

In the **Data Dictionary Manager** you can define various **Optimization** property values for variable classes or function classes to determine optimization methods for the selected class.

Note

- Code Generator options always affect optimization globally. **Optimization** property values, however, can be defined differently for variable classes and function classes, thus allowing you to control optimizations more specifically.
- Optimizations do not intend to change the model or simulation behavior. Sometimes, however, there are exceptions, for example, if user specifications override the analyses results of the Code Generator.

Optimization option

Before the final production code for the current subsystem is generated, the Code Generator carries out a detailed analysis on the internal intermediate code representation of the TargetLink subsystem. The results of this analysis are used to perform code optimizations like the elimination of variables or the simplification of logical structures.

By default, optimization is enabled. To switch it off, clear the **Generate optimized code** checkbox found on the **Advanced** page of the **TargetLink Main Dialog**.

Option	Optimization
off	No optimization is performed.
on	Complete optimization, all possible optimizations are performed: ▪ Code structures are simplified. ▪ Block outputs are computed only if necessary. ▪ Superfluous block output variables are eliminated, for example, logical and relational blocks are combined in single Boolean expressions. Appropriate code comments are generated. ▪ Superfluous auxiliary variables are eliminated.

Weakly atomic subsystems

Simulink can treat subsystems as atomic units so that all inputs are calculated before the subsystem is executed. Then the subsystem and at lastly the subsequent blocks are calculated. The behavior of subsystem execution is comparable with a function call in C. TargetLink starts by generating contiguous code for atomic subsystems, i.e., non-optimized code does not contain interleaving code from blocks outside the respective subsystem.

There are two types of atomic subsystems:

- Strictly atomic, where the optimized code does not interleave with code of other subsystems or blocks.
- Weakly atomic, where the optimized code can interleave with code of other subsystems or blocks.

Conditionally executed subsystems, such as enabled and triggered subsystems, are always atomic. The **Treat as atomic unit** checkbox is unavailable, so the

specification cannot be changed. TargetLink treats these subsystems as weakly atomic subsystems.

Subsystems, whose **Treat as atomic unit** checkbox you can select and subsystems that contain a function block can be treated as weakly or strictly atomic subsystems:

- Strictly atomic if you set the **AllowInterleavingCodeForAllSubsystems** Code Generator option to **off**.
- Weakly atomic if you set the **AllowInterleavingCodeForAllSubsystems** Code Generator option to **on**.

For more information on atomic subsystems, refer to Basics on Using Subsystems on page 226.

Note

Use strictly atomic subsystems for the following reasons:

- The generated code of strictly atomic subsystems is easier to match to the Simulink source.
- Without interleaving code, you can ensure execution orders or other workarounds.

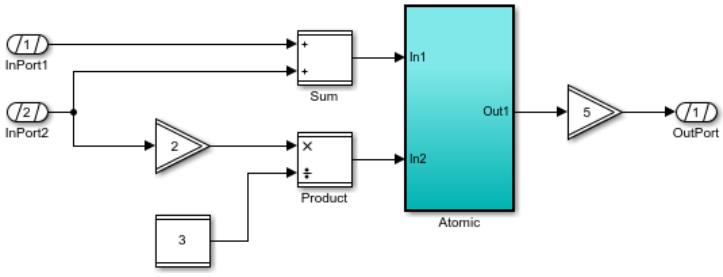
To achieve optimizations similar to strictly atomic subsystems, you can make the following settings:

- Use a function block with a function class whose optimization property contains neither **CG_AUTO_INLINE** nor **CG_FORCE_INLINE**. The subsystem is executed as a separate function.
- Assign a variable class whose optimization property contains neither **ERASABLE** nor **MOVABLE** to the inport and outport blocks of the subsystem, and deactivate the **CombineControlFlowStatements** Code Generator option.

.

Example for a weakly atomic subsystem

Assume you built a model such as the following:



Atomic contains only a **Sum** Block called **InnerSum**. If the **AllowInterleavingCodeForAllSubsystems** Code Generator option is set to **off**, **Atomic** is treated as strictly atomic. The calculations start with **Sum** and **Product** followed by **Atomic**. In the last step, **Outport** is calculated.

```
{
   Float32 InnerSum;
   Float32 Product;
   Float32 Sum;

   Sum = InPort1 + InPort2;

   Product = (InPort2 * 2.F) / 3.F;
   InnerSum = Sum + Product;

   OutPort = InnerSum * 5.F;
}
```

Atomic is treated as weakly atomic if the AllowInterleavingCodeForAllSubsystems Code Generator option is set to on. The calculations can be performed at the same time:

```
{
   OutPort = (InPort1 + InPort2 + ((InPort2 * 2.F) / 3.F)) * 5.F;
}
```

Note

Intermediate variables are eliminated only if this does not change the behavior. Therefore, without optimization weakly atomic have the same production code as atomic subsystems, but weakly atomic subsystems have more optimization options than atomic subsystems.

Related topics

Basics

Basics on Using Subsystems 226

HowTos

How to Enable Code Optimization 502

References

Code Generator Options (📖 TargetLink Model Element Reference)
TargetLink Main Dialog Block (📖 TargetLink Model Element Reference)

How to Enable Code Optimization

Objective

To let TargetLink carry out code optimizations such as elimination of intermediate variables, dead code elimination or scope reduction.

Method

To generate optimized code

1. In the TargetLink Main Dialog, change to the Advanced page.
2. In the Advanced production code options group box, select the Generate optimized code checkbox to activate the code optimization process.

Result

You have now enabled the code optimization option.

Tip

- If you want to identify particular code sections in your generated code clear the Generate optimized code checkbox.
- Regardless of the selected optimization, optimizations can be determined via the following Code Generator options:
  - GlobalOptimizationIterationThreshold which specifies how often optimization iterates at maximum over the whole generated code.
  - OptimizationIterationThreshold which specifies how often optimization iterates at maximum over a function.

Related topics

Basics

Basics on Optimizing the Production Code 498

References

Code Generator Options (📖 TargetLink Model Element Reference)
GlobalOptimizationIterationThreshold (📖 TargetLink Model Element Reference)
OptimizationIterationThreshold (📖 TargetLink Model Element Reference)

Optimizing Variables

Where to go from here

Information in this section

Basics on Optimizing Variables..... 504
How to Optimize Variables..... 512
Basics on the Scope Reduction of Variables..... 513
How to Specify the Scope Reduction of Explicit Variables..... 515
Basics on Eliminating Temporary Variables..... 518
Basics on Optimizing Struct Variables..... 521

Basics on Optimizing Variables

Optimization property

TargetLink provides an efficient method for optimizing variables. The variable classes stored in the **`Pool/VariableClasses`** folder in the TargetLink Data Dictionary contain an Optimization property which lets you specify property values for determining the optimization of variables.

Supported property values for variable optimization

The following property values are supported for the Optimization property of variable classes:

- MOVABLE
- ERASABLE
- SCOPE_REDUCIBLE
- MERGEABLE

For the Optimization property to become effective, you must activate optimization. Refer to Basics on Optimizing the Production Code on page 498. The default variable class usually allows these optimizations.

Note

If you set the **Optimization** property to MOVABLE, ERASABLE, or SCOPE_REDUCIBLE, you guarantee that there are no unknown variable accesses:

- There are no variables accesses outside of the CGU.
- There are no variables accesses from external functions.

To enable TargetLink to move value assignments into conditionally executed control flow branches, eliminate variable accesses or unused variables, and reduce visibility/storage duration, respectively.

MOVABLE

Determines that the Code Generator is allowed to move a variable or the block which is used to calculate the variable into a conditional control flow. This means that variables are moved into conditionally executed control flow branches based on the data flow of the block diagram.

If you want to move variables, be aware of the following restrictions:

- Blocks can be moved only if all the block output variables are set to **movable**.
- The Code Generator moves parts of code into or out of atomic subsystems only if the following conditions are fulfilled:
  - Only the propagation of variables and constants is affected.
  - The MATLAB/Simulink option **Treat as Atomic Unit** is enabled.
  - The subsystem contains a **Function** block with a step function class that meets the following conditions:
    - The **Optimization** property value **cg_force_inline** is set.
    - The function class causes the inlining of the step function.

  If the above restrictions do not apply, calculations can be performed in conditionally executed control flow branches.

  For information on weakly atomic subsystems, refer to Basics on Optimizing the Production Code on page 498.

Code Generator options affecting moving The moving of variables is influenced by the following Code Generator options:

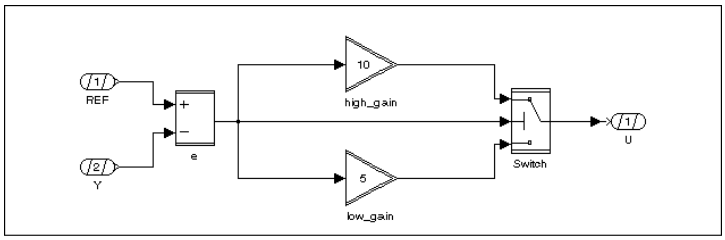
- **DoNotMoveIfLogged**
- **EnableBlockDiagramBasedSwitchOptimization**
- **ConsiderAllVariablesForBlockDiagramBasedSwitchOptimization**
- **ConsiderMergeableForBlockDiagramBasedSwitchOptimization**
- **AllowMovingOfCustomCodeForBlockDiagramBasedSwitchOptimization**

Differences in simulation behavior Moving code to conditionally executed control flow branches mainly reduces average and worst-case execution times of the generated code, because the respective calculations are not performed in all cases. However, the intermediate results are updated at different points in time than before the optimization. This can cause differences to the simulation

behavior of Simulink. TargetLink distinguishes between the following simulation behaviors:

- Simulation behavior equivalent to Simulink (MIL mode)
- Simulation behavior of optimized production code (SIL mode and PIL mode)

The following example shows the differences:

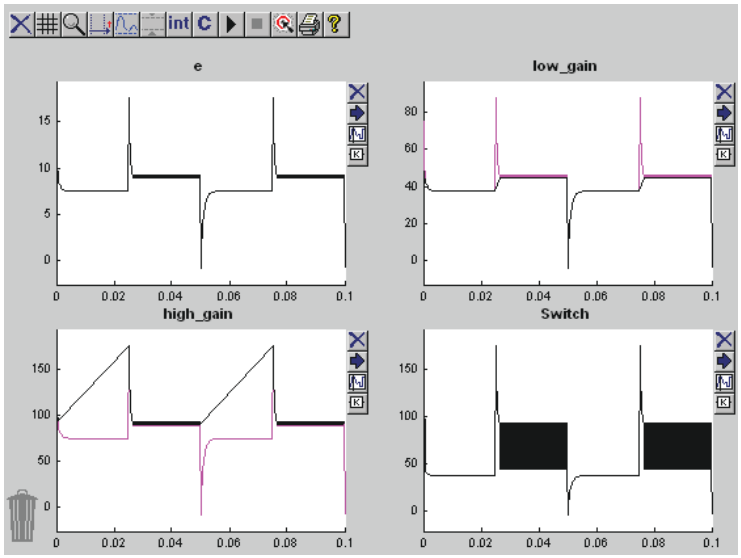


Both **Gain** block output variables are logged and the variable class is set to the **DISP** variable class.

The generated code below shows that both **Gain** block outputs are computed conditionally and logged in their associated branches.

```
DISP Int16 Sa1_high_gain;
DISP Int16 Sa1_low_gain;
...
void Switchcontroller(void)
{
...
   /* Sum: Switchcontroller/e */
   Sa1_e = (Int16) (((UInt16) Sa1_REF) - ((UInt16) Sa1_Y));
   LOG_VAR(a, _Sa1_e, Sa1_e);

   /* Switch: Switchcontroller/Switch */
   if (Sa1_e >= 0) {
      /* Gain: Switchcontroller/high_gain */
      Sa1_high_gain = Sa1_e * 10;
      LOG_VAR(a, _Sa1_high_gain, Sa1_high_gain);
      Sa1_Switch = Sa1_high_gain;
   }
   else {
      /* Gain: Switchcontroller/low_gain */
      Sa1_low_gain = Sa1_e * 5;
      LOG_VAR(a, _Sa1_low_gain, Sa1_low_gain);
      Sa1_Switch = Sa1_low_gain;
   }

   /* TargetLink outport: Switchcontroller/U */
   Sa1_U = (Int16) (Sa1_Switch >> 1);
}
```

The optimization of the conditionally logged signals results in different simulation behavior to Simulink, where both block outputs are computed for each sample. One plot line shows the simulation results equivalent to Simulink and the other plot line shows TargetLink's representation of the simulation results of optimized production code.

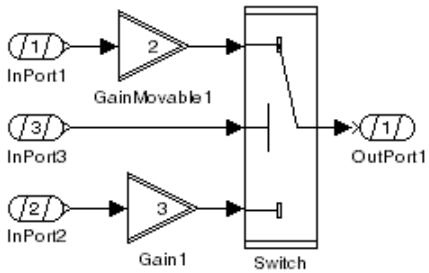


Note

Selecting the Simulink Conditional input branch execution checkbox on the Optimization dialog of the Configuration Parameters dialog box can lead to similar differences.

Example for MOVABLE

Assume you built a model like the following:



The generated code looks like this:

```
{
   /* SLLocal: Default storage class for local variables | Width: 64 */
   Float64 Sa1_Switch;
   /* Gain: gen_Subsystem/Gain1 */
   Sa1_Gain1 = Sa1_InPort2 * 3.;
```

```
   /* Switch: gen_Subsystem/Switch */
   if (Sa1_InPort3 >= 0.) {
      /* Gain: gen_Subsystem/GainMovable1 */
      Sa1_GainMovable1 = Sa1_InPort1 * Sa1_GainMovable1_gain;
      Sa1_Switch = Sa1_GainMovable1;
   }
   else {
      Sa1_Switch = Sa1_Gain1;
   }
   /* TargetLink outport: gen_Subsystem/OutPort1 */
   Sa1_OutPort1 = Sa1_Switch;
}
```

ERASABLE

Determines whether a variable can be removed.

Direct and indirect removal Variables can be removed either directly or indirectly. Direct removal means that a variable is removed from executed C code. Indirect removal means that the whole control flow branch where a variable is used is removed from C code. Control flow optimizations remove control flow branches that are never executed. *Variable removal* usually refers to direct removal.

Note

Indirect removal does not eliminate superfluous assignments to a variable or its declaration unless these statements are located in the removed control flow branch.

Kinds of removal The following kinds of variable removal exist:

- Replacing variables with their value or eliminating them completely

  This applies mostly to ranges and other value propagations. If the Code Generator option **ExploitRangesIndependentOfErasable** is enabled, unnecessary variables (no macros) with scope less or equal to module global are eliminated no matter whether the **Optimization** property is set to `ERASABLE`. By default, this option is turned on. However, if this option is turned off the following example applies:

  `logical = a && b && c && 0`

  can only be replaced by

  `logical = 0`

  if `a`, `b`, and `c` each belong to a variable class for which the **Optimization** property is set to `ERASABLE`. For example, if the variable class of `b` is not set to `ERASABLE`, only the statement `c && 0` can be replaced by `0`. For more information, refer to Code Generator Options.
- Eliminating temporary variables

  The elimination of temporary variables relates to unused pointer-to-struct parameters, unnecessary assignments, block outputs followed by signal line splits, etc. Refer to Basics on Eliminating Temporary Variables on page 518.
- Eliminating variable declarations/definitions that do not occur elsewhere in the code

Unnecessary variables are removed if the **Optimization** property of their variable classes is set to `ERASABLE`. The removal of unused parameters, called backward obsolete parameter removal, also belongs to this kind of elimination. For more information on backward obsolete parameter removal, refer to Basics on Optimizing the Function Interface on page 528.

- Replacing variables with more efficient variable kinds:
  - Variable sharing - Several variables are replaced by one auxiliary variable.
  - Dimension downgrade - A vector or matrix variable is replaced by one or more scalar variables.
  - Removal of struct components from a struct type - A struct component is replaced by a function-local variable for each struct instance in conjunction with the `SCOPE_REDUCIBLE` value of the **Optimization** property.

Restrictions for removal A variable can be eliminated only if the removal does not change the simulation behavior of the model at its interface. A variable cannot be removed from the production code if any of the following conditions is specified:

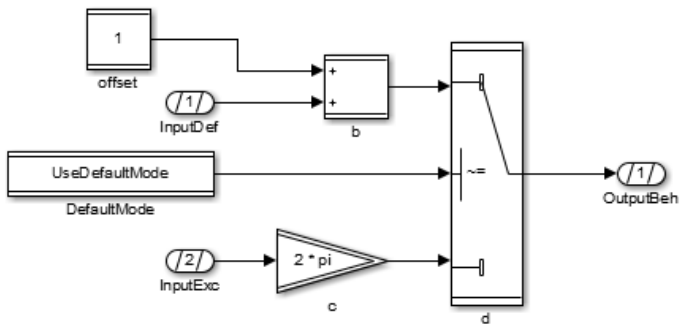
- The variable is logged.

  If a variable is logged, its signal values are saved for data analysis or plotting.
- The variable is set to a variable class other than **default** and the **Optimization** property of the variable class is not set to `ERASABLE` in the **TargetLink Data Dictionary**.

Example for ERASABLE

The following example shows different kinds of removal that are possible if the variable is `ERASABLE`, i.e., if the variable class **Optimization** property contains `ERASABLE`.

Assume you built a model like the following:



If optimization is disabled, e.g., via the **Optimization** Code Generator option, the generated code looks like this:

```
ConstSystemParameter Bool DefaultMode = 1;
…
   NonErasable Float32 c;

   Float32 b;
   Float32 d;

   b = InputDef + 1.F;
   c = InputExc * 6.2831853071795862F;

   if (DefaultMode != 0) {
      d = b;
   } else {
      d = c;
   }

   OutputBeh = d;
```

Suppose the block output variables `b` and `d` and the `DefaultMode` parameter are `ERASABLE`. The variables `b` and `d` can be eliminated as intermediate variables, the `DefaultMode` parameter via *initial value propagation*. These are examples for *direct removal*.

```
NonErasable Float32 c;

   c = InputExc * 6.2831853071795862F;

   if (1 != 0) {
      OutputBeh = InputDef + 1.F;
   } else {
      OutputBeh = c;
   }
```

The block output variable `c` is not `ERASABLE`, i.e., it cannot be directly removed. However, if the **Optimization** property is set to `MOVABLE`, then indirect removal is possible:

```
NonErasable Float32 c;

   if (1 != 0) {
      OutputBeh = InputDef + 1.F;
   } else {
      c = InputExc * 6.2831853071795862F;
      OutputBeh = c;
```

Utilizing `1 != 0` is always false, the else branch can be removed, effectively removing all accesses to `c`:

```
NonErasable Float32 c;

   OutputBeh = InputDef + 1.F;
```

The variable definition of `c` cannot be removed, because the **Optimization** property is not set to `ERASABLE`.

SCOPE_REDUCIBLE	Indicates that the scope of a variable can be reduced, for example, from global to local, to increase run–time performance and reduce RAM consumption. Scope reduction is also influenced by variable class properties like **ScopeReducedClass**, **Module**, **Storage**, and **Scope**. Refer to Basics on the Scope Reduction of Variables on page 513.
MERGEABLE	If possible, mergeable variables are merged during code generation. This means multiple definitions of a variable are identified by its name and mapped to one variable representing parameters of multiple blocks in the generated code. Mergeable variables can be referenced from multiple blocks in the model. Refer to Example of Merging Variables on page 123.
Optimizing user-specified variables	Even though TargetLink's default settings ensure efficient production code, it might sometimes be necessary to adapt TargetLink's defaults, for example, to meet your company's coding style or special target requirements. Therefore, TargetLink lets you create your own variables and adapt them to your needs. You can, for example, specify whether a variable you created is optimizable according to the terms described above. For more information, refer to Adapting Code to Company Coding Styles on page 77.
Logging variables	Logged variables cannot be directly removed from production code, but usually can be moved into conditionally executed control flow branches. For more information, refer to MOVABLE and ERASABLE in this section. Furthermore, logged variables increase both stack and RAM memory consumption, and also execution time. To get efficiently optimized code, it is therefore useful to log variables only if absolutely necessary.
Logging properties	The logging properties can be changed either block-by-block on the **Logging & Autoscaling** page of the corresponding block dialog, or via the **Code and logging options** group box on the **Code Generation** page of the TargetLink Main Dialog. If you want your code to be generated without the effect of logging, select the **Clean Code** checkbox in the **Code and logging options** group box on the **Code Generation** page of the TargetLink Main Dialog. For more information, refer to Basics on Logging (🕮 TargetLink Preparation and Simulation Guide).

Related topics

Basics

Basics on Logging (TargetLink Preparation and Simulation Guide)
Basics on Optimizing Functions via Property Values..... 524

HowTos

How to Optimize Variables..... 512

Examples

Example of Merging Variables..... 123

References

AllowMovingOfCustomCodeForBlockDiagramBasedSwitchOptimization (TargetLink Model Element Reference)
Code Generator Options (TargetLink Model Element Reference)
ConsiderAllVariablesForBlockDiagramBasedSwitchOptimization (TargetLink Model Element Reference)
ConsiderMergeableForBlockDiagramBasedSwitchOptimization (TargetLink Model Element Reference)
DoNotMoveIfLogged (TargetLink Model Element Reference)
EnableBlockDiagramBasedSwitchOptimization (TargetLink Model Element Reference)
ExploitRangesIndependentOfErasable (TargetLink Model Element Reference)

How to Optimize Variables

Objective

To improve your code, TargetLink allows you to specify property values for variable optimizations according to your needs.

Optimizing variables

Variables are optimized according to their variable class settings. Before you specify **Optimization** property values for a variable class, it is always advisable to check whether a variable class with your desired settings already exists.

Precondition

You must have opened a model and the Data Dictionary Manager.

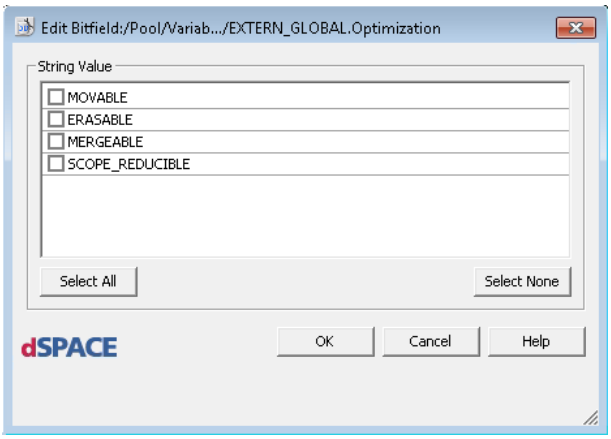
Method

To optimize variables

1 In the **Data Dictionary Manager**, change to the variable class you want to optimize, for example, change to **`Pool/VariableClasses/GLOBAL`**.

2 In the **Property Value List**, click the value edit field of the **Optimization** property.

The **Edit Bitfield** dialog opens.



3 Choose the desired value(s) from the list and click **OK**.

Result

You have now chosen the optimization value(s) for the selected variable class.

Tip

You can copy an existing variable class and change its properties according to your needs. This may be particularly useful if you are not working with the variable classes provided by the templates for DD project files.

Related topics

Basics

Basics on Optimizing Variables..... 504
Details on Preconfigured VariableClass Objects in the Data Dictionary..... 90

Basics on the Scope Reduction of Variables

Reducing the scope of variables

Reducing the scope of variables from global to module-local minimizes the dependencies between several C modules and reduces RAM consumption if a variable is reduced to local scope (stack variable). Thus, the reliability of the code and runtime performance are increased (less RAM, higher speed).

Influencing the scope reduction of variables

You can influence the scope reduction of variables by setting their associated DD VariableClass object's Optimization property to `SCOPE_REDUCIBLE`.

Additionally, you can use the following Code Generator options:

- ReduceScopeOfVariablesOnlyDownToFunctionLevel
- AvoidStaticLocalScope

Types of scope reduction

Variables with a global scope can be reduced as far down as local scope. There are two types of scope reduction for variables:

- Scope reduction of explicit variables
- Scope reduction of implicit variables and user-defined variables with the default variable class (implicit properties)

Scope reduction of explicit variables

To enable the scope reduction of explicit variables, set their associated DD VariableClass object's Optimization property to `SCOPE_REDUCIBLE`. Additionally, you can specify the scope that this variable is reduced to via a DD VariableClass object that you reference via the ScopeReducedClass property. Refer to How to Specify the Scope Reduction of Explicit Variables on page 515.

Note

- Explicit struct components can be reduced either to a local scope or to a user-specified value of the ScopeReducedClass property. For scope reduction of struct components the following preconditions must be fulfilled. These preconditions depend on the type of struct component you are working with:
  - DD struct components:
    The variable class must be a user variable class which must be set to `ERASABLE` and `SCOPE_REDUCIBLE`.
  - All the other struct components:
    The variable class must be either a default one or a user variable class. Each of these classes must be set to `ERASABLE` and `SCOPE_REDUCIBLE`.
- If you do not specify a scope-reduced class, TargetLink generates a variable class with a reduced storage and scope property.

Defining a scope-reduced class When you define a scope-reduced class for a variable, consider the following:

- The scope of the scope-reduced class must be smaller than the scope of the original class.
- Variables referencing a variable class with Alias set to on or Storage set to `extern` are not reduced in scope. TargetLink places external declarations in the header if they are intended for multiple use, and in the C file if they are intended for local module use.
- Scope-storage pairs other than the ones described are currently not supported.
- Variables whose Info property is set to a value other than none are reduced to static global scope only.

- If a global scope is reduced to a static global scope, the **RestartFunctionName** property must be identical for both variable classes (macros can be reduced only from global to module-local visibility. Unlike variables, the definition module cannot change).

Scope reduction of implicit and default variables

The Code Generator performs the scope reduction automatically for both implicit variables and variables with the default variable classes. Implicit variables are specified by the Code Generator using the variable class templates in the **TargetLink Data Dictionary**. Refer to Details on Scope Reduction and VCTs on page 217.

Related topics

Basics

Basics on Optimizing Variables..... 504
Controlling Objects and Properties Created by the Code Generator (Templates)..... 203
Details on Preconfigured VariableClass Objects in the Data Dictionary..... 90

HowTos

How to Specify the Scope Reduction of Explicit Variables..... 515

References

AvoidStaticLocalScope (📖 TargetLink Model Element Reference)
Hook Scripts (📖 TargetLink File Reference)
ReduceScopeOfVariablesOnlyDownToFunctionLevel (📖 TargetLink Model Element Reference)

How to Specify the Scope Reduction of Explicit Variables

Objective

To increase the reliability and the efficiency of the production code with regard to RAM consumption, TargetLink allows scope reduction for variables specified in your model.

Specifying scope reduction

The process of specifying scope reduction for explicit variables consists of the following steps:

1. Create a variable in the **TargetLink Data Dictionary**.
2. Assign the desired variable class to the variable.
3. Specify if the variable class is scope-reducible.

The instructions below show how to specify scope reduction from global scope to static global scope for the **position** variable.

Preconditions

The following preconditions must be fulfilled:

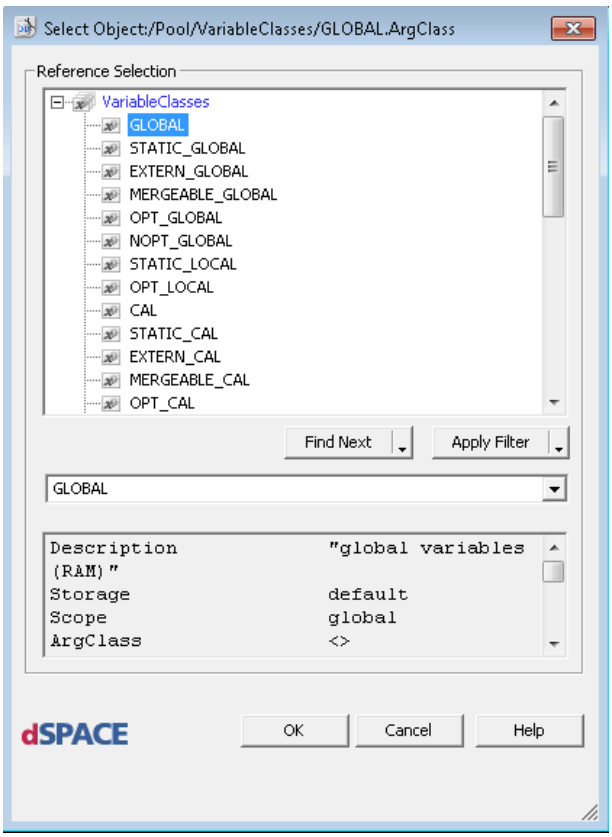
- You must have opened the Data Dictionary Manager.
- You must have opened a model.
- You must have created a position variable in the TargetLink Data Dictionary.

Method

To specify scope reduction of explicit variables

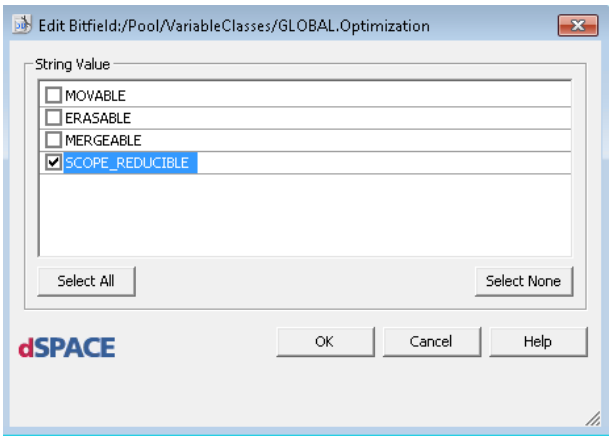
1. In the Data Dictionary Navigator of the Data Dictionary Manager, change to the `Pool/Variables/position` object.
2. In the Property Value List, click the Browse button of the Class edit field.

   The Select Object dialog opens.



3. From the VariableClasses list, select GLOBAL and click OK.
4. In the Data Dictionary Navigator of the Data Dictionary Manager, change to the `Pool/VariableClasses/GLOBAL` object.
5. In the Property Value List, click the Optimization edit field.

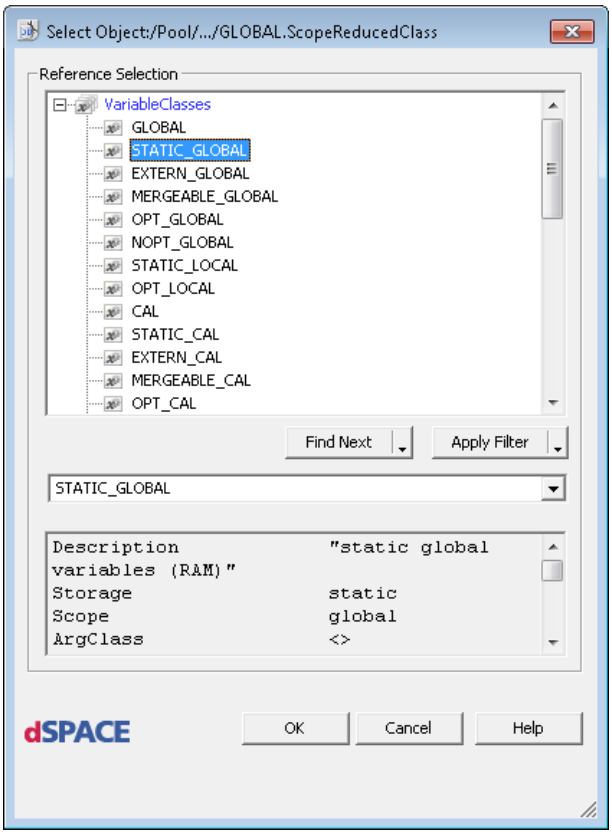
The Edit Bitfield dialog opens.



6 In the list, select SCOPE_REDUCIBLE and click OK.

7 In the Property Value List, click the Browse button of the ScopeReducedClass edit field.

The Select Object dialog opens.



8 From the VariableClasses list, select STATIC_GLOBAL and click OK.

Result

You have now specified scope reduction of the **position** variable from global scope to static global scope. The next step is to specify the **ScopeReducedClass** property for the STATIC_GLOBAL and STATIC_LOCAL variable class. This is done according to the steps described above.

Related topics

Basics

Basics on the Scope Reduction of Variables..... 513

Basics on Eliminating Temporary Variables

Eliminating direct assignments

Look at the following example:

```
b = a
c = b
```

Only if **b** belongs to a variable class with the **Optimization** property set to **ERASABLE**, the example can be optimized to:

```
c = a
```

In this example direct assignments as well as the uses of the variable value are eliminated.

Special cases of this removal are:

- The removal of unused pointer-to-struct parameters (forward obsolete parameter removal)
- The elimination of unnecessary assignments

Eliminating struct function arguments

If a function has a parameter of pointer-to-struct type that is either an input or an output parameter, for example, specified via TargetLink Bus InPort or OutPort blocks, and if the argument that is passed is the address operator applied to a struct variable, then TargetLink can eliminate this struct variable if a complete copy from/to another struct takes place before/after the function call.

Note

Actually, this is an elimination of all the struct components that have to be assigned beforehand. That is, TargetLink considers the non-struct-type struct components of the struct variable and all its (recursive) substruct components, the so-called *leaf components*. TargetLink can only perform this optimization if the leaf components are eligible for the elimination of temporary variables. The included struct components must therefore have a default variable class or a user variable class with the **ERASABLE** **Optimization** property set to Enabled.

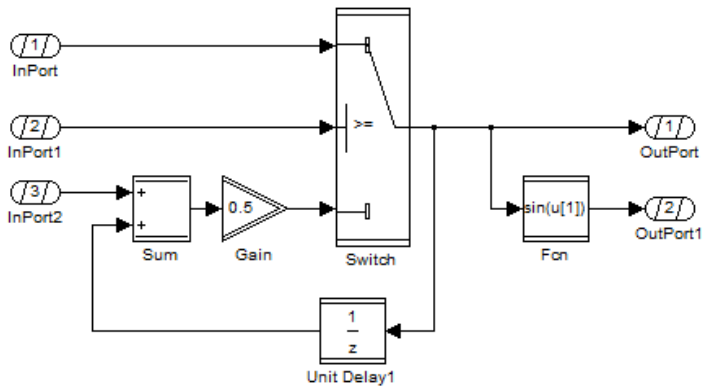
For a code example, refer to Basics on Optimizing Struct Variables on page 521.

Code generator option This optimization feature is related to the UtilizeValueEqualityStructFunctionArguments Code Generator option.

Eliminating block outputs followed by signal line splits

If the UtilizeValueEqualitySignalLineSplit Code Generator option is enabled, then TargetLink is allowed to eliminate variable accesses that match a pattern involving signal splits with a TargetLink Outport or Unit Delay block as one of the successor blocks.

Example Suppose code is generated from the following model part:



The generated code differs with respect to the UtilizeValueEqualitySignalLineSplit optimization option as follows:

Optimization option enabled (default) The Switch output is replaced by the Unit Delay1 state:

```
void SignalLineSplit(void)
{
static Float64 X_Sa1_Unit_Delay1 = 0.;

if (Sa1_InPort1 >= 0.) {
   X_Sa1_Unit_Delay1 = Sa1_InPort;
}
else {
   X_Sa1_Unit_Delay1 = (Sa1_InPort2 + X_Sa1_Unit_Delay1) * 0.5;
}
Sa1_OutPort1 = (Float64) sin(X_Sa1_Unit_Delay1);
Sa1_OutPort = X_Sa1_Unit_Delay1;
}
```

Optimization option disabled The code is less efficient because the Switch output is kept:

```
void SignalLineSplit(void)
{
Float64 Sa1_Switch;
static Float64 X_Sa1_Unit_Delay1 = 0.;

if (Sa1_InPort1 >= 0.) {
   Sa1_Switch = Sa1_InPort;
}
else {
   Sa1_Switch = (Sa1_InPort2 + X_Sa1_Unit_Delay1) * 0.5;
}
Sa1_OutPort1 = (Float64) sin(Sa1_Switch);
Sa1_OutPort = Sa1_Switch;
X_Sa1_Unit_Delay1 = Sa1_Switch;
}
```

Dimension downgrade

TargetLink can replace accesses to the same index range of a vector or matrix variable by a scalar variable if all of the following conditions are met:

- All the accesses to the variable are governed by the same loop variable or expression.
- All the accesses to the variable occur within the same function.
- The variable is ERASABLE.

Note

TargetLink performs this optimization only if one of the following conditions is met:

- The total amount of memory used by the scalar replacement variables is less than the amount used by the original vector or matrix variable.
- You set the ReplaceUnrolledVectorsAndMatricesByScalars Code Generator option to on.
  TargetLink will then perform this optimization even if the amount of memory used by the scalar replacement variables is equal to the amount used by the original vector or matrix variable.

Example The following example shows this optimization for two vector variables:

Unoptimized	Dimension Downgrade

`for(i = 0; i < 10; i++){`
`  vector1[i] = a[i] + b[i];`
`  f(&vector1[i]);`
`  c[i] = vector1[i] + 3;`
`}`
`vector2[e-d] = c[e-d]*7;`
`g(&vector2[e-d]);`
	`for(i = 0; i < 10; i++){`
	`  Aux_S16 = a[i] + b[i];`
	`  f(&Aux_S16);`
	`  c[i] = Aux_S16 + 3;`
	`}`
	`Aux_S16_a = c[e-d]*7;`
	`g(&Aux_S16_a);`

Related topics

Basics

Basics on Optimizing Functions via Property Values..... 524

HowTos

How to Optimize Variables..... 512

References

ReplaceUnrolledVectorsAndMatricesByScalars (📖 TargetLink Model Element Reference)
UtilizeValueEqualitySignalLineSplit (📖 TargetLink Model Element Reference)
UtilizeValueEqualityStructFunctionArguments (📖 TargetLink Model Element Reference)

Basics on Optimizing Struct Variables

Assignment of structs

TargetLink can simplify the assignment of struct and pointer-to-struct variables of the same (underlying) struct type. One overall assignment of a whole struct replaces multiple assignments of struct components:

```
struct T S1;
struct T S2;
...
S2 = S1;
```

instead of

```
struct T S1;
struct T S2;
...
S2.m1 = S1.m1;
S2.m2 = S1.m2;
S2.m3.m1 = S1.m3.m1;
S2.m3.m2 = S1.m3.m2;
S2.m4 = S1.m4;
...
S2.mX = S1.mX;
```

Code Generator option This optimization feature is related to the `AllowStructAssignments` Code Generator option.

Elimination of struct function arguments

TargetLink can eliminate struct variables for certain parameters of pointer-to-struct type. For basic information, refer to Basics on Eliminating Temporary Variables on page 518.

Example Suppose, the following code is generated because no struct optimization is performed at all:

```
struct tag_T S1;
struct tag_T S2;
...
S2.a = S1.a;
S2.sub.b = S1.sub.b;
S2.sub.c = S1.sub.c;
S2.d = S1.d;
...
/* Pointer-to-struct parameter only used for input */
r = f(&S2);
```

If struct assignments are simplified via the `AllowStructAssignments` Code Generator option, the `S1` struct is assigned to the `S2` struct as a whole:

```
struct tag_T S1;
struct tag_T S2;
...
S2 = S1;

/* Pointer-to-struct parameter only used for input */
r = f(&S2);
```

If struct function arguments are eliminated via the `UtilizeValueEqualityStructFunctionArguments` Code Generator option, the `S2` struct is replaced by the `S1` struct:

```
struct tag_T S1;
...
/* Pointer-to-struct parameter only used for input */
r = f(&S1);
```

Conversely, the TargetLink optimization replaces `S3` if all leaf components are assigned a new value in a function, for example:

```
struct tag_T S4;
...
/* Pointer-to-struct parameter only used for output */
g(in1, in2, &S4);
```

instead of this less efficient code pattern:

```
struct tag_T S3;
struct tag_T S4;
...
/* Pointer-to-struct parameter only used for output */
g(in1, in2, &S3);
...
S4 = S3;
```

Related topics

References

AllowStructAssignments (📖 TargetLink Model Element Reference)
UtilizeValueEqualityStructFunctionArguments (📖 TargetLink Model Element Reference)

Optimizing Functions

Where to go from here

Information in this section

Basics on Optimizing Functions via Property Values..... 524
Basics on Optimizing the Function Interface..... 528
How to Optimize Functions..... 531
Basics on the Scope Reduction of Functions..... 531
Basics on Merging External Functions..... 532

Basics on Optimizing Functions via Property Values

Optimization property

TargetLink provides an efficient method for optimizing functions. The function classes stored in the DD **`Pool/FunctionClasses`** object in the TargetLink Data Dictionary contain an Optimization property, which lets you specify property values (see below) for determining the optimization of functions.

Supported property values for function optimization

Supported property values for the Optimization property of function classes are the following:

- `SIDE_EFFECT_FREE`
- `MOVABLE`
- `CG_AUTO_INLINE`
- `CG_FORCE_INLINE`
- `CG_NO_INLINE`

Note

- For the Optimization property values SIDE_EFFECT_FREE and MOVABLE to become effective, you must have activated optimization. For information on optimization, refer to Basics on Optimizing the Production Code on page 498.
- The execution of the Optimization property values CG_AUTO_INLINE, CG_FORCE_INLINE and CG_NO_INLINE does not depend on the setting of the optimization option, but on the InliningThreshold Code Generator option .

SIDE_EFFECT_FREE

`SIDE_EFFECT_FREE` primarily indicates that the Code Generator might assume that all data flow relationships of a function or an atomic subsystem are made explicit via signal lines. Depending on whether you consider a block diagram or generated code, `SIDE_EFFECT_FREE` has the following meaning:

- Block diagram: In an atomic subsystem or a system with a **Function** block, there are no blocks containing internal states and unclear data flow, which can occur, for example, if variables are specified as `MERGEABLE` or if **Data Store Read** and **Data Store Write** blocks are used. Examples are subsystems with **Function** blocks for which incremental code generation is enabled or calls to functions defined in user look-up table scripts.
- Generated code: Apart from the specified function interface, the function and the functions calling it do not communicate via global variables, and there are no calls to non-`SIDE_EFFECT_FREE` functions that are defined outside TargetLink, i.e., from the viewpoint of the caller of the function. Variables are accessed only from the function's signature and modeled ports.

Code Generator analysis By default, the Code Generator analyzes models regarding the use of global variables and internal states. The analysis takes place for all subsystems that are atomic or contain **Function** blocks.

Note

Internal states are variables that are modified and evaluated only in the function and all functions called directly or indirectly by it.
A variable is considered a state if the evaluation of some or all struct components and/or elements is not always preceded by a value modification during the current function execution.
If the caller of the function or one of its direct or indirect callers modifies or evaluates the value of a state accessed in the function, then the state is not considered function-internal.
Internal states do not necessarily affect the output values of the function, e.g., a starting point of an analysis whose outcome depends only on inputs and parameters.
If you specify a function class as `SIDE_EFFECT_FREE` and/or MOVABLE, this is communicated to TargetLink.

If you define a subsystem as side-effect-free, no analysis is performed and the particular subsystem is treated as side-effect-free without further checks. This does not mean, however, that optimizations for function classes are prevented, if side-effect-free is not set.

Possible optimizations If a function is specified as side-effect-free, the following additional optimizations can be performed:

- Global variables can be moved over function calls and vice versa.
- Variables with aliases can be moved over function calls.
- Function calls can be moved over global variables.

- Function calls can be moved to dependent control flow branches.

> **Tip**
>
> If a function has only internal states or accesses only global variables that are not accessed in the calling context, then it may be useful to make it `SIDE_EFFECT_FREE` but not MOVABLE. This lets you move global variables or calls to other functions past the function call (or move the function call), but you cannot move the function call into a conditional control flow branch or out of a loop.

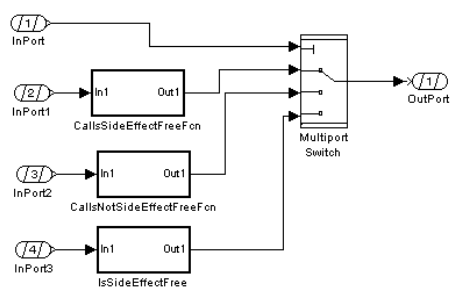
Functions that are not side-effect-free For functions that are not specified as side-effect-free, the Code Generator does not make the following assumptions without further analysis:

- Global variables are not accessed within the function except for those entering the function via the function call or via inputs or outputs of subsystems.
- The function does not contain state variables that are accessed or changed.

External functions If an external function is specified as side-effect-free, the Code Generator assumes that data is written to the outputs for every single call of the function. Thus, a call to function for `ref_param` parameters is interpreted as if the parameters were subject to an assignment. This makes it possible to eliminate argument variables if they are copied to another variable and reduce their scope.

Example for SIDE_EFFECT_FREE

Assume you built a model like the following:



The generated code looks like this:

```
{
/* SLLocal: Default storage class for local variables | Width: 16 */
 Int16 Sa1_Multiport_Switch;
/* call of function: Functions_1/CallsNotSideEffectFreeFcn */
Sa2_CallsNotSideEffectFreeFcn();
```

```
/* Multiport switch: Functions_1/Multiport Switch */
switch(Sa1_InPort){
case1:{
/* call of function: Functions_1/CallsSideEffectFreeFcn */
Sa3_CallsSideEffectFreeFcn();
 Sa1_Multiport_Switch = IF_Sa6_Out1;
break;
}
case2:{
 Sa1_Multiport_Switch = Sa5_Unit_Delay;
break;
}
default:{
/* Default case created by TargetLink for the block 'Multiport
 switch: Functions_1/ Multiport Switch'. The default case is
 identical to the case for the last input of the block call
 of function: Functions_1/IsSideEffectFree */
Sa4_IsSideEffectFree();
 Sa1_Multiport_Switch = IF_Sa4_Out1;
break;
}
}
/* TargetLink outport: Functions_1/OutPort */
 Sa1_OutPort = Sa1_Multiport_Switch;
}
```

MOVABLE

Determines that function calls belonging to user function classes can be moved into conditional control flows or out of iterated control flows within the model or the code. Specify a function as MOVABLE only if you are sure that it does not modify states affecting the system behavior (refer to Code Generator analysis on page 525) and if you do not wish to observe the function's operation for every execution, for example, due to desired variable value or execution time measurement.

Function inlining

For more information on CG_AUTO_INLINE, CG_FORCE_INLINE, and CG_NO_INLINE, refer to Basics on Function Inlining on page 350.

Related topics

Basics

Basics on Optimizing Variables..... 504

HowTos

How to Optimize Functions..... 531

References

InliningThreshold (📖 TargetLink Model Element Reference)

Basics on Optimizing the Function Interface

Removing superfluous parameters

The Code Generator analyses the function interface as specified during modeling. It is capable of detecting unnecessary (unused, unchanged) parameters and removes them, reducing stack consumption and code execution time.

There are two types of parameter removal:

- Backward obsolete parameter removal

  If a parameter is not used in a function, it is unnecessary and is therefore removed. This kind of removal iteratively deletes unused parameters.
- Forward obsolete parameter removal

  Suppose a function always obtains the same value for a certain parameter, for example the address of a specific structure. This value occurs in all the calling functions, but not outside of the function call. Thus, the parameter can be replaced by the value in all the calling functions and need not to be calculated anymore. At the moment it only takes place for pointer-to-struct parameters.

Note

- The function interface is not modified in cases where it is an interface to external code. For example, the function interface of the TargetLink root system, externally function-call-triggered subsystems, subsystems configured for incremental code generation, external subsystems, and reused functions are not modified. If a pointer-to-struct parameter is listed as additional parameter in a **Function** block, it will also not be modified.
- A parameter is considered optimizable if it is specified as **erasable** or **default** via the associated **Optimization** property of its variable class in the DD.

Propagation of Constant blocks

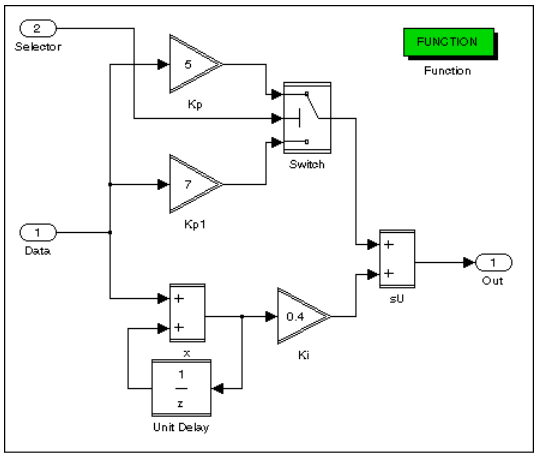
You can place a **Constant** block without scaling information directly in front of a TargetLink subsystem that does not contain an TargetLink **InPort** block. This is possible because TargetLink can propagate **Constant** blocks into subsystems, which reduces stack consumption (fewer function parameters) and simplifies modeling. During code generation the optimization is applied iteratively, moving the constant even in nested subsystems.

TargetLink applies similar optimization algorithms if:

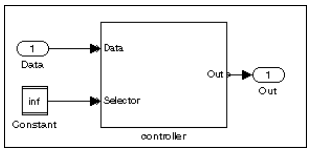
- The constant is a vector/matrix; in this case, the algorithms can be applied to one or more vector/matrix elements.
- You connect several **Constant** blocks via connection blocks such as Simulink Mux, Demux, Selector and Bus Selector blocks.

Example

Assume you built a library containing a configurable controller as shown below.



In your implementation, the proportional factor Kp is controlled via the Selector inport.



The unscaled constant of the inport Selector is treated as if it is part of the subsystem. It is therefore not used as a parameter when the function interface is generated. As a result, only the function parameters for the Data and Out ports and, if required, the implementation of the proportional factor Kp, are generated.

```
Int16 S01_in;
static Int16 IF_S03_sU /* LSB: 2^-6 OFF:  0 MIN/MAX:  -512 .. 511.984375 */;
static Int16 S03_Data;
/*************************************************************************\
   STATIC_CAL: static calibratable parameters (ROM)
\*************************************************************************/
STATIC_CAL Int8 P_S03_Ki = 26 /*0.4*/ /*
   Description: integral gain factor
   LSB: 2^-6 OFF:  0 MIN/MAX:  -2 .. 1.984375 */;
STATIC_CAL Int8 P_S03_Kp = 40 /*5.*/ /*
   Description: proportional gain factor
   LSB: 2^-3 OFF:  0 MIN/MAX:  -16 .. 15.875 */;
STATIC_CAL Int8 P_S03_Kp1 = 56 /*7.*/ /*
   Description: proportional gain factor
   LSB: 2^-3 OFF:  0 MIN/MAX:  -16 .. 15.875 */;
/*************************************************************************\
   STATIC_DISP: static observable variables (RAM)
\*************************************************************************/
STATIC_DISP Int16 S03_sU /*
   Description: this is the controller output signal
   LSB: 2^-6 OFF:  0 MIN/MAX:  -512 .. 511.984375 */;
...
void S03_controller(void)
{
   /* SLLocal: Default storage class for local variables */
   Int16 S03_x /* LSB: 2^-6 OFF:  0 MIN/MAX:  -512 .. 511.984375 */;
   /* SLStaticLocalInit: Default storage class for static local variables with
      initvalue */
   static UInt16 X_S03_Unit_Delay = 0; /* LSB: 2^-6 OFF:  0
                                          MIN/MAX:  0 .. 1023.984375 */
   /* Sum: Subsystem/Subsystem/controller/x */
   S03_x = (Int16) (((UInt16) (Int16) (S03_Data << 6)) + X_S03_Unit_Delay);
   LOG_VAR(0, _S03_x, S03_x);
   /* Sum: Subsystem/Subsystem/controller/sU
      # combined # Gain: Subsystem/Subsystem/controller/Kp
      # combined # Gain: Subsystem/Subsystem/controller/Ki */
   S03_sU = (Int16) (((UInt16) (Int16) (((Int16) (((Int16)
      ((S03_Data * ((Int16) P_S03_Kp)) << 3)) >> 6)) << 6)) + ((UInt16) (Int16)
      ((((Int32) S03_x) * ((Int32) P_S03_Ki)) >> 6)));
   /* update of global auxiliary variable(s) associated with
      Subsystem/Subsystem/controller/sU */
   IF_S03_sU = S03_sU;
   LOG_VAR(0, _S03_sU, S03_sU);
   /*  --- update section ---  */
   /* Unit delay: Subsystem/Subsystem/controller/Unit Delay */
   X_S03_Unit_Delay = (UInt16) S03_x;
}
...
void Subsystem(void)
{
   S03_Data = S01_in;
   /* call of function: Subsystem/Subsystem/controller */
   S03_controller();
   /* Outport: Subsystem/out. */
   S01_out = (Int16) (IF_S03_sU >> 6);
}
```

How to Optimize Functions

Objective

To improve your code, TargetLink allows you to specify values for function optimizations via the **Optimization** property in the **Property Value List** of the Data Dictionary Manager.

Precondition

You must have opened a model and the Data Dictionary Manager.

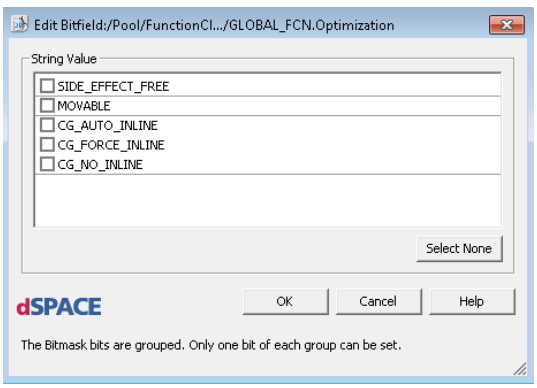
Method

To optimize functions

1 In the **Data Dictionary Manager**, change to the function class you want to optimize, for example, change to `Pool/FunctionClasses/GLOBAL_FCN`.

2 In the **Property Value List**, click the value edit field of the **Optimization** property.

The **Edit Bitfield** dialog opens.



3 Choose the desired value(s) from the list and click **OK**.

Result

You have now chosen the optimization value(s) for the selected function class.

Related topics

Basics

Basics on Optimizing Variables..... 504

Basics on the Scope Reduction of Functions

Reducing implicit classes

TargetLink supports the scope reduction of functions to optimize the production code. Reducing the scope of functions from global to module-local minimizes

the dependencies between several C modules. Only functions generated via implicit function classes are reduced in scope. Scope reduction of implicit functions works in the same way as scope reduction of implicit variables. Functions and macros with global scope can be reduced to static global scope according to where they are used in the generated code. Implicit classes are reduced according to the following mapping rule:

`Global` → `static global`

Note

There is no subdivision into several classes as there is with variables.

Related topics

Basics

Basics on the Scope Reduction of Variables..... 513

Basics on Merging External Functions

Basics

TargetLink lets you merge external functions with identically modeled interfaces, for example, functions resulting from subsystems which are duplicated in the model by copy & paste. To have TargetLink merge external functions, you have to enable the Code Generator option **EnableExternFunctionMerge**.

Note

TargetLink does not verify if the subsystems whose functions are to be merged are identical. It only verifies if the following points match:

- The settings in the **Function** blocks and
- The settings of the blocks specifying the interface variables

You must ensure that the subsystems representing the external functions to be merged are identical. However, check that you do not erroneously enable implicit function reuse.

Limitations

If you merge external functions, you must note the following limitations:

- Stateflow charts cannot be merged.
- Merging external functions is permitted only for `Step`, `Init`, `Term`, and `Restart` functions of subsystems that contain a **Function** block, and **Charts**.
- External functions cannot be merged if their interfaces contain bus ports that specify structs.

- External functions that have pointers-to-structs as formal parameters cannot be merged with other subsystems.

Related topics

Basics

Basics on Function Reuse 298
Basics on Optimizing the Production Code 498

References

EnableExternFunctionMerge (📖 TargetLink Model Element Reference)

Optimizing Block-Specific Code

Introduction

TargetLink offers special algorithms to optimize the code for certain TargetLink blocks.

Basics on Optimizing Block Output Saturation

Saturating block outputs

Block outputs can be saturated to prevent overflows. If a saturated block output exceeds its implemented range, the saturation sat() limits this output to the specified max and min values.

Cascaded saturation

TargetLink's interblock optimization can find combinations of blocks with saturated outputs and Saturation blocks to eliminate superfluous saturation operations. Cascaded saturation operations are combined if:

- The saturation limits of a block output (block 1) are more restrictive than the upper/lower limits of a downstream Saturation block (block 2), which means that upperlimit1 <= upperlimit2 and lowerlimit1 >= lowerlimit2. In this case, the saturation operation can be simplified as: sat1(sat2(a)) == sat1(a) i.e. the saturation of block 2 is omitted.
- The upper/lower limits of a Saturation block (block 2) are more restrictive than the saturation limits of the preceding block output (block 1), which means that upperlimit1 >= upperlimit2 and lowerlimit1 <= lowerlimit2. In this case, the saturation can be simplified as: sat1(sat2(a)) == sat2(a) i.e. the saturation of block 2 is omitted.

In addition, the variable class of the upper and lower limits of the Saturation block must contain the **default** variable class.

The following table lists all the valid combinations for saturation operations that can be combined during block level optimization:

Block	Followed by Block
Gain	Saturation
Sum	Saturation
Product	Saturation
Saturation	Saturation
Discrete Transfer Function	Saturation
Discrete Filter	Saturation
Discrete State-Space	Saturation

Variable classes of saturation limits

To stop a limit value from being removed from the production code, set it to a variable class other than **default**. As a result, TargetLink does not include it in

the optimization process. If only one limit of a saturation operation has the **default** variable class (for example, VarClass(lower limit) = **default** and VarClass(upper limit) = **global**), TargetLink can partially optimize the saturation operation.

Related topics

Basics

Implementation of Block Output Saturation (📖 TargetLink Preparation and Simulation Guide)

Optimizing Target-Specific Code

Where to go from here

Information in this section

Basics on Target-Specific Optimization..... 536

Details on Code Optimization on Block and Block-Independent Level..... 537

Details on Look-Up Table Settings for Target-Optimized Code Generation..... 540

Basics on Target-Specific Optimization

Optimization modules

Although the generic ANSI C code produced by the TargetLink Base Suite is already highly efficient, this efficiency can be increased even more, if TargetLink's Code Generator takes special processor features into account. This depends on the application and on the target microcontroller and compiler you use. TargetLink provides optimization modules for several of these microcontroller/compiler combinations.

Tested optimization modules

These special optimization modules give you not only higher code efficiency, but also the assurance that extensive tests were performed with the processor/compiler combination. The code of thousands of test patterns on block and model level was compiled with the specific compiler and executed on the specific microcontroller, and the results were compared with the results from Simulink simulations. Compiler bugs were detected during these tests. TargetLink generates work-arounds for these bugs.

Using target-specific optimizations

The target-specific optimizations can be used on two different levels:

- ANSI C compliant code
- Non-ANSI C code that includes assembly code or compiler extensions (special functions for hardware features that cannot be used with ANSI C)

ANSI C compliant code

Although ANSI C code can be compiled for each microcontroller (for example, the code optimized for the Infineon C167 can also be compiled on a Freescale CPU32), it is optimized for a specific one.

There are often several ways to implement a particular functionality in C. For example, you can use either the for or the while C construct to implement a loop. The for loop might be the best implementation for one microprocessor,

and the while loop for another. Thus, your code is target-specific and complies with ANSI C. The code patterns TargetLink uses to implement different Simulink blocks were chosen with respect to these target-specific features.

Non-ANSI C code

Sometimes the C language does not allow you to use special hardware features. You can enhance code efficiency by directly specifying the machine instructions in assembly language.

Some algorithms are inefficient and difficult to implement in C because of language limitations, not because of inefficient compilers. TargetLink can therefore optionally produce small portions of the code in assembly language. Moreover, compilers often provide non-ANSI C extensions for hardware features that cannot be used with ANSI C. TargetLink uses these extensions if available. Compilers can usually integrate these extensions better than assembly code.

The drawbacks of assembly code are not important when code is generated automatically. Maintenance and portability are performed on a Simulink model level rather than on a code level.

Benefits of target-specific code generation

The benefits of target-specific code generation depend on your application and the microcontroller you use. For example, if you often use saturation (overflow protection) because you are not sure which range your signals can take, you should let TargetLink check the controller's overflow flag in assembly language. On the other hand, if you have a 32-bit CISC machine like the Freescale CPU32 family, which has a very extensive instruction set, there might be only little efficiency gain with target-specific code for many applications. However, you can sometimes save execution time and code size with optimized code for the processor.

Details on Code Optimization on Block and Block-Independent Level

Introduction

You can apply code optimizations to different levels.

Levels of target-specific optimization

Target-specific optimizations are introduced on:

- A block level
- A block-independent instruction or operation level

Block level

This optimization applies mainly to high-level blocks like the Look-Up Table block, the Discrete Transfer Fcn block, etc. During production code generation, TargetLink generates a code pattern that lets the compiler implement the most efficient machine code. This can result in additional intermediate variables, different data types, or different conditional expressions, for example, if this helps the compiler to optimize.

Sometimes special block parameters are necessary to optimize the code best: for example, to let TargetLink use the look-up table machine instruction of the CPU32 microcontrollers or to use the C167's optimized table search loop. For detailed information on these block parameters, see Details on Look-Up Table Settings for Target-Optimized Code Generation on page 540.

Block-independent instruction level

TargetLink can also perform target-specific code optimizations on a block-independent instruction or operation level. TargetLink's internal data structure represents the whole model by hierarchy graphs containing operations and operands. For example, a fixed-point Sum block is represented by adds, shifts, and multiplications and divisions.

These basic operations can often be implemented according to a specific target. For example, the result of a multiplication usually has a double width. Most microcontrollers provide machine instructions to handle these situations. The C language does not contain expressions with different size operands. Divisions cause similar problems. Depending on the compiler, different methods are used to make the compiler use the optimal instructions. Sometimes only non-ANSI C extensions lead to the desired solution.

Example

As shown below, there are different ANSI C and assembler solutions for the same functionality of a saturated addition.

```
 y  = x1 + x2;
x1: Int16, LSBx1 = 1, ox1 = 0
x2: Int16, LSBx2 = 1, ox2 = 0
 y: Int16, LSBy  = 1,  oy = 0
```

Solution 1 (ANSI C)

The first way to prevent the addition from overflowing is to increase the word width for the operation and to check if the result fits into the desired output data type afterwards.

```
Int16 x1, x2, y;
Int32 dummy;
dummy = (Int32)x1 + (Int32)x2;
y = (Int16)dummy;
if(dummy > 32767)
   y = 32767;
else if(dummy < -32768)
   y = -32768;
```

On most 32-bit microcontrollers, where 32-bit operations are no problem, the code would be implemented as shown above.

Solution 2 (ANSI C)

If you use 16-bit controllers, the following solution, with a kind of plausibility check, is more efficient.

```
Int16 x1, x2, y;
y = x1 + x2;
if ((x1 >= 0) & (x2 >= 0) & (y < 0)) {
      y = 32767;
   } else {
      if (  (x1 < 0) & (x2 < 0) & (y >= 0)  ) {
         y = -32768;
      }
   }
```

The efficiency varies significantly between these two implementations on the different microcontrollers.

Solution 3 (ASSEMBLY)

An additional increase in efficiency can be achieved with assembly code. The overflow flag is checked. If it was set by the hardware, the output is saturated to the limits after the 16-bit addition.

This is only one real C example for target-specific code on an operation level. Here are a few other target-dependent implementations:

- Some microcontrollers are bad at shifting by particular numbers of bits. TargetLink tries to avoid these critical constants.
- Some microcontrollers provide combined multiply-accumulate operations. TargetLink uses these operations not only in classic DSP applications like filters (Discrete Transfer Fcn), but also each time the implementation of blocks results in appropriate add and multiply operations.

Sometimes some arithmetic is available only with target-specific non-ANSI C code (at least in practical applications). If you need high accuracy for a part of your model, for example, for a high-order Discrete State-Space block or Discrete Transfer Fcn block, you might want to use 32-bit variables. This is no problem for 32-bit processors, but for intermediate results (for example, after a gain multiplication), you need variables over 32-bit wide. Processor architectures usually take this into account and provide double registers for these intermediate results.

There are no corresponding data types in C. TargetLink provides pure ANSI C macros for this 64-bit arithmetic but for efficiency reasons, they can only be used in exceptions. If TargetLink is allowed to generate non-ANSI C code, special assembly macros are used. They dramatically increase efficiency, and make this high-precision arithmetic available for practical applications.

Related topics

Basics

Details on Look-Up Table Settings for Target-Optimized Code Generation..... 540

Details on Look-Up Table Settings for Target-Optimized Code Generation

Selecting a target

The TargetLink Code Generator can generate code optimized for a specific microcontroller, provided that the associated target optimization backend is available. On the Code Generation page of TargetLink's Main Dialog, you can select the target that the code should be optimized for.

Specifying table settings

To enable the Code Generator to exploit target-specific properties, you must specify the table settings you wish to have target-optimized code generated for. Table settings can be tailored for the targets: Infineon C16x, Freescale HC12/HCS12 and Freescale MPC55x and MPC55xx.

Infineon C16x

For the linear search, start low search algorithm, the C167 processor provides the assembler comparison instruction CMP (refer to *Table Searching* in the *C167 Derivatives User's Manual* available from Infineon).

Conditions for 1-D Look-Up Table blocks If code is optimized for C16x targets, the instruction is generated into the table function if the following conditions are fulfilled:

- The Assembler and C extensions checkbox is selected on the Code Generation page of the TargetLink Main Dialog.
- The linear search, start low search algorithm is selected.
- The data type of the associated table vector is an 8- or 16-bit integer type.
- The number of the table vector's values is greater than 10 for 8-bit and greater than 15 for 16-bit data types, because inserting assembler code is not recommended for small tables.

Freescale HC12/HCS12

For interpolation, HC12/HCS12 processors support the TBL assembler instruction. For more information, refer to the *HC12/HCS12 Reference Manual* provided by Freescale.

Conditions for 1-D Look-Up Table blocks If code is optimized for HC12/HCS12 targets, the assembler instruction is generated into the table function if the following conditions are fulfilled:

- On the Code Generation page of the TargetLink Main Dialog, the Assembler and C extensions checkbox is selected.
- The x-axis vector is equidistantly implemented with the Int8 or UInt8 data type.
- The y-axis vector is implemented with the Int8 or UInt8 data type.
- The Interpolation – Use End Values look-up method is selected.

Conditions for Interpolation (n-D) using PreLook-Up blocks If code is optimized for HC12/HCS12 targets, the assembler instruction is generated into the table function if the following conditions are fulfilled:

- On the Code Generation page of the TargetLink Main Dialog, the Assembler and C extensions checkbox is selected.

- The Number of table dimensions is 1.
- The Apply interpolation checkbox is selected.
- The block input signal has the UInt8 data type.

Freescale MPC5xx/MPC55xx

For the PreLook-Up Index Search block and the Interpolation (n-D) using PreLook-Up block, the TargetLink run-time library contains functions that are fully implemented in assembler.

Conditions for PreLook-Up Index Search blocks If code is optimized for MPC5xx/MPC55xx targets, the above functions are called if the following conditions are fulfilled:

- The Assembler and C extensions checkbox is selected on the Code Generation page of the TargetLink Main Dialog.
- One of the following search algorithms is selected:
  - Linear search, start low
  - Linear search, start high
  - Local search
- The Output configuration is set to Separate output variables for index and fractions.
- The output is a UInt8 data type.

Conditions for Interpolation (n-D) using PreLook-Up blocks If code is optimized for MPC5xx/MPC55xx targets, the above functions are called if the following conditions are fulfilled:

- The Assembler and C extensions checkbox is selected on the Code Generation page of the TargetLink Main Dialog.
- The Apply interpolation checkbox is selected.
- The block input signal has the UInt8 data type.

Fixed-Point Look-Up Tables on Floating-Point CPUs

TargetLink offers a special version of look-up tables (1-D and 2-D tables only) to be used with floating-point CPUs. On a floating-point CPU, all variables should be set to a floating-point data type such as Float32 or Float64. Thus, the floating-point instructions of the CPU are used and no scaling code is necessary. A disadvantage of table vectors and matrices implemented with floating-point data types is the additional amount of memory needed. Depending on the selected data type (Float32 or Float64), each table entry uses 4 or 8 bytes of ROM memory. This is not acceptable for the large numbers of tables used in an ECU.

With TargetLink, you can store the table data in a scaled integer type and interpolate the table entries in a floating-point type:

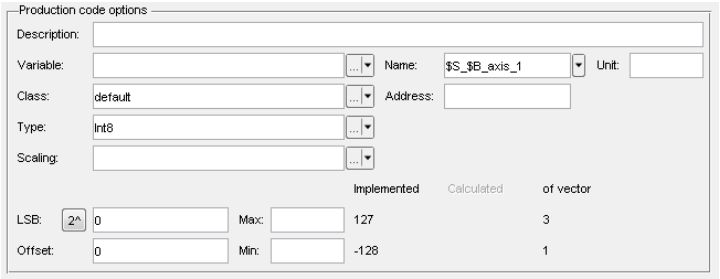
- The table search is performed with fixed-point data types, which requires that the table data is scaled. The additional data for the LSB and offset of the table vectors is stored in the associated table data struct. The look-up functions include some additional code to scale the floating-point input to fixed-point and to rescale the interpolation result to floating-point. The LSBs of the input

vectors for 1-D tables and 2-D tables are stored as reciprocal values (1/LSB). This results in more efficient scaling code.

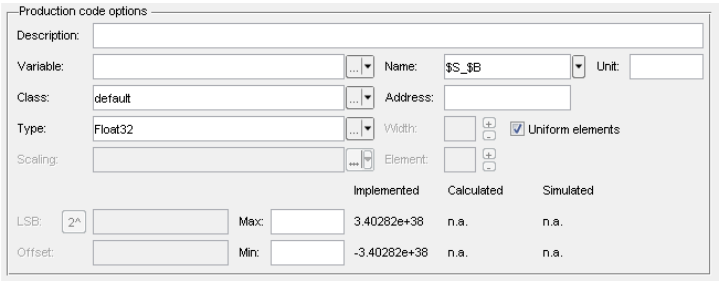
- The linear interpolation of the table entries is done in floating-point to increase the accuracy of the result and to decrease the execution time.

This special version of look-up tables is generated by the Code Generator automatically when the following conditions are set in the Production code options group box:

- For 1-D tables the x-axis vector and y-axis vector are set to an integer data type. For 2-D tables the row/column-axis vector and the table matrix are set to an integer data type.



- The output variable of the Look-Up Table block is set to a floating-point data type.



Tip

The particular code is generated even if you do not use a floating-point CPU. In this case the compiler performs the floating-point calculation with the compiler's floating-point library.

Optimizing Control Flows

Optimizing control flows

TargetLink can optimize control flows in production code.

Where to go from here

Information in this section

Simplifying If And Switch Statements..... 543
Moving Code Into Conditional Branches..... 543
Merging Loops..... 545
Combining Code of If Statements..... 547

Simplifying If And Switch Statements

Removing unreachable branches

TargetLink replaces the control flow statement by one of its branches or removes the statement entirely if one of the following conditions is fulfilled:

- The controlling expression is constant.
- The controlling expression's range can be uniquely determined.

Removing unnecessary control flow statements

TargetLink can remove unnecessary control flow statements:

- If all branches are empty, TargetLink removes the entire control flow statement.
- If the controlling expression contains side effects, TargetLink substitutes the control flow statement by an assignment of the controlling expression to an auxiliary variable.

Moving Code Into Conditional Branches

Moving code into conditional branches

Code is moved into conditional branches whenever possible to avoid superfluous calculations. This optimization technique is determined by the following Optimization property values:

- MOVABLE (definable for variable classes)
- MOVABLE (definable for function classes)
- SIDE_EFFECT_FREE (definable for function classes)

For details on the mentioned property values, refer to Basics on Optimizing Variables on page 504 and Basics on Optimizing Functions via Property Values on page 524 respectively.

In the following example, the variable C is evaluated before the `if` statement.

```
Int A, B, C;
Int conditional_result;
Int out;
A = ...;
B = ...;
C = 5 + A + B;
conditional_result = 7;
if (A > B) {
 conditional_result = C;
}
out = conditional_result;
```

After optimization, the variable C is moved to the `then` branch of the `if` statement.

```
Int A, B, C;
Int conditional_result;
Int out;
A = ...;
B = ...;
conditional_result = 7;
if (A > B) {
 C = 5 + A + B;
 conditional_result = C;
}
out = conditional_result;
```

If the variable C is specified as `ERASABLE`, it will be optimized next.

Note

Block variables whose **Optimization** property is set to `MERGEABLE` potentially result in data flow connections that are not visible in the block diagram. By default, TargetLink does not move block code associated with one of these variables into conditional branches that are based on block diagram data flow. TargetLink usually moves parts or all of this block code during C-code-based optimization if the data flow implied by data store memory and `MERGEABLE` permits this, but is not guaranteed to do so.

If you are sure that you specified only those variables as `MERGEABLE` that can be safely moved, you can instruct TargetLink to perform further analysis on the C code by setting the ConsiderMergeableForBlockDiagramBasedSwitchOptimization Code Generator option to `off`. This reduces the duration of code generation and can improve code efficiency.

Usually, only block parameters can safely be specifed as `MERGEABLE`. When in doubt, do not specify block variables as `MERGEABLE`.

Related topics

References

ConsiderMergeableForBlockDiagramBasedSwitchOptimization (📖 TargetLink Model Element Reference)

Merging Loops

Merging consecutive loops

TargetLink can merge consecutive `for` loops in the same control flow branch if all of the following conditions are fulfilled:

- The for loops have the same iteration range.
- The loop variable is used as follows:
  - It is used exclusively inside the `for` loop.
  - It is accessed only in the `for` loops's initialization, condition, and update statements.

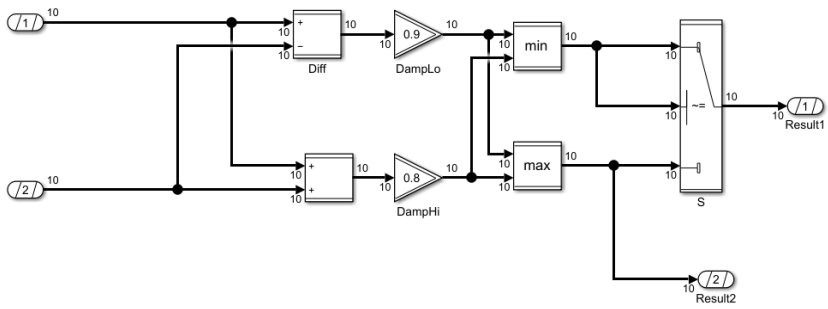
In addition, consecutive `for` loops of a different iteration range can also be merged if they can be made compatible via an offset.

Loop merging is primarily intended for `for` loops resulting from the processing of non-scalar variables but works for certain other loops as well.

Merging non-consecutive loops

If you set the CombineControlFlowStatements Code Generator option to on, TargetLink tries to merge loops that are non-consecutive. This works even when logging via log macros is enabled.

Modeling example Consider the following exemplary modeling situation:



Unmerged Non-Consecutive Loops	Merged Non-Consecutive Loops

`for (Aux_S32 = 0; Aux_S32 < 10; Aux_S32++)`
`{`
`  /* Sum: Logging/Diff */`
`  Diff[Aux_S32] = Input1[Aux_S32] - Input2[Aux_S32];`
`}`
`LOG_VEC(a, _Diff, 10, Diff);`
`for (Aux_S32 = 0; Aux_S32 < 10; Aux_S32++)`
	`for (Aux_S32 = 0; Aux_S32 < 10; Aux_S32++)`
	`{`
	`  /* Sum: Logging/Diff */`
	`  Diff[Aux_S32] = Input1[Aux_S32] - Input2[Aux_S32];`
	
	`  /* Gain: Logging/DampLo */`
	`  DampLo[Aux_S32] = Diff[Aux_S32] * 0.9F;`

Unmerged Non-Consecutive Loops

```
{
  /* Gain: Logging/DampLo */
  DampLo[Aux_S32] = Diff[Aux_S32] * 0.9F;
}
LOG_VEC(a, _DampLo, 10, DampLo);
for (Aux_S32 = 0; Aux_S32 < 10; Aux_S32++)
{
  /* Sum: Logging/Sum */
  Sum[Aux_S32] = Input1[Aux_S32] + Input2[Aux_S32];
}
LOG_VEC(a, _Sum, 10, Sum);
for (Aux_S32 = 0; Aux_S32 < 10; Aux_S32++)
{
  /* Gain: Logging/DampHi */
  DampHi[Aux_S32] = Sum[Aux_S32] * 0.8F;
}
LOG_VEC(a, _DampHi, 10, DampHi);
for (Aux_S32 = 0; Aux_S32 < 10; Aux_S32++)
{
  /* MinMax: Logging/MaxMinMax: Logging/Max */
  if (DampLo[Aux_S32] > DampHi[Aux_S32]) {
     Max[Aux_S32] = DampLo[Aux_S32];
  }
  else {
     Max[Aux_S32] = DampHi[Aux_S32];
  }
}
LOG_VEC(a, _Max, 10, Max);
for (Aux_S32 = 0; Aux_S32 < 10; Aux_S32++)
{
  /* TargetLink outport: Logging/Result2 */
  Result2[Aux_S32] = Max[Aux_S32];

  /* MinMax: Logging/MinMinMax: Logging/Min */
  if (DampLo[Aux_S32] < DampHi[Aux_S32]) {
     Min[Aux_S32] = DampLo[Aux_S32];
  }
  else {
     Min[Aux_S32] = DampHi[Aux_S32];
  }
}
LOG_VEC(a, _Min, 10, Min);
for (Aux_S32 = 0; Aux_S32 < 10; Aux_S32++)
{
  /* Switch: Logging/S */
  if (Min[Aux_S32] != 0.F) {
     /* # combined # TargetLink outport:
Logging/Result1 */
     Result1[Aux_S32] = Min[Aux_S32];
  }
  else {
     /* # combined # TargetLink outport:
Logging/Result1 */
     Result1[Aux_S32] = Max[Aux_S32];
  }
}
```

Merged Non-Consecutive Loops

```
  /* Sum: Logging/Sum */
  Sum[Aux_S32] = Input1[Aux_S32] + Input2[Aux_S32];

  /* Gain: Logging/DampHi */
  DampHi[Aux_S32] = Sum[Aux_S32] * 0.8F;

  /* MinMax: Logging/MaxMinMax: Logging/Max */
  if (DampLo[Aux_S32] > DampHi[Aux_S32]) {
     Max[Aux_S32] = DampLo[Aux_S32];
  }
  else {
     Max[Aux_S32] = DampHi[Aux_S32];
  }

  /* TargetLink outport: Logging/Result2 */
  Result2[Aux_S32] = Max[Aux_S32];

  /* MinMax: Logging/MinMinMax: Logging/Min */
  if (DampLo[Aux_S32] < DampHi[Aux_S32]) {
     Min[Aux_S32] = DampLo[Aux_S32];
  }
  else {
     Min[Aux_S32] = DampHi[Aux_S32];
  }

  /* Switch: Logging/S */
  if (Min[Aux_S32] != 0.F) {
     /* # combined # TargetLink outport: Logging/Result1 */
     Result1[Aux_S32] = Min[Aux_S32];
  }
  else {
     /* # combined # TargetLink outport: Logging/Result1 */
     Result1[Aux_S32] = Max[Aux_S32];
  }
}
LOG_VEC(a, _Diff, 10, Diff);
LOG_VEC(a, _DampLo, 10, DampLo);
LOG_VEC(a, _Sum, 10, Sum);
LOG_VEC(a, _DampHi, 10, DampHi);
LOG_VEC(a, _Max, 10, Max);
LOG_VEC(a, _Min, 10, Min);
}
```

TargetLink reduces the number of `for` loops from seven to one and gatheres the log macros at the end of the remaining `for` loop.

Related topics

References

CombineControlFlowStatements (TargetLink Model Element Reference)

Combining Code of If Statements

Introduction

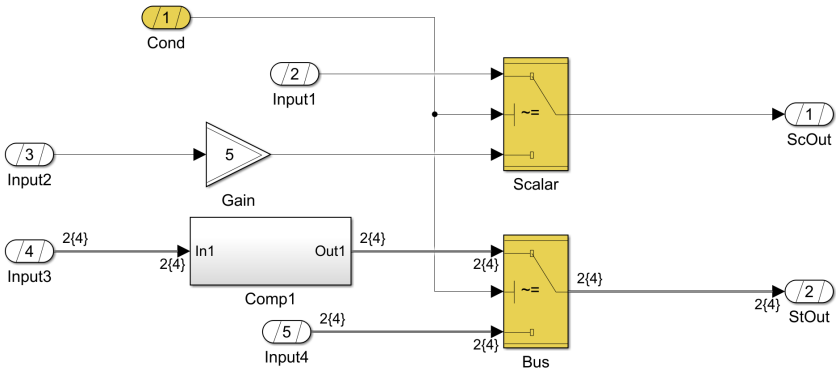
When modeling control flows that depend on the *same* or *negated* condition, you can use modeling patterns that increase recognizability. This can conflict with the goal of modeling for efficient code.

TargetLink can optimize the code resulting from such modeling constellations if you set the **CombineControlFlowStatements** Code Generator option to **on**. Exemplary modeling constellations are:

- Switch blocks driven by the same condition on page 547
- Enabled subsystems driven by a condition or its negation on page 548
- Nested conditions on page 549

Switch blocks driven by the same condition

The following modeling pattern shows two **Switch** blocks that are driven by the same **Cond** condition:



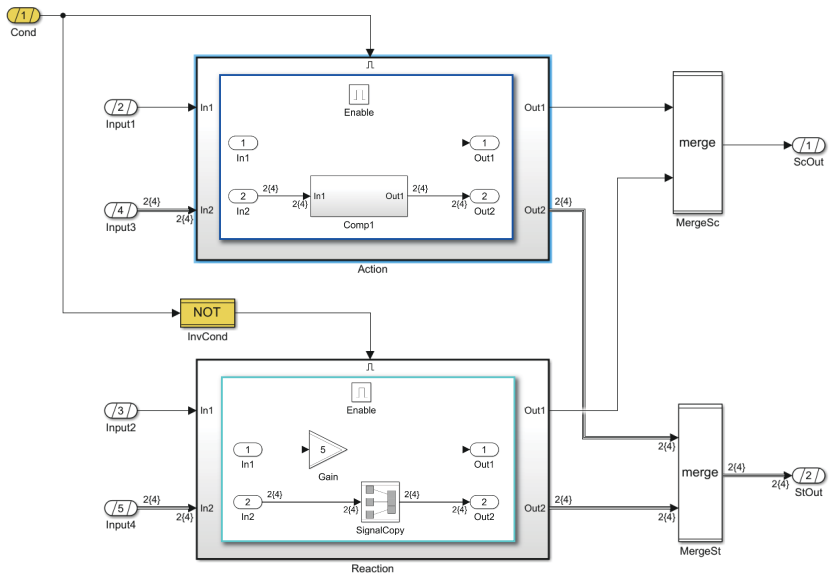
This improves the recognizability of the data inputs. The production code looks as follows:

Original Control Flow	Combined Control Flow

`if (Cond) {`
`  ...`
`  StOut = ...`
`}`
`else {`
`  StOut = Input4;`
`}`
 	
`if (Cond) {`
`  ScOut = Input1;`
`}`
`else {`
`  ScOut = Input2 * 5;`
`}`
	`if (Cond) {`
	`  ...`
	`  StOut = ...`
	
	`  ScOut = Input1;`
	`}`
	`else {`
	`  StOut = Input4;`
	
	`  ScOut = Input2 * 5;`
	`}`

Enabled subsystems driven by a condition or its negation

The following modeling pattern shows two enabled subsystems. One is driven by the Cond condition, the other by the condition's negation, which is modeled via a Logical Operator block called `InvCond`:



This lets you group several precalculations. The production code looks as follows:

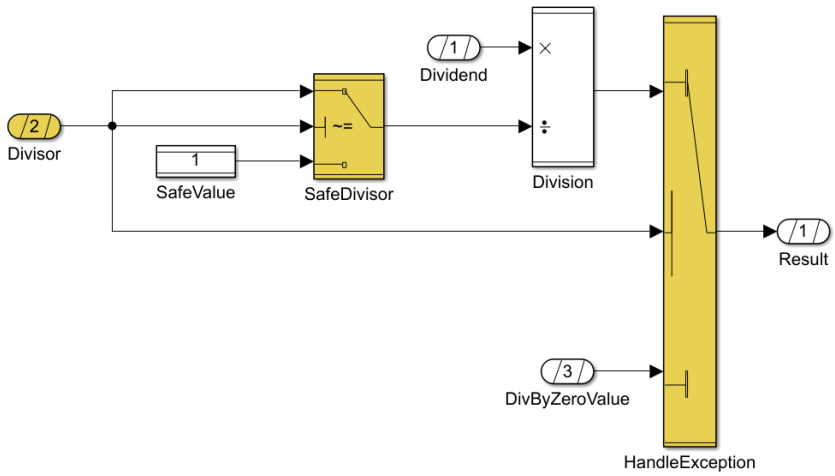
Original Control Flow[1)]	Combined Control Flow

`if (Cond) {`
`  ScOut = Input1;`
 	
`  ...`
`  StOut = ...`
`}`
 	
`if (!(Cond)) {`
`  ScOut = Input2 * 5;`
 	
`  StOut = Input4;`
`}`
	`if (Cond) {`
	`  ...`
	`  StOut = ...`
	
	`  ScOut = Input1;`
	`}`
	`else {`
	`  StOut = Input4;`
	
	`  ScOut = Input2 * 5;`
	`}`

1) If you provided the same initial value for the block variables of the Merge and OutPort/Bus Outport blocks. The order of the calculations might vary.

Nested conditions

The following modeling pattern shows two Switch blocks that are driven by the same condition called `Divisor`, but the condition is nested:



In this case, the first data input of the second Switch block, called `HandleException`, uses the output of the first Switch block, called `SafeDivisor`, which is driven by the same condition as `HandleException`. The production code looks as follows:

Original Control Flow	Combined Control Flow

`if (Divisor != 0.F) {`
`  Float32 SafeDivisor;`
 	
`  if (Divisor != 0.F) {`
`  SafeDivisor = Divisor;`
`  }`
`  else {`
`  SafeDivisor = 1.F;`
`  }`
 	
`  HandleException = Dividend / SafeDivisor;`
`}`
`else {`
`  HandleException = DivByZeroValue;`
`}`
	`if (Divisor != 0.F) {`
	`  HandleException = Dividend / Divisor;`
	`}`
	`else {`
	`  HandleException = DivByZeroValue;`
	`}`

Related topics

Basics

Basics on Configuring the Code Generator for Production Code Generation..... 49

References

Bus Outport Block (📖 TargetLink Model Element Reference)
CombineControlFlowStatements (📖 TargetLink Model Element Reference)
Merge Block (📖 TargetLink Model Element Reference)
OutPort Block (📖 TargetLink Model Element Reference)
Switch Block (📖 TargetLink Model Element Reference)

Optimizing via Range Propagation

Basics on Optimizing the Code via Range Propagation

Range propagation

TargetLink's range propagation mechanism can exploit range information to select the optimum code during the code generation process. Each output of a TargetLink block has a floating-point or fixed-point range that results from the arithmetic or Boolean operation of the block. The Code Generator performs the model computation, including the overflow behavior, on the host PC or on the target.

During code generation, TargetLink propagates the available range information throughout the model. Atomic subsystems and functions, which will be inlined if the Code Generator option **InliningThreshold** is sufficiently high, are not treated as range propagation analysis boundaries. Range propagation considers variables which belong to a variable class with the **Info** property set to **readwrite** or **adjustable** as capable of changing their value between (writing and reading) accesses. Interblock optimization does not influence range propagation.

Tip

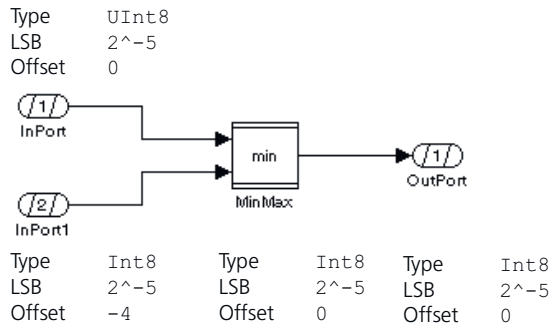
If possible, specify constrained limits for the inports of a TargetLink subsystem and for the blocks in a subsystem. These limits are used for the range propagation mechanism. The second example below shows how constrained limits facilitate range propagation.

Note

Integer operations may exceed the 64-bit integer range; for example, if three 32-bit input signals are multiplied. In this case, the range propagation mechanism saturates the range to Int64. This does not apply to floating-point operations.

Optimizing the code of a MinMax block

This example shows how the Code Generator optimizes the code for a **MinMax** block. Since the block output of the **InPort** block (Min/Max: 0 .. 7.96875 is always greater than the output of the **InPort1** block (Min/Max: -8 .. -0.03125, the latter can be eliminated from the generated code.

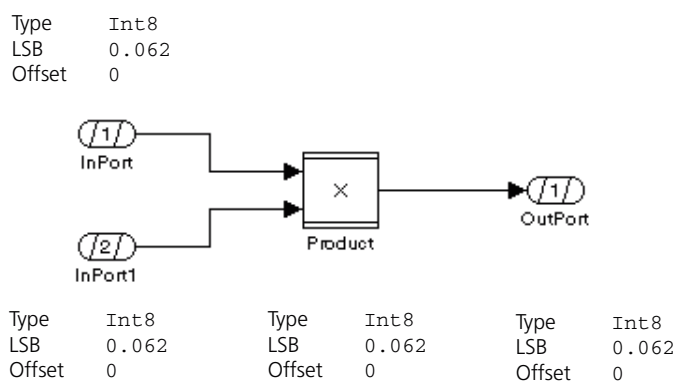


The generated code looks like this:

```
UInt8 Sa1_InPort /* LSB: 2^-5 OFF:  0 MIN/MAX:  0 .. 7.96875 */;
Int8 Sa1_OutPort /* LSB: 2^-5 OFF:  0 MIN/MAX:  -4 .. 3.96875 */;
...
void Subsystem(Int8 * Sa1_OutPort)
{
   /* SLLocal: Default storage class for local variables | Width: 8 */
   Int8 Sa1_MINMAX /* LSB: 2^-5 OFF:  0 MIN/MAX:  -4 .. 3.96875 */;
   /* MinMax: Subsystem/MINMAX
      Subsystem/MINMAX: Signal of the first input is always
      greater than the second input signal. Only
      using the first input signal. */
   Sa1_MINMAX = (Int8) Sa1_InPort;
   LOG_VAR(a, _Sa1_MINMAX, Sa1_MINMAX);
   /* TargetLink outport: Subsystem/OutPort */
   Sa1_OutPort = Sa1_MINMAX;
}
```

Using constrained limits for range propagation

This example shows how the Code Generator uses constrained limits for range propagation. The data types of the InPort and InPort1 blocks are cast to Int16 before the multiplication. No constrained limits have been specified for the output of two inports. As a result, the multiplication has to be performed in 32 bit.



The generated code looks like this:

```
/* TargetLink outport: Subsystem/OutPort
# combined # Product: Subsystem/Product */
Sa1_OutPort = (Int8) ((((Int32) (((Int16) Sa1_InPort) *
((Int16)Sa1_InPort1))) * 31) / 500);
```

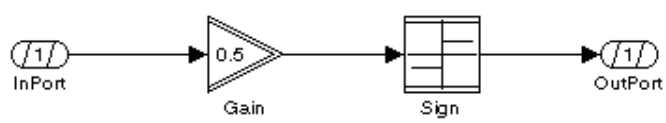
Suppose you know that the output of the two **InPort** blocks never exceeds the range [–1,1]. After you enter this range in the constrained limits edit field and generate code, the code looks like this:

```
/* TargetLink outport: Subsystem/OutPort
# combined # Product: Subsystem/Product */
Sa1_OutPort = (Int8) ((((Int16) (((Int16) Sa1_InPort) *
((Int16) Sa1_InPort1))) * 31) / 500);
```

The multiplication can now be performed in 16 bit.

Analyzing block inputs and outputs for code simplification

In the example below, the gain value of the **Gain** block (data type: Int16, LSB: 2^0) is 0.5. Thus, its output cannot overflow and the input of the **Sign** block always is positive or zero. The implementation of the **Sign** block in the generated code can therefore be simplified.

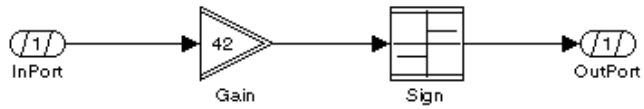


Type	UInt16		Type	UInt16
LSB	2^0		LSB	2^0
Offset	0		Offset	0

The generated code looks like this:

```
UInt16 Sa1_InPort;
Int16 Sa1_OutPort;
...
void Subsystem(void)
{
   /* Sign: Subsystem/Sign
      Subsystem/Sign: Input always positive or zero.
      # combined # Gain: Subsystem/Gain */
   Sa1_OutPort = ((Int16) (Sa1_InPort >> 1)) > 0/* 0. */;
}
```

In the example below, the gain value of the **Gain** block (data type: Int16, LSB: 2^0, offset: 0) is 42. Since the data type of its input is UInt16 (LSB: 2^0, offset: 0) the **Gain** block's output might overflow. Due to the *compute-through-overflows* behavior of unsaturated integer values, the input of the **Sign** block might become negative. The implementation of the block in the generated code therefore cannot be simplified.



Type	UInt16	Type	UInt16
LSB	2^0	LSB	2^0
Offset	0	Offset	0

The generated code looks like this:

```
UInt16 Sa1_InPort;
Int16 Sa1_OutPort;
...
void Subsystem(void)
{
   /* SLLocal: Default storage class for local variables | Width: 16
*/
   Int16 Sa1_Gain;
   /* Gain: Subsystem/Gain */
   Sa1_Gain = ((Int16) Sa1_InPort) * 42;
   /* Sign: Subsystem/Sign */
   if (Sa1_Gain < 0) {
      /* # combined # TargetLink outport: Subsystem/OutPort */
      Sa1_OutPort = -1/* -1. */;
   }
   else {
      /* # combined # TargetLink outport: Subsystem/OutPort */
      Sa1_OutPort = (Int16) (Sa1_Gain > 0 /* 0. */);
}
```

Related topics

Basics

Basics on Worst-Case Range Calculation (📖 TargetLink Preparation and Simulation Guide)

References

InliningThreshold (📖 TargetLink Model Element Reference)

Optimizing Bitfields for Booleans and Stateflow States

Basics on Collecting Booleans and Stateflow States in Global Bitfield Structures

Global bitfields for optimization

TargetLink can optimize the code by collecting booleans or Stateflow states in global bitfield structures that are implicitly generated during code generation.

Because these booleans or states can originate from different contexts, TargetLink might create different bitfield structures:

Variable's			Name Template
Origin	**Scope**	**Initialization**	
Simulink	Global	No	`GBFS_$I$R_$(Task)`
		Yes	`GIBFS_$I$R_$(Task)`
Stateflow	Global	No	`SBFS_$O_$I$R`
		Yes	`SIBFS_$O_$I$R`

For details on customizing the bitfield structure's names, refer to Global-Bitfield-Related StructSpec Strings and Basics on Using Name Macros on page 78.

Collecting booleans

Note

This only works for booleans specified at TargetLink blocks or in Stateflow. Booleans specified in the Data Dictionary that are referenced at TargetLink blocks are not supported.

Booleans will be collected in global bitfield structures under the following circumstances:

- The UseGlobalBitfields Code Generator option is set to `on`
- The variable is scalar.
- The variable's type is the `Bool` base type.
- The variable class is set to `default`.

Removing single variables from the global bitfield structure You can remove single variables from the global bitfield structures by setting their variable class to `OPT_LOCAL`. This instructs TargetLink to treat these variables as local and fully optimizable. You can also set any other variable class.

Collecting Stateflow states

Stateflow states will be collected in global bitfield structures if the **StateflowUseBitfields** (📖 TargetLink Model Element Reference) Code Generator option is set to `on`.

Related topics

Basics

Basics on Arranging Data in Structured Variables..... 148

References

StateflowUseBitfields (📖 TargetLink Model Element Reference)
UseGlobalBitfields (📖 TargetLink Model Element Reference)

Optimizing Code Efficiency via Memory-Specific Pointers

Where to go from here

Information in this section

Basics on Memory-Specific Pointers..... 557

How to Enhance the AddressQualifiers XML object..... 559

Basics on Memory-Specific Pointers

Memory-specific pointers

Memory-specific pointers are pointers with explicit memory type qualifiers such as `far`, `near`, or `huge`. A memory-specific pointer generates more efficient code for each variable access and is often used in segmented memory architectures. For example, a far pointer can be used to mark objects in 16-bit targets that are placed at a memory location with an address of more than 16 bits.

Memory-specific pointers in TargetLink

In TargetLink, a pointer is declared as a memory-specific pointer by specifying address qualifiers. Address qualifiers are target-specific and contain the explicit memory type qualifier such as `far`, `near`, or `huge`. Address qualifiers are specified via type prefixes in the Data Dictionary.

Type prefixes become address qualifiers only if they are also specified as AddressQualifier for the current code generation target. The code generation target references a `TargetConfig.xml` file. If the used `TargetConfig.xml` file does not contain address qualifiers, the `AddressQualifiers` XML object must be enhanced with the memory type qualifier. Refer to How to Enhance the AddressQualifiers XML object on page 559.

Note

To specify address qualifiers for platforms that are not directly supported by the Target Simulation Module (TSM), refer to the Generating TargetLink Code and ASAP2 Files for Arbitrary Platform application note, which you can download from the TargetLink Support Center.

Specifying address qualifiers

You specify address qualifiers as follows:

- Set the explicit memory type qualifier via the type prefix.
- Select a code generation target with the specified memory type qualifiers.

Set memory type qualifier via the type prefix You must set the explicit memory type qualifier, such as `far`, `near`, or `huge`, in the Data Dictionary via a type prefix. Refer to Example of Specifying Type Qualifiers via Type Prefixes on page 122.

Note

If you want to implement the address qualifier in postfix notation, contact dSPACE Support (www.dspace.com/go/supportrequest).

Select a code generation target with specified memory type qualifiers In the Preference Editor, select a code generation target with the specified memory type qualifiers. Refer to Basics on Using the Preferences Editor on page 22.

Related topics

Basics

Basics on Variable Classes 88

HowTos

How to Enhance the AddressQualifiers XML object 559

Examples

Example of Specifying Type Qualifiers via Type Prefixes 122

How to Enhance the AddressQualifiers XML object

Objective

To enhance the `AddressQualifiers` XML object with explicit memory type qualifiers such as `far`, `near`, or `huge`.

Preconditions

You cloned a `TargetConfig.xml` file by using the `TlTsmManager` utility. Refer to How to Clone Target/Compiler Combinations to Outside the TargetLink Installation on page 44.

Method

To enhance the AddressQualifiers XML object

1. Open the cloned `TargetConfig.xml` file.
2. Enhance the `AddressQualifiers` XML object with an explicit memory type qualifier and a description.

   In this example, you enhance the `AddressQualifiers` XML object with the memory type qualifier `far` and the description `Data that is accessed with the far addressing scheme`:

```
<ddObj name="AddressQualifiers" Kind="GenericObject">
    <ddObj name="far" Kind="GenericObject">
        <ddProperty Name="Description" Type="String">Data that is accessed with the far addressing scheme</ddProperty>
        <ddProperty Name="PrefixString" Type="String">far</ddProperty>
    </ddObj>
</ddObj>
```

3. You can repeat step 2 for each explicit memory type qualifier you want to be enhanced in the `TargetConfig.xml` file.
4. Save the modified `TargetConfig.xml` file.

Result

You enhanced the `AddressQualifiers` XML object with an explicit memory type qualifier.

Related topics

Basics

Basics on Memory-Specific Pointers..... 557

HowTos

How to Clone Target/Compiler Combinations to Outside the TargetLink Installation..... 44

Managing Variant Coding and Conditional Code Compilation

Introduction

To adapt the generated production code and its parameters to different types of vehicles without changing the model, TargetLink offers you the following two complementary methods.

- Variant coding of variable properties
- Conditional code compilation

Where to go from here

Information in this section

Introduction to Variant Coding and Conditional Code Compilation..... 562

Specifying Code Variants and Data Variants..... 564

Specifying Variable Vector Widths via Width Macros..... 586

Specifying Conditional Control Flows..... 616

Information in other sections

Modeling Control Flows without Using Stateflow (🕮 TargetLink Preparation and Simulation Guide)

Introduction to Variant Coding and Conditional Code Compilation

Introduction

This section introduces you to the basics of code and data variants, variable vector widths, and conditional control flows.

Basics on Variant Coding and Conditional Code Compilation

Variant definition

A variant describes a particular type of a product that arises from a common basis and differs in variant-specific characteristics, called variants. Variant coding allows you to implement variants according to the specified options, e.g., to the variable property values specified for a particular vehicle control application.

Reason for using variants

To handle the increasing number of variants in the field of vehicle control applications, variant coding of variable property values and conditional code compilation are two methods for adapting variants to the specific needs of a control application. Both methods make it easy to update production code and/or data for specific variants, i.e., they let you adapt the production code to be generated and its parameters to different types of vehicles with a minimum of modification work.

Methods for implementing variants

There are basically different types of variant coding and different methods to implement these types in the generated production code. The methods described below are supported by TargetLink and mainly differ in the time at which a variant is coded or activated, and where a variant is specified. With regard to the short activation time of a variant, these methods provide an increasing degree of flexibility.

Code variants When implementing code variants, you create variants at code generation time. From a set of alternative variable property values (variants) defined in the **TargetLink Data Dictionary** (DD), you specify a specific variant to be the active one during production code generation. For example, you can design different data types (Int16 and Float64) or scaling parameters to a variable before the code generation is started. Only one variant of a property is allowed to be active during code generation, otherwise an error occurs. Code variants cannot be switched during ECU run time.

Note

Only one code variant of a property value is allowed to be active during code generation, otherwise an error occurs.

Compilation variants When implementing compilation variants, you create variants at compile time using preprocessor directives (symbolized by a hash #). Preprocessor directives used for conditional code compilation are suitable to develop different variants within one C module, but only for compiling and thus for executing a specific part of the code while excluding other ones.

The following kinds of compilation variants are feasible:

- Conditional control flows:

  TargetLink supports different means to design control flows represented by preprocessor directives in the generated code. This is useful since it makes programs easier to develop, and easier to compile in different execution environments.

- Variable vector widths:

  Using a specific *width macro* ,TargetLink generates width-varying code for a vectorized variable, for example, `MyVar[NumberOfCylinders]` instead of `MyVar[8]`. The generated code suites different width contexts at code compilation time.

Data variants When implementing data variants, you can activate or deactivate code fragments at ECU run time. Data variants are specified in the TargetLink Data Dictionary (DD). They are generated into the same C code and are usually switched via a run-time changeable parameter, usually implemented as a calibratable switch variable. For example, the Value property of a gain parameter can have the variants 2, 3, and 4. The separate variants are selected via if-else or switch-case statements according to the run-time changeable parameter.

Note

Variables and functions must have fixed names independent of model elements or DD objects, i.e., do not use CGU-dependent name macros, such as **`$I`**.

Related topics

Basics

Specifying Code Variants..... 575
Specifying Conditional Control Flows..... 616
Specifying Data Variants..... 579
Specifying Variable Vector Widths via Width Macros..... 586

Specifying Code Variants and Data Variants

Introduction

This section explains the different variant types and describes when and how to use them.

Where to go from here

Information in this section

Preparing the Data Dictionary for Code Variants and Data Variants..... 564
Specifying Code Variants..... 575
Specifying Data Variants..... 579

Preparing the Data Dictionary for Code Variants and Data Variants

Introduction

This section makes you familiar with the principles of implementing code and data variants in the Data Dictionary.

Where to go from here

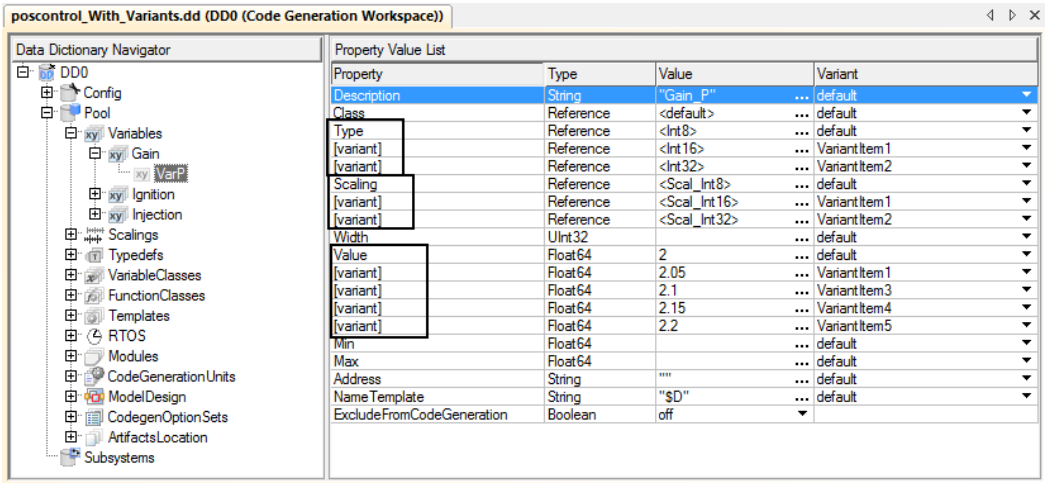
Information in this section

Basics on Variant Handling in the TargetLink Data Dictionary..... 564
How to Create Variant Items..... 568
How to Create Variant Configurations..... 570
How to Create Switch Variables for Data Variants..... 573

Basics on Variant Handling in the TargetLink Data Dictionary

Alternative property values

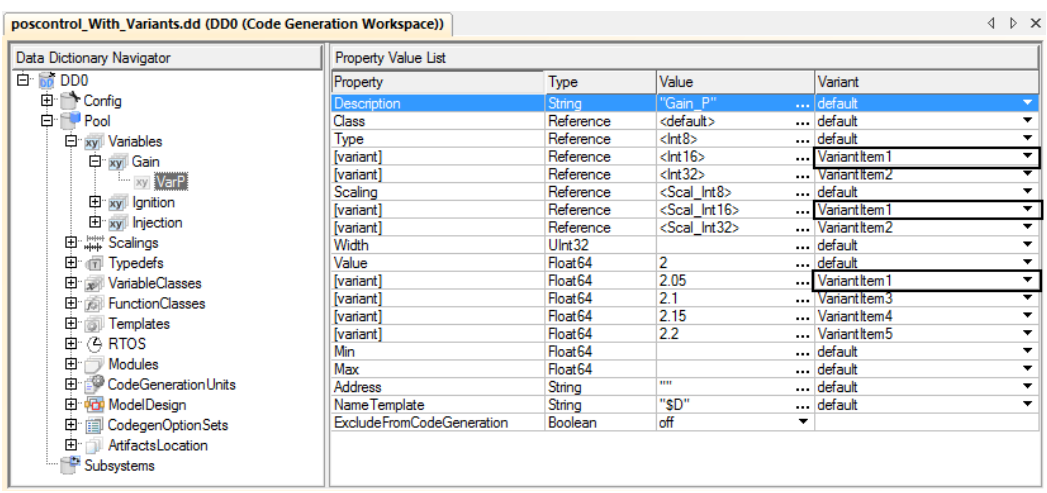
You often need alternative values (variants) for the properties of a variable object. TargetLink lets you specify such variants in the **TargetLink Data Dictionary** (DD), for example, **Type**, **Scaling**, or **Value**. Variant coding is possible only for variable objects in the DD. The DD lets you specify up to 127 variants for a property value.



Variant items used as flags for variant control

To control which variant is currently the active one, so-called variant items are used as flags. If a variant item is set, its related value becomes the active value of the property. If no variant item is set, the default value is active.

Suppose only VariantItem1 is set, then Type becomes Int16, Scaling becomes Scal_Int16, and Value becomes 2.05. All other variants are neglected because they are related to variant items that are not set.



Note

You cannot assign a variant item to a property value until you have defined the variant item. Refer to How to Create Variant Items on page 568.

Variant configuration as variant items container

Variant items cannot be set explicitly, but only implicitly, by activating a so-called variant configuration. A variant configuration acts as a container for

variant items. It commonly references a set of variant items. When you activate a variant configuration, all its residing variant items are set. You can define an arbitrary number of variant configurations, but only one variant configuration can be active and used for code generation and simulation at a time. For detailed instructions, refer to How to Create Variant Configurations on page 570 and How to Activate Code Variants on page 576.

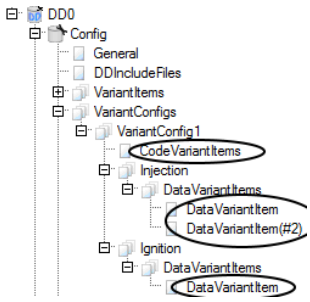
If **VariantConfig1** is active, comprising **VariantItem1**, **VariantItem3**, and **VariantItem4**, then **Type** becomes **Int16**, **Scaling** becomes **Scal_Int16**, and the **Value** property has the Data variants 2 and 3.

Property	Type	Value	Variant
Description	String	"Gain_P"	default
Class	Reference	<default>	default
Type	Reference	<Int8>	default
[variant]	Reference	<Int16>	VariantItem1
[variant]	Reference	<Int32>	VariantItem2
Scaling	Reference	<Scal_Int8>	default
[variant]	Reference	<Scal_Int16>	VariantItem1
[variant]	Reference	<Scal_Int32>	VariantItem2
Width	UInt32		default
Value	Float64	1	default
[variant]	Float64	2	VariantItem3
[variant]	Float64	3	VariantItem4
[variant]	Float64	4	VariantItem5
Min	Float64		default
Max	Float64		default
Address	String	""	default
NameTemplate	String	"$D"	default

VariantConfig1 = [VariantItem1, VariantItem3, VariantItem4]

Specification of variant configurations

Variant configurations are to be specified in the **Config** area of the Data Dictionary Manager.



After a variant item has been created in the **TargetLink Data Dictionary**, it is neutral. When you assign it to a variant configuration (for example, **VariantConfig1**), it becomes either a code variant item or a data variant item. This depends on the subfolder which you assign the variant item to (**CodeVariantItems** or **DataVariantItems** subfolder). For detailed instructions, refer to How to Create Variant Configurations on page 570.

Note

If you create a new variant configuration, it contains only a **CodeVariantItems** subfolder. If you need **DataVariantItems** subfolders, you must add them via the context menu.

Specification as code or data variant Whether a variant is a code or a data variant depends on the variant item. If a variant item belongs to a **CodeVariantItems** group in the variant configuration (see illustration above), the variant is a code variant. If a variant item belongs to a **DataVariantItems** group, it is a data variant. Data variants require a calibratable **Variable** object to switch the data variants at ECU run time.

The following table shows the variant kinds that can be assigned to each variable property in the **TargetLink Data Dictionary** (DD):

Variable Property	Code Variant	Data Variant
Description	✓	–
Class	✓	–
Type	✓	–
Scaling	✓	–
Width	✓	–
Value	✓	✓
Min	✓	–
Max	✓	–
Address	✓	–
NameTemplate	✓	–

Note

Value is the only variable property which a data variant item can be assigned to.

Process steps

To specify variants in the DD, perform the following steps:

1. Define variant items. These function as flags to activate variants. Refer to How to Create Variant Items on page 568.
2. Specify the variants. Each variant has a variant item assigned to it. Refer to How to Specify Code Variants on page 575 and How to Specify Data Variants on page 581.
3. Define a variant configuration as a container of specific variant items. Refer to How to Create Variant Configurations on page 570.
4. Define a switch variable for switching data variants during ECU run time, if you have specified data variants. Refer to How to Create Switch Variables for Data Variants on page 573.
5. Select the variant configuration to be used in code generation. Refer to How to Activate Code Variants on page 576.

Tip

You can vary the order of steps 2 to 4, only steps 1 and 5 are fixed.

Related topics

HowTos

How to Activate Code Variants..... 576
How to Specify Code Variants..... 575
How to Specify Data Variants..... 581

References

Permissible Properties for Variables (📖 TargetLink API Reference)

How to Create Variant Items

Objective

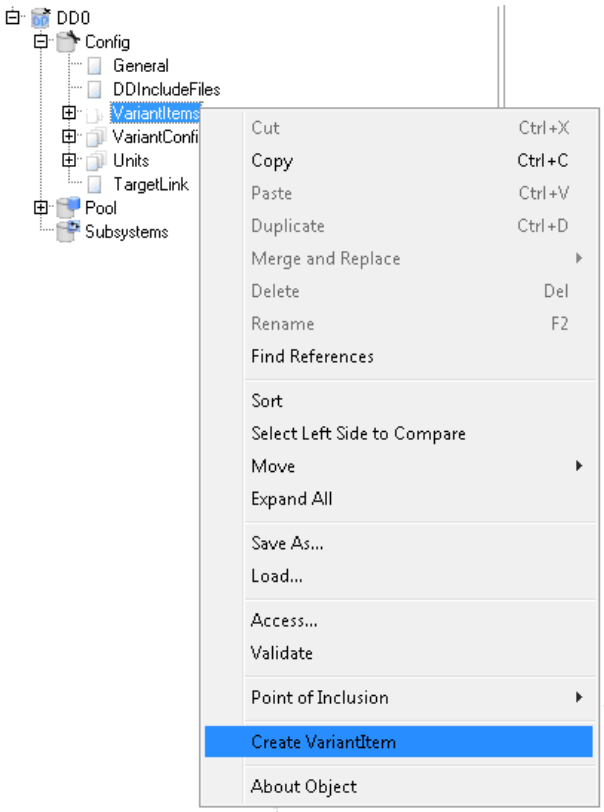
When variants are created, you must define variant items as they are required as flags.

Method

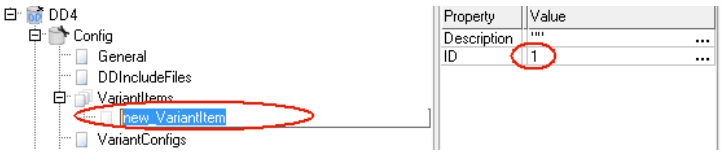
To create a variant item

1 In the DD object tree, select the /Config/VariantItems DD object.

2 From the context menu, select **Create VariantItem**.



3 Name the new variant item as needed and assign a unique ID number to it in the range 1...65535.



Result

A new variant item object is created in the DD object tree. The name of the new variant item appears in the Variant list, which is displayed in the property value list of each variable.

How to Create Variant Configurations

Objective

To activate variant items you must define a variant configuration.

Variant item container

A variant configuration acts as a container for variant items. When you activate a variant configuration, all its residing variant items are also activated.

Example

If VariantConfig1 is active, comprising VariantItem1, VariantItem3, and VariantItem4, then Type becomes Int16, Scaling becomes Scal_Int16, and the Value property has the Data variants 2 and 3.

Property	Type	Value	Variant
Description	String	"Gain_P"	default
Class	Reference	<default>	default
Type	Reference	<Int8>	default
[variant]	Reference	<Int16>	VariantItem1
[variant]	Reference	<Int32>	VariantItem2
Scaling	Reference	<Scal_Int8>	default
[variant]	Reference	<Scal_Int16>	VariantItem1
[variant]	Reference	<Scal_Int32>	VariantItem2
Width	UInt32		default
Value	Float64	1	default
[variant]	Float64	2	VariantItem3
[variant]	Float64	3	VariantItem4
[variant]	Float64	4	VariantItem5
Min	Float64		default
Max	Float64		default
Address	String	""	default
NameTemplate	String	"$D"	default

VariantConfig1 = [VariantItem1, VariantItem3, VariantItem4]

Restrictions

A variant item must not be referenced in more than one subfolder of a variant configuration.

Method

To create a variant configuration

1. In the DD object tree, select the `/Config/VariantConfigs` DD object.
2. From the context menu, select Create VariantConfig.

   The new variant configuration contains a CodeVariantItems subfolder but no DataVariantItems subfolder.

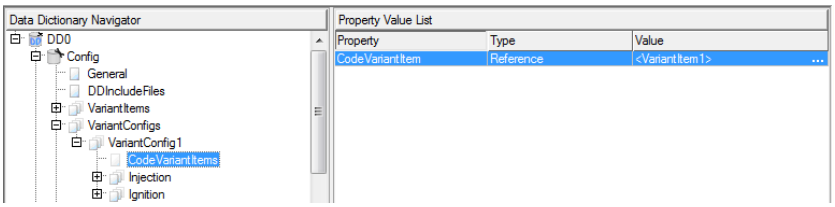
   Note

   You can define an arbitrary number of variant configurations, but only one variant configuration can be active and is used for code generation at a time.

3. Name the new variant configuration as needed, for example, VariantConfig1.
4. Select the CodeVariantItems sub node. From the context menu, select Create CodeVariantItem and reference a variant item on the Property Value List, for example, VariantItem1.

Note

The Show Unset Properties option in the View menu must be selected.



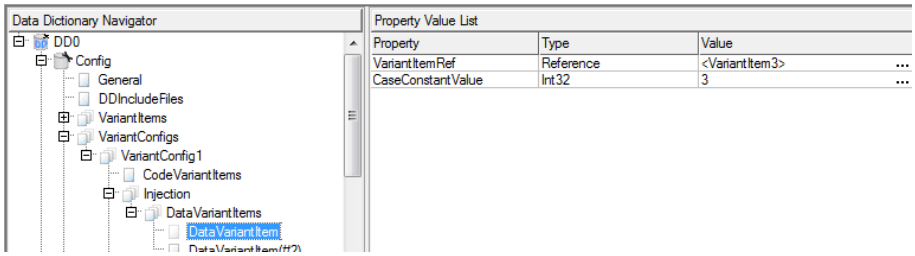
5 Select the VariantConfig object (for example, VariantConfig1), and select Create DataVariant from the context menu.

A new_DataVariant object is inserted.

6 Name the new data variant as needed, for example, Injection.

7 Go to its DataVariantItems sub node and select Create DataVariantItem from the context menu.

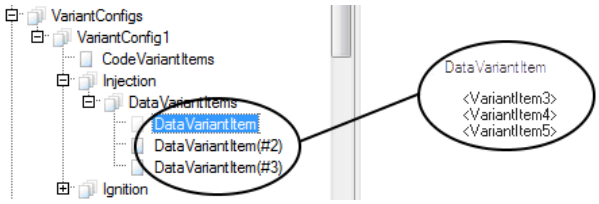
8 Reference a variant item and specify the CaseConstantValue property.



The CaseConstantValue specifies the value of the case statement of the related switch function in the C code, for example, 3 in `case 3`.

Note

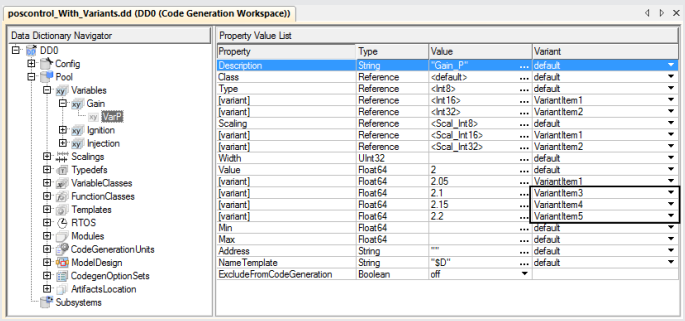
If a CaseConstantValue is not specified, the Code Generator takes the value of the ID property of the variant item into account.



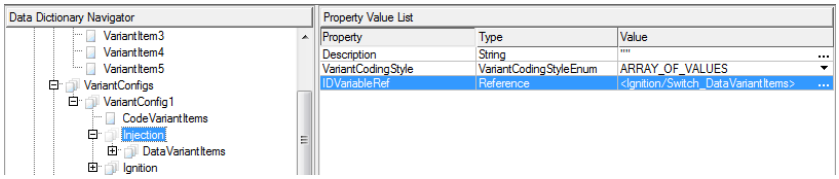
9 If you need to specify further data variant items, repeat the two preceding steps.

Note

The variant items of a data variant, as specified in `/Config/VariantConfigs/` (see illustration above), must be referenced in complete in the specification of the related property in `/Pool/Variables/` (see illustration below). Otherwise an error occurs during code generation. However, the variant items of more than one data variant can be referenced in `/Pool/Variables/`.



10 In the property value list of the new data variant, select the desired VariantCodingStyle and the associated IDVariableRef, for example:



Note

The ID variable must exist in the `/Pool/Variables` node in the TargetLink Data Dictionary. The value of the ID variable defines which variant is initially selected in the generated code.

Result

The new variant configuration appears in the DD object tree. Its name is also displayed in the Active Variant list.

Next step

If you have defined a variant configuration, you can easily select the variant needed for your current task. For details, refer to How to Activate Code Variants on page 576.

Related topics

HowTos

How to Activate Code Variants..... 576

How to Create Switch Variables for Data Variants

Objective

To switch between data variants (not code variants) during ECU run time you must define a switch variable. Switch variables must be specified as calibratable variables in the TargetLink Data Dictionary.

Restrictions

The following restrictions apply to a switch variable:

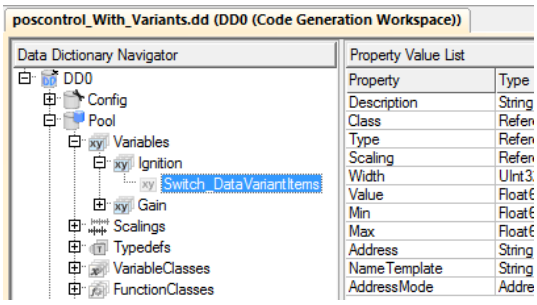
- It must not have variants itself.
- It must not be a struct.

Method

To create a switch variable for data variants

1 In the DD object tree, select the `/Pool/Variables DD object`.

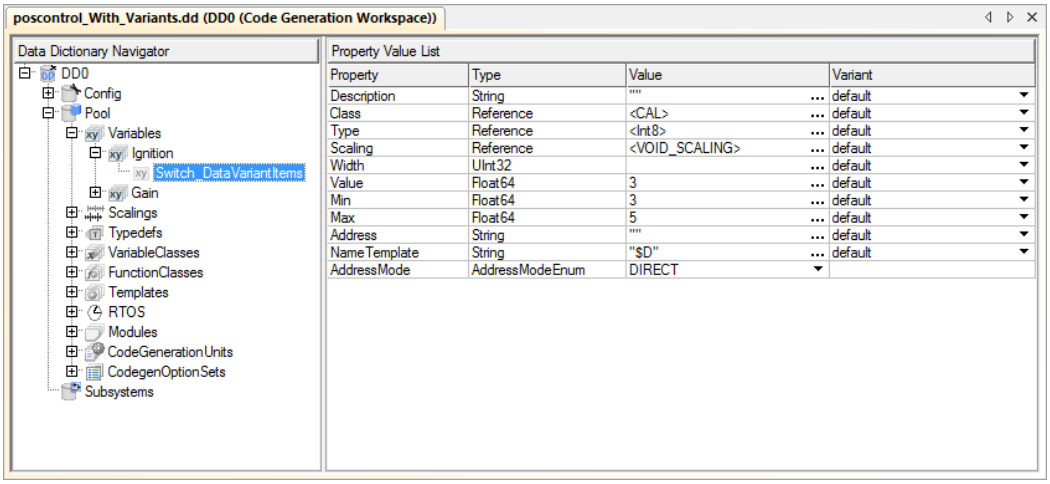
2 Via the **Variables** object context menu, create a **VariableGroup** object (e.g., *Ignition*) and then a **Variable** object (e.g., *Switch_DataVariantItems*).



3 Specify the variable:

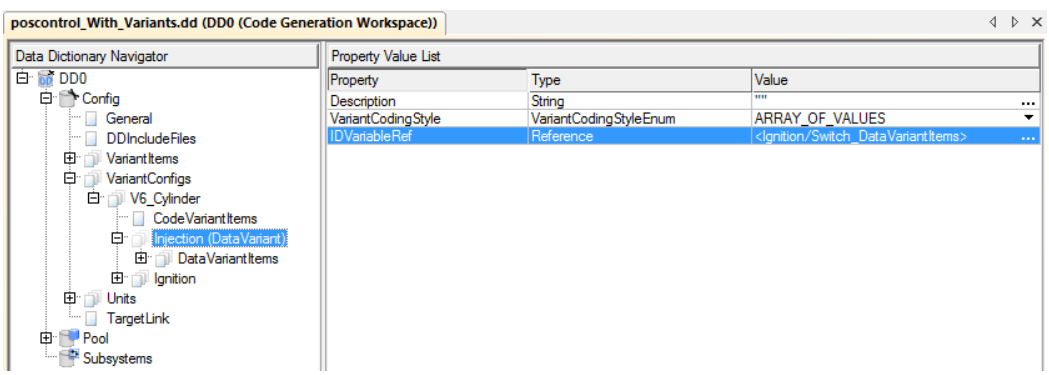
- The variable must have an integer data type, for example, `Int8`.
- The scaling must be set to `LSB = 2^0, offset = 0`. The predefined VOID_SCALING object provides these settings.
- The Class property must allow the variable to be calibratable.

- The Value property must equal the **CaseConstantValue** property of the DataVariantItem that is to be the initial one (if no **CaseConstantValue** is specified, it must equal the ID property of the VariantItem that is to be the initial one).
- The **Min** and **Max** properties must define a range that can take the ID numbers of all variant items that make up the data variants.



4 In the DD object tree, go to the data variant object, for example, `/Config/VariantConfigs/V6_Cylinder/Injection 8DataVariant)`.

5 Reference the switch variable: for example, reference the `/Pool/Variables/Ignition/Switch_DataVariantItems` variable as the value of the **IDVariableRef** property.



Result

You have defined a switch variable. You can use this (calibratable) variable to switch between data variants during ECU run time.

Note

If you switch between data variants at ECU run time, you must restart the application running on the ECU.

Specifying Code Variants

Introduction

To specify variants for the properties of a variable. Only one variant can be active during production code generation.

Where to go from here

Information in this section

How to Specify Code Variants..... 575

How to Activate Code Variants..... 576

How to Specify Code Variants

Objective

If you want alternative property values for a variable, for example, the values of the Type property, and you want a specific value to be the active one at code generation time, you need a code variant.

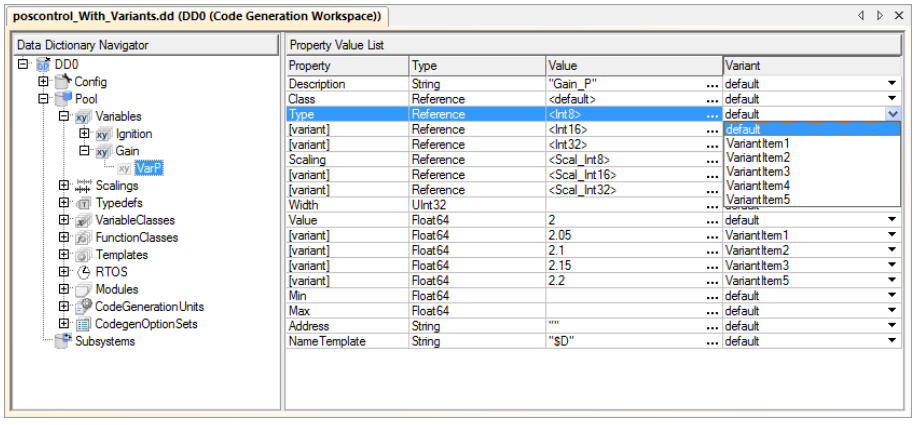
Preconditions

Appropriate variant items must already be defined. Refer to How to Create Variant Items on page 568.

Method

To specify a code variant

1. In the DD object tree, select the variable object, for example, `/Pool/Variables/Gain/VarP`.
2. In the Variant column, select a variant item from the list, for example:



A variant is inserted below the selected property.

Property Value List

Property	Type	Value	Variant
Description	String	"Gain_P" ...	default
Class	Reference	<default> ...	default
Type	Reference	<Int8> ...	default
[variant]	Reference	<Int16> ...	VariantItem1

3 Modify the property values as desired.

Property Value List

Property	Type	Value	Variant
Description	String	"Gain_P" ...	default
Class	Reference	<default> ...	default
Type	Reference	<Int8> ...	default
[variant]	Reference	Int16	VariantItem1

4 If you want to create a second variant, and so on, you must always select **default** arrow of the property row, not in a variant row. Otherwise, the existing variant item is replaced by the new one.

Result

You have specified a code variant.

Related topics

HowTos

How to Create Variant Items..... 568
How to Specify Data Variants..... 581

How to Activate Code Variants

Objective

If your DD project file contains several variant configurations, you must activate the one to be used during code generation and simulation.

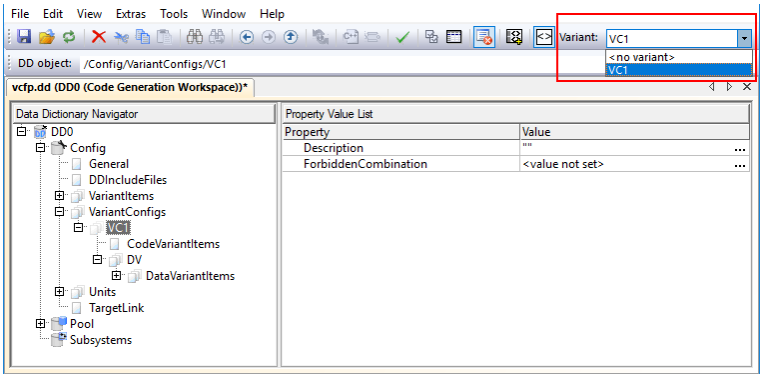
Precondition

You must have defined a variant configuration. Refer to How to Create Variant Configurations on page 570.

Method

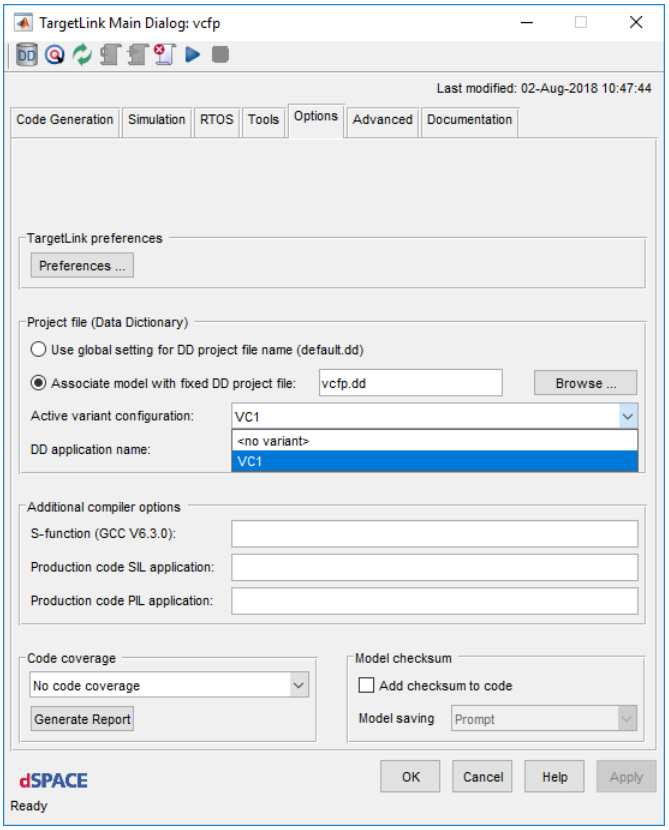
To activate code variants

1 In the Active Variant list of the Data Dictionary Manager, select the variant configuration you need.



Note

Alternatively, you can enter the command `dsdd('SetActiveVariant',<VariantConfigName>)` in the MATLAB Command Window or you can select the active variant configuration on the Options page of the TargetLink Main Dialog (see the following screenshot):



Result

TargetLink uses the active variant configuration during production code generation. The active variant configuration is stored in the Config area of the DD, and is written to the generated header and code files as well.

Note

After you activated another variant configuration in the DD, the Active variant field in the TargetLink Main Dialog is updated as soon as you close and reopen the TargetLink Main Dialog. After you activated another variant configuration in the TargetLink Main Dialog, the Active variant field in the DD is updated as soon as click in the DD object tree (left pane) or perform a Refresh (via menu View/Refresh, or shortcut F5).

Related topics

HowTos

How to Activate Data Variants..... 583
How to Create Variant Configurations..... 570

Specifying Data Variants

Introduction

To specify variants for the Value property of a variable. You can switch the variants at ECU run time.

Where to go from here

Information in this section

Basics on Coding Styles for Data Variants..... 579

How to Specify Data Variants..... 581

How to Activate Data Variants..... 583

Basics on Coding Styles for Data Variants

Variables of data variants

All variables belonging to one data variant item are collected in a struct, for example:

```
typedef struct Vca_DV_tag {
    const volatile UInt8 parameter1;
    const volatile UInt8 parameter2;
    const volatile UInt8 parameter3;
} Vca_DV_tp; /* Struct data type for data variant configuration DV */
```

One instance of the struct is generated for each variant item.

Note

When used with variant coding, structs representing look-up tables lead to structs of structs in the generated code. This is error-prone and might cause problems. You are therefore recommended to clear the Generate map structure option on the Table page to generate code for look-up tables without map structures when working with variant coding.

Supported coding styles

TargetLink supports two different variant coding styles:

Separate structs You can generate one separate struct for each data variant item, for example:

```
Vca_DV_tp Vca_DVA = {...}; /* for data variant item A */
Vca_DV_tp Vca_DVB = {...}; /* for data variant item B */
```

Array of structs You can generate an array of structs containing all instances, for example:

```
Vca_DV_tp Vca_DVA[2] = {...}; /* for both data variant items */
```

Switch variable

In the generated code, the variant-coded parameters are accessed indirectly by a pointer which references the selected variant struct. The switch variable is used to determine the active variant and to initialize the pointer accordingly. It is evaluated in the restart function.

Data variant coding

If the variant coding style is SEPARATE_STRUCTS and the switch variable is named varID, the following code is generated:

```
switch(varID) {
   case 1: {
        pVca_DV = &Vca_DVA;
        break;
   }
   case 2: {
        pVca_DV = &Vca_DVB;
        break;
   }
   default: {
        pVca_DV = &Vca_DVA;
        break;
   }
}
```

Tip

The name and class of the generated struct, and the pointer, can be defined using struct templates and variable templates in the Data Dictionary. Refer to Adapting Code to Company Coding Styles on page 77.

Note

You can specify the values of the case statements (for the switch function shown above: 1 and 2) via the CaseConstantValue property in the DD. For details refer to How to Create Variant Configurations on page 570.

Related topics

Basics

Adapting Code to Company Coding Styles..... 77

How to Specify Data Variants

Objective

If you want alternative values for a variable to be selectable at ECU run time, for example, alternative values of a gain parameter, you need to specify a data variant.

Preconditions

Appropriate variant items are already defined. Refer to How to Create Variant Items on page 568.

Restrictions

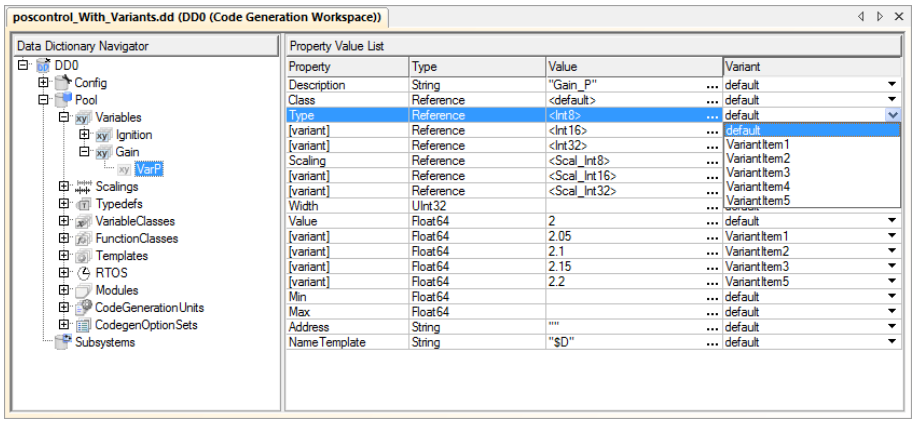
Data variants are possible only for the Value property of variable objects in the DD.

Method

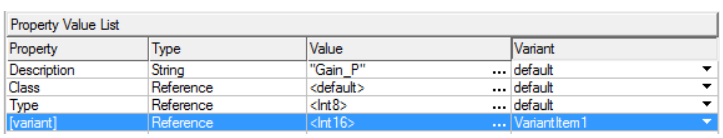
To specify a data variant

1 In the DD object tree, select the variable object, for example, `/Pool/Variables/Gain/VarP`.

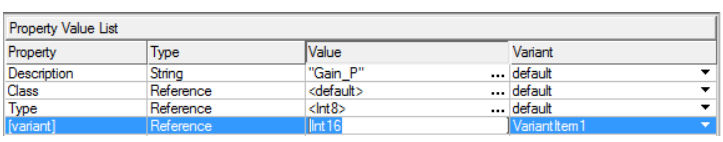
2 In the Variant column, select a variant item from the list, for example:



A variant is inserted below the selected property.



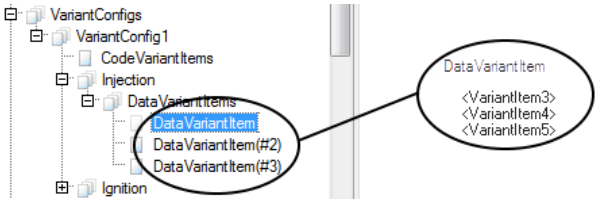
3 Modify the property values as desired.

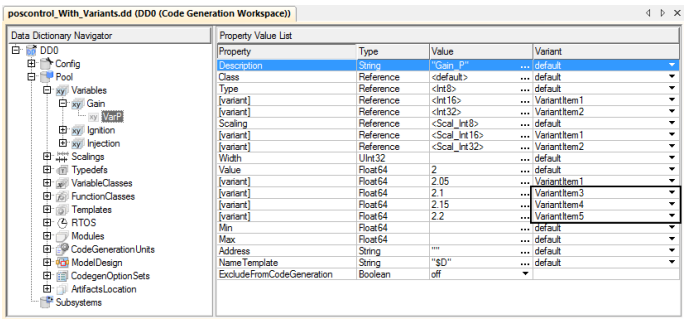


4 If you want to create a second variant, and so on, you must always select **default** arrow of the property row, not in a variant row. Otherwise, the existing variant item is replaced by the new one.

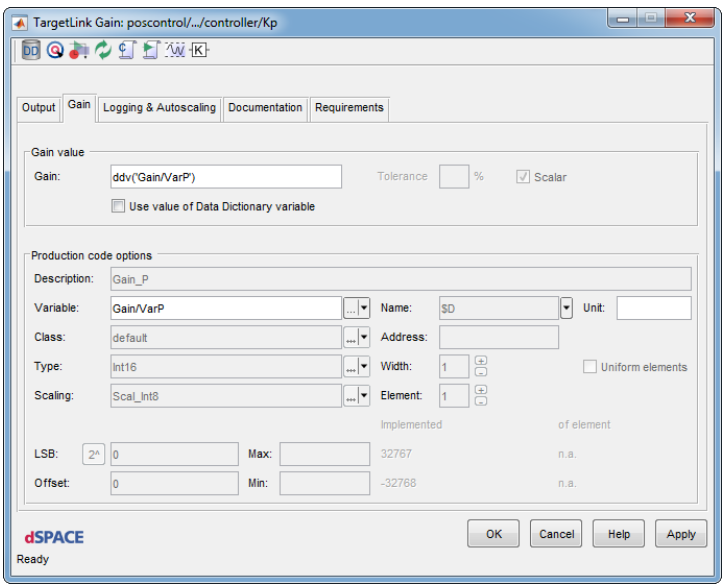
Note

The variant items of a data variant, as specified in `/Config/VariantConfigs/`, must be referenced in complete in the specification of the related property in `/Pool/Variables/`. Otherwise an error occurs during code generation. However, the variant items of more than one data variant can be referenced in `/Pool/Variables/`.





5 Enter `ddv('<ddObject>')` in the value edit field of the block parameter. For example, enter `ddv('Gain/VarP')` in the Gain edit field of a Gain block's dialog in your model. The API function `ddv('<ddObject>')` returns the value of a DD variable according to the active variant configuration.



Note

The value assigned to the initial variant item must equal the value specified in the dialog of the TargetLink block, otherwise an error occurs during code generation. The API function `ddv('<ddObject>')` guarantees that this requirement is fulfilled.

6 Click Apply and close the dialog.

Result

You have specified a data variant.

Related topics

HowTos

How to Create Variant Items..... 568
How to Specify Code Variants..... 575

References

ddv (📖 TargetLink API Reference)

How to Activate Data Variants

Objective

If your DD project file contains several variant configurations, you must activate the one to be used during code generation and simulation.

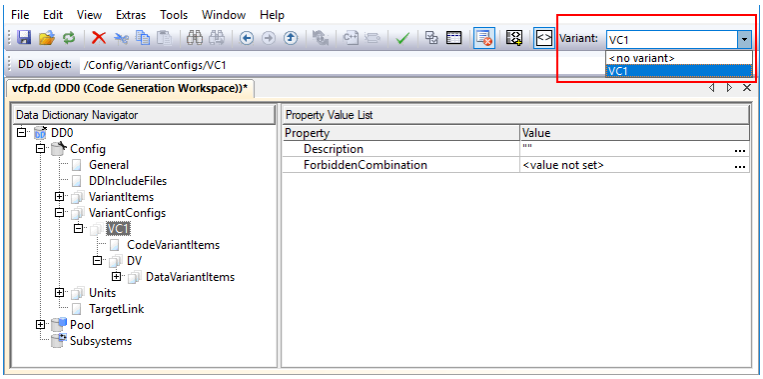
Precondition

You must have defined a variant configuration. Refer to How to Create Variant Configurations on page 570.

Method

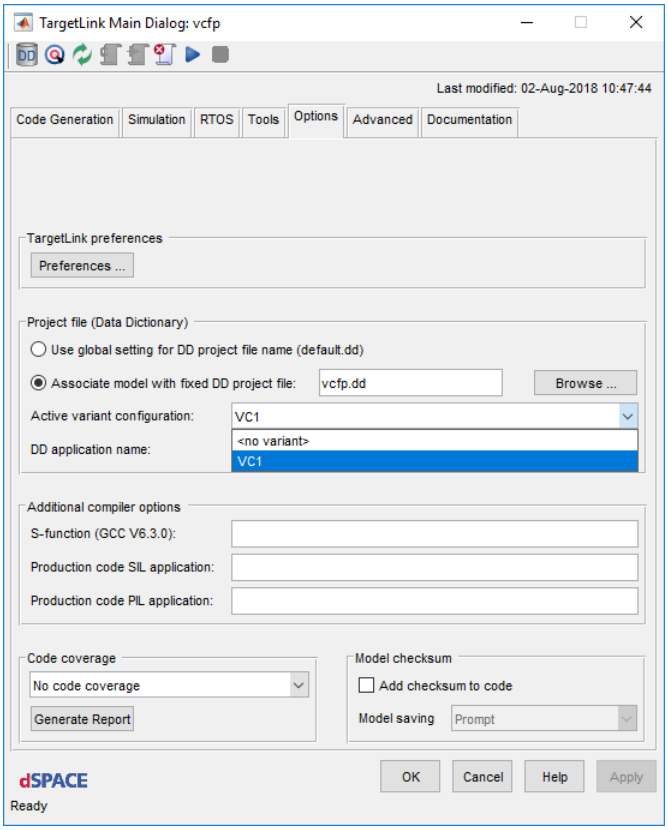
To activate data variants

1 In the Active Variant list of the Data Dictionary Manager, select the required variant configuration.



Note

Alternatively, you can enter the command `dsdd('SetActiveVariant',<VariantConfigName>)` in the MATLAB Command Window or you can select the active variant configuration on the Options page of the TargetLink Main Dialog (see the following screenshot):



Result

TargetLink uses the active variant configuration during production code generation. The active variant configuration is stored in the Config area of the DD, and is written to the generated header and code files as well.

Note

After you activated another variant configuration in the DD, the Active variant field in the TargetLink Main Dialog is updated as soon as you close and reopen the TargetLink Main Dialog. After you activated another variant configuration in the TargetLink Main Dialog, the Active variant field in the DD is updated as soon as click in the DD object tree (left pane) or perform a Refresh (via menu View/Refresh, or shortcut F5).

Related topics

HowTos

How to Activate Code Variants..... 576
How to Create Variant Configurations..... 570

Specifying Variable Vector Widths via Width Macros

Introduction

Specifying variable vector widths is a special use scenario for variables that have width variants.

Where to go from here

Information in this section

Introduction to Variable Vector Widths..... 586
Making Variables Width-Varying..... 594
Examples of Width-Varying Variables..... 607

Information in other sections

Specifying Code Variants..... 575

Introduction to Variable Vector Widths

Introduction

To introduce you to variable vector widths and make you familiar with basic implementation issues.

Where to go from here

Information in this section

Basics on Variable Vector Widths..... 586
Details on Variable Vector Width Specification..... 589
Details on Variable Vector Width Implementation..... 592

Basics on Variable Vector Widths

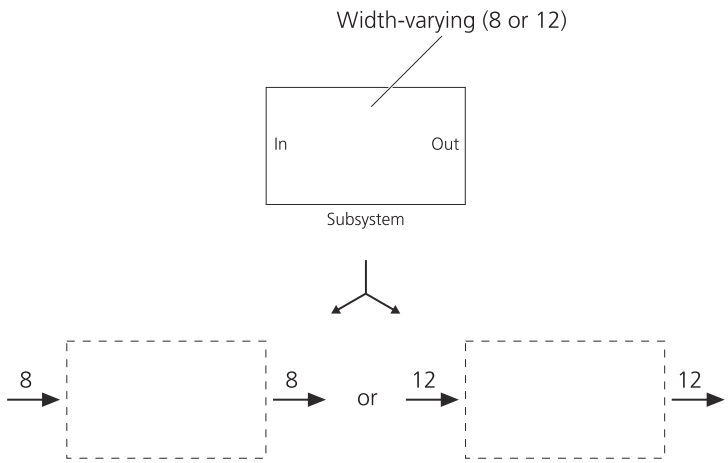
Variables fit multiple width contexts

TargetLink can generate code for vectorized variables so that they become width-varying at code compilation time. Consequently, the code of complete subsystems can become width-varying at code compilation time and can

therefore be used and reused in different widths contexts at code compilation time (see illustration below).

In consequence, the effort for testing code is minimized, because only one width-varying code instance must be tested instead of multiple width-specific code instances.

The following illustration shows the compatibility of width-varying code in different width contexts.



The width context (here: 8 or 12) is defined by the width variant that is active during code generation.

Note

Do not confuse width-*varying* code and width-*varianted* code. The latter is related to standard code variants, i.e., different width-specific code parts.

Loops as a precondition for width-varying variables

As a precondition that the code of a vectorized variable is width-varying at code compilation time, the code must be implemented as a loop, instead of a set of single statements. For example:

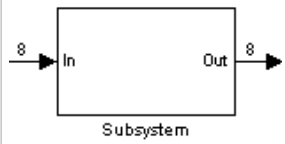
`for(i = 0; i < 8; i++)`
`{`
`  Var[i] = ... ;`
`}`
	instead of	`Var[0] = ... ;`
		`Var[1] = ... ;`
		`...`
		`Var[7] = ... ;`

The loop limit (here: i < 8) is derived from the width of the vectorized variable in the Simulink model at code generation time.

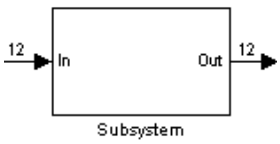
Drawback of fixed loop limits

When code is implemented as a loop for different vector widths (width variants) at different times, the vector width is defined by a number. This number depends on the width variant that was active during code generation.

The table below shows code generated for a subsystem used in different width contexts (8 and 12).




```
Int16 Out[8];
Int16 In[8];

for(i = 0; i < 8; i++)
{
   Out[i] = In[i] ... ;
}
```



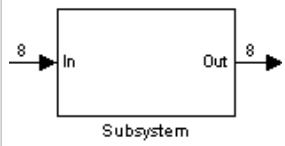

```
Int16 Out[12];
Int16 In[12];

for(i = 0; i < 12; i++)
{
   Out[i] = In[i] ... ;
}
```

The code shown above is suitable only for the active width variant, exclusively 8 or 12.

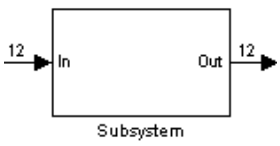
Advantage of loop limits defined by a width macro

When a vectorized variable is implemented as a loop for different vector widths (width variants) at different times and a *width macro* is used to define the vector width, the vector width can vary in the generated code.

The table below shows code generated for a width-varying subsystem used in different width contexts (8 and 12).




```
#define NumOfCyl 8
```




```
#define NumOfCyl 12
```

```
Int16 Out[NumOfCyl];
Int16 In[NumOfCyl];

for(i=0; i < NumOfCyl; i++)
{
   Out[i] = In[i] ... ;
}
```

The code is identical and can be used in different width contexts, i.e., the code is width-varying, except for the `#define` line.

Improved code readability

The readability of the generated code increases if width macros instead of numbers with no relevance are used as loop limits, which also applies for a single vector width specified. Width definitions become more expressive in the following, for example:

With Width Macro	Without Width Macro
`#define NumOfCyl 8`	

`Int16 Out[NumOfCyl];`
`Int16 In[NumOfCyl];`
 	
`for(i=0; i < NumOfCyl; i++)`
`{`
`  Out[i] = In[i] ... ;`
`}`
	`Int16 Out[8];`
	`Int16 In[8];`
	
	`for(i = 0; i < 8; i++)`
	`{`
	`  Out[i] = In[i] ... ;`
	`}`

Refer to Example of Improved Code Readability due to Width Macros on page 614.

Details on Variable Vector Width Specification

Required DD objects

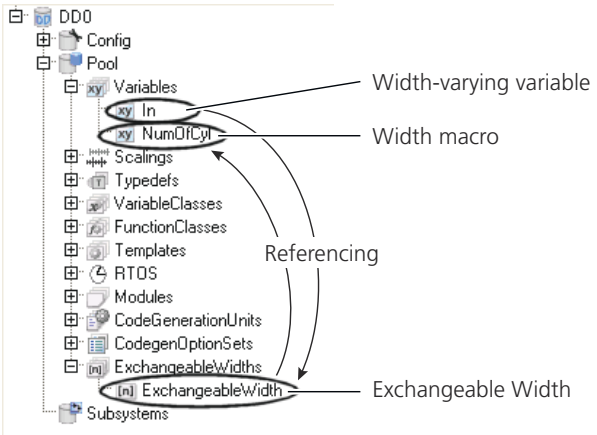
To apply a varying width to a vectorized variable at code compilation time, several Data Dictionary objects are required:

- A DD variable (here: **In**).

  This DD object references an **ExchangeableWidth** DD object. Its **Width** property is empty.
- An ExchangeableWidth object (here: **ExchangeableWidth**).

  This DD object lets you specify the variant-specific vector widths. It is used by TargetLink to propagate the variable vector width characteristic throughout a model. This DD object optionally references a width macro (DD variable).
- A width macro (DD variable, here: **NumOfCyl**).

  This DD object implements the width of the width-varying DD variable instances in the generated code. If no width macro exists, a default width macro is automatically created by TargetLink during code generation.

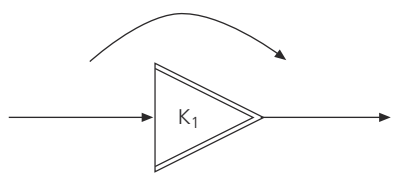


For detailed instructions, refer to Making Variables Width-Varying on page 594.

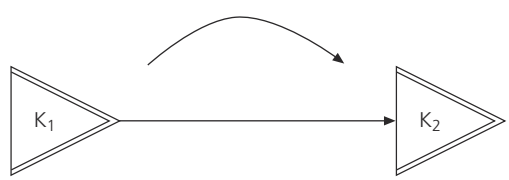
Inheritance of variable vector widths

If a variable is width-varying, other variables in the model can inherit this characteristic. This is beneficial if you want all the vectorized to be width-varying, for example in a subsystem, so that the complete subsystem can be used in different width contexts.

Propagation process First, if a variable is width-varying, the Code Generator acts as if the other vectorized variables that belong to the same block also reference the same ExchangeableWidth object (in-block propagation). A block can only be width-varying in its entirety. Hence, you can speak of width-varying blocks even though only one of the block's variables actually references the ExchangeableWidth object at modeling time.



Second, a width-varying block transmits its width variability to adjacent blocks following the signal flow during code generation if its Propagation property is set to On (inter-blocks propagation). For the inheriting blocks, again, in-block propagation takes place, and so on.



Propagation mechanism If a variable B inherits a variable vector width from another variable A, the following applies: A must reference an ExchangeableWidth object with its Propagation property set to **`Default`** or **`On`**. The Code Generator acts as if B references the ExchangeableWidth object by itself.

Note

B does not reference the ExchangeableWidth object at modeling time. The Code Generator only acts as if the reference existed.
For the **`Default`** setting, vector widths are propagated only if at least two widths are specified in the ExchangeableWidth object.

Propagation start points Blocks that reference an ExchangeableWidth object act as start points.

Propagation end points Propagation stops at blocks that do not support loop generation, refer to Blocks not supporting loop generation (🕮 TargetLink Limitation Reference).

Propagation stops:

- At blocks that reference an **ExchangeableWidth** object. This object has its **Propagation** property set to **off**.

  Tip

  Use these blocks if you want to terminate propagation on purpose. For detailed instructions, refer to How to Terminate Width Propagation on page 603.

- At blocks that reference other **ExchangeableWidth** objects. Hence, they act as the starting point.
- At blocks that reference a vectorized DD variable that does not reference an **ExchangeableWidth** object.
- At blocks for which the following applies:
  - They do not reference an **ExchangeableWidth** object.
  - They are fed with multiple signal lines that convey signals referencing different **ExchangeableWidth** objects.
- At **Mux**, **Demux**, and **Fcn** blocks.
- At the input variables of **Custom Code** blocks:
  - The input variables can become width-varying.
  - Whether the other block variables, e.g., output, become width-varying depends on the custom code.
- At **MinMax**, **Multiport Switch**, **Product**, and **Sum** blocks if they have only one vector input and one scalar output.
- At boundaries of certain systems or subsystems (AUTOSAR, external, incremental, model, reusable systems)

  Note

  For scaling-invariant systems, you must modify the relevant M scripts manually. For detailed instructions, refer to Implementing Scaling-Invariant Functions on page 320.

Blocks requiring specific settings For the following blocks, propagation is possible if specific settings are applied:

- **Assignment** block:

  The **1st dimension** parameter is set to **Index vector (dialog)** or **Starting index (dialog)**.

  The **2st dimension** parameter is set to **not used**.

  Propagation to the output port is possible via the **Y0** input port, the variable vector width is implemented in the `Init` code section. For the **U** input port, only intrablock propagation takes place, the variable vector width is implemented in the `Calc` code section.
- **Delay** block:

  The **Delay Length** parameter is ≤1. If set, the **Upper Limit** parameter is also ≤ 1

- **Selector** block (supported Simulink block):
  The **Index Option** is set to **Index vector (dialog)**.

Leaf bus elements

Leaf bus elements can be width-varying. Each width-varying element must be related to an **ExchangeableWidth** object. Inheritance of variable vector widths is possible for each leaf bus element separately.

Stateflow variables

Stateflow variables (SF Input, SF Output) can become width-varying. However, propagation of the width-variability is not supported within a Stateflow chart. The following conditions must therefore be met:

- Each vectorized variable of a Stateflow statement must reference the same **ExchangeableWidth** object. This is a workaround to achieve the same result as in-block propagation.
- All the statements in a Stateflow chart that you want to become width-varying must reference an **ExchangeableWidth** object. This is a workaround to achieve the same result as interblocks propagation.

Note

The same object means *one* object, not two or more objects with identical specifications.

For instructions on how to make Stateflow variables width-varying, refer to How to Make Stateflow Variables Width-Varying on page 601.

Referencing width macros in the model

Width macros can be referenced in block dialogs in the model as follows:

- As parameters
- As output of a **Constant** block
- As output of a **Probe** block
- As variables of a **Data Store Memory** block if the block does not relate to a **Data Store Write** block.

You must maintain that the width of a block parameter matches the width macro at code generation time. Refer to Example of a Width Macro Referenced in the Model on page 611.

Details on Variable Vector Width Implementation

Report on propagation failures

By setting a specific code generator option, TargetLink generates a report about the propagation of variable vector widths throughout the model. The report documents the blocks and ports at which this propagation terminates and the reasons.

For instructions, refer to How to Generate a Report on the Propagation Failures on page 606.

Loops as the fundamental code pattern

Basically, when TargetLink generates code for a width-varying variable, the code looks like this, for example:

```
for(i = 0; i < NumOfCyl; i++)
{ /* Gain: Model/Subsystem/Kp [0..NumOfCyl-1] */
   ... ;
}
```

The code of the vectorized variable is implemented as a loop with a zero-based index. The loop limit is defined by a *width macro* (**NumOfCyl**). Path information on width-varying blocks is described in a comment.

For more details on generating loops for vectorized variables and code examples, refer to Basics on Processing Vectors and Matrices (📖 TargetLink Preparation and Simulation Guide).

Limitation TargetLink does not support a width variant set to 1. This also applies to the default variant.

Preprocessor directives for initial values

You can specify variants of initial values in the Data Dictionary (Value property). The variants can have different vector widths (in the example below the widths are 2, 4, and 6).

Value	Float64	[1 2]	···	default
[variant]	Float64	[1 2 3 4]	···	V1
[variant]	Float64	[7 8 9 6 5 4]	···	V2

These code variants are implemented as preprocessor directives if the variable has width variants:

```
#if NumOfCyl == 2
    Int16 GainValue[NumOfCyl] = {1, 2};
#elif NumOfCyl == 4
    Int16 GainValue[NumOfCyl] = {1, 2, 3, 4};
#elif NumOfCyl == 6
    Int16 GainValue[NumOfCyl] = {7, 8, 9, 6, 5, 4};
#endif
```

Multiple variables can reference the same ExchangeableWidth object. Hence, they are related to the same width macro (**NumOfCyl**). The directives are then combined as much as possible to minimize code size:

```
#if NumOfCyl == 2
    Int16 GainValue1[NumOfCyl] = {1, 2};
    Int16 GainValue2[NumOfCyl] = {2, 2};
#elif NumOfCyl == 4
    Int16 GainValue1[NumOfCyl] = {1, 2, 3, 4};
    Int16 GainValue2[NumOfCyl] = {2, 2, 2, 2};
#elif NumOfCyl == 6
    Int16 GainValue1[NumOfCyl] = {7, 8, 9, 6, 5, 4};
    Int16 GainValue2[NumOfCyl] = {2, 2, 2, 2, 2, 2};
#endif
```

Faulty specification in the Data Dictionary Suppose a DD ExchangeableWidth object with the following **Width** variants (see illustration below) is referenced by a DD variable object with the following **Value** variants. TargetLink displays an error message during code generation (variant item **V3** is active), because variant item **V3** addresses a value variant but does not address a width variant.

Width	UInt32	2	···	default
[variant]	UInt32	4	···	V1
[variant]	UInt32	6	···	V2
···				
Value	Float64	[1 2]	···	default
[variant]	Float64	[1 2 3 4]	···	V1
[variant]	Float64	[7 8 9 6 5 4]	···	V2
[variant]	Float64	[7 8]	···	V3

Note

A variant item that addresses a value variant must also address a matching width variant.

Code Generator options

The following Code Generator options are relevant for the implementation of variable vector widths:

- ForceCodeGenOfMisspecifiedVariableWidthsBlocks
- LoopUnrollThreshold
- **ReportVariableVectorWidthDetails** (📖 TargetLink Model Element Reference)

Related topics

References

ForceCodeGenOfMisspecifiedVariableWidthsBlocks (📖 TargetLink Model Element Reference)
LoopUnrollThreshold (📖 TargetLink Model Element Reference)
ReportVariableVectorWidthDetails (📖 TargetLink Model Element Reference)

Making Variables Width-Varying

Introduction

To make a variable and the code of the block that references this variable width-varying. In addition, terminating width propagation and defining width macros in custom modules is addressed.

Where to go from here

Information in this section

How to Define Width Macros..... 596
How to Define the Width Values of Width-Varying Variables..... 598
How to Make Variables Width-Varying..... 600
How to Make Stateflow Variables Width-Varying..... 601
How to Terminate Width Propagation..... 603
How to Define Width Macros in Custom Header Files..... 604
How to Test Width-Varying Code..... 605
How to Generate a Report on the Propagation Failures..... 606

How to Define Width Macros

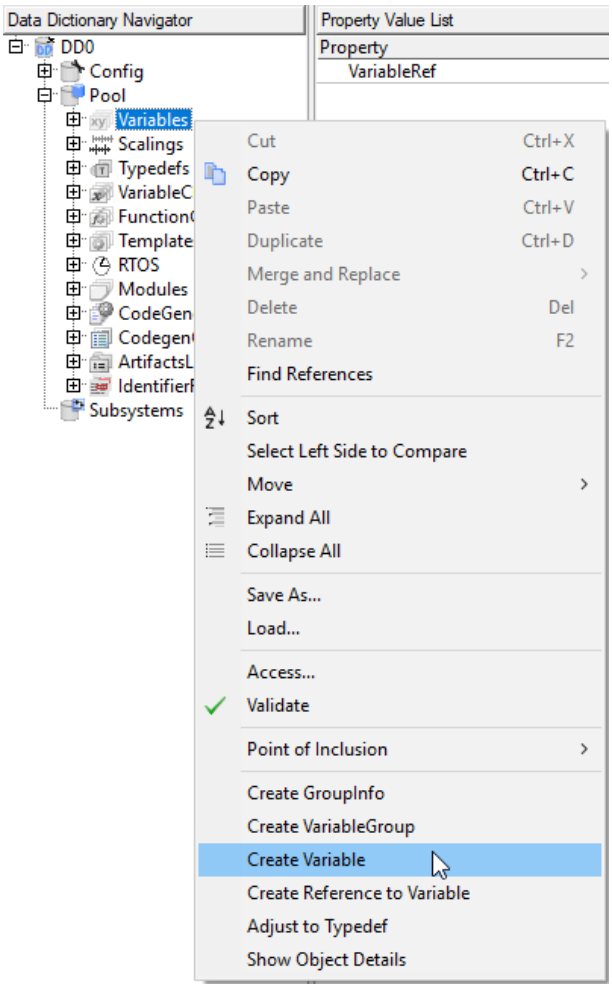
Objective

To use your own width macro instead of the TargetLink default width macro.

Method

Defining width macros

1 In the Pool area of the Data Dictionary, create a variable object in the Variables folder.



2 Select the new variable object.

3 Set its properties as follows:

Property	Possible Value
Class	EXCHANGEABLE_WIDTH_MACRO **Note** This variable class is available if you use a predefined system template, e.g., dsdd_master_basic.
Type	Int32
Scaling	VOID_SCALING
Width	–
Value	–
Min	Must be less than or equal to the smallest width variant value of the ExchangeableWidth object.
Max	Must be greater than or equal to the greatest width variant value of the ExchangeableWidth object.
NameTemplate	$D, $M, $N, $(Group), $T, $(BaseType), or $(Type)

Result

You defined a width macro. If this macro is referenced by an ExchangeableWidth object, TargetLink uses it to declare the variable (width) and for the loop limit in the generated code, for example (`NumOfCyl` is the width macro.):

```
Int16 Out[NumOfCyl];
Int16 In[NumOfCyl];
...
for(i = 0; i < NumOfCyl; i++)
{
   ...
   Out[i] = In[i] ... ;
   ...
}
```

Related topics

HowTos

How to Create New Data Dictionary Objects (TargetLink Data Dictionary Basic Concepts Guide)
How to Set Properties of Data Dictionary Objects (TargetLink Data Dictionary Basic Concepts Guide)

How to Define the Width Values of Width-Varying Variables

Objective

To define all the different widths, a vectorized variable must be able to take these at code compilation time.

> **Note**
>
> Do not confuse width-varying and width-varianted. The latter is related to standard code variants.

Preconditions

Code variant items must exist in the Data Dictionary for each width variant. For defining variant items, refer to How to Create Variant Items on page 568.

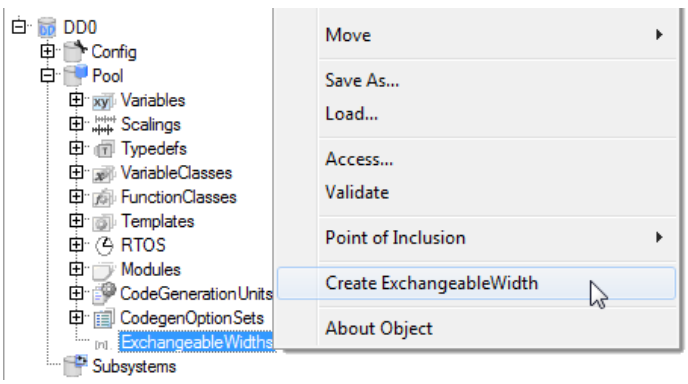
Restrictions

TargetLink does not support a width variant set to 1. This also applies to the default variant.

Method

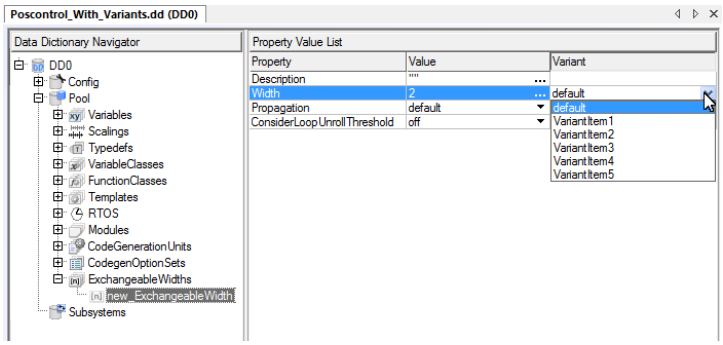
To define the width value of a width-varying variable

1 Via the Pool object context menu, create an ExchangeableWidths object and then an ExchangeableWidth object.



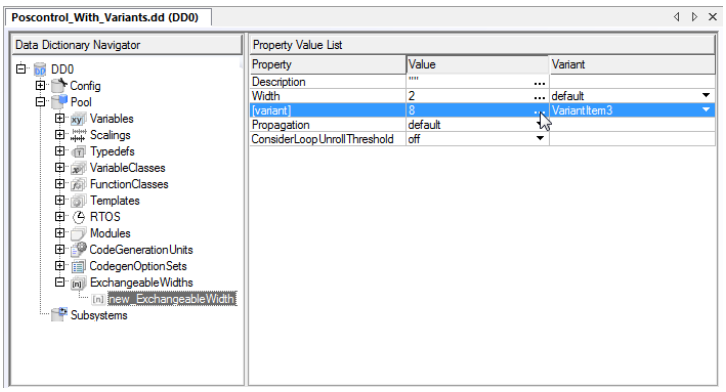
A new_ExchangeableWidth object is created.

2 For the **Width** property of the **new_ExchangeableWidth** object, select a variant item from the **Variant** column.



A variant property is inserted below the **Width** row.

3 Specify the variant's width.



4 If you want to create a second or more variants, you must always click the arrow in the **default** column, not in a variant column. Otherwise, the existing variant item is replaced by a new one.

5 Reference a width macro in the **MacroRef** property field.

Tip

If no width macro exists, a default width macro is automatically created.

6 Change the other property values if needed.

Result

You have defined the different widths that the variable can take at code compilation time.

Related topics

HowTos

How to Specify Code Variants..... 575

How to Make Variables Width-Varying

Objective

To make a vectorized variable usable in different width contexts at code compilation time.

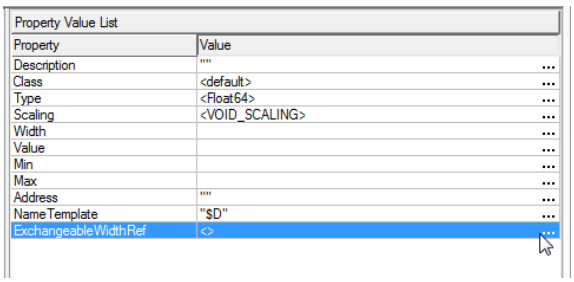
Preconditions

- The variable must exist in the Data Dictionary. For creating DD variable objects, refer to How to Create New Data Dictionary Objects (📖 TargetLink Data Dictionary Basic Concepts Guide).
- An appropriate ExchangeableWidth object must exist in the Data Dictionary. For defining such objects, refer to How to Define the Width Values of Width-Varying Variables on page 598.

Method

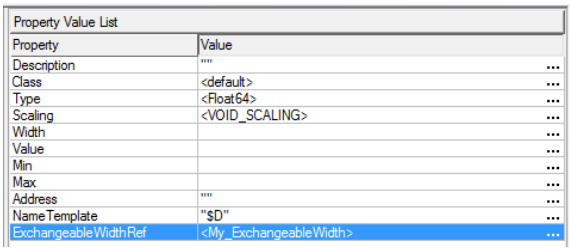
To make variables width-varying

1. In the DD Navigator, select the variable object you want to make width-varying.
2. Click the Browse button in the **Value** column of the **ExchangeableWidthRef** property in the Data Dictionary.



The **Select object** dialog opens and displays all the available **ExchangeableWidth** objects.

3. Select an **ExchangeableWidth** object and click **Ok**.



Result

You have made the variable width-varying. Its width is defined by a width macro (default or user-defined) so that the width can take several values at code compilation time, for example (variable: **`Out`**, width macro: **`NumOfCyl`**):

Note

The width values are specified in the **ExchangeableWidth** object that references the width macro. The variable's **Width** property must not be set.

```
Int16 Out[NumOfCyl];
for(i = 0; i < NumOfCyl; i++)
{
   Out[i] = ... ;
}
```

Related topics

HowTos

How to Create New Data Dictionary Objects (TargetLink Data Dictionary Basic Concepts Guide)
How to Define the Width Values of Width-Varying Variables..... 598

How to Make Stateflow Variables Width-Varying

Objective

To enable a Stateflow variable so its width can take different values at code compilation time.

Preconditions

- The Stateflow variable must exist in the Data Dictionary (only method 1).
- An ExchangeableWidth object must exist in the Data Dictionary. For defining these objects, refer to How to Define the Width Values of Width-Varying Variables on page 598.

Possible methods

You can make a Stateflow variable width-varianted, independent of whether it exists in the Data Dictionary (DD variables).

- If a DD variable exists (recommended), refer to Method 1.
- If a DD variable does not exist, refer to Method 2.

Method 1

To make a Stateflow variable width-varianted with DD variable

1. At the variable object that you want to make width-varying, select the **ExchangeableWidthRef** property and reference the **ExchangeableWidth** object.

   For detail, refer to How to Make Variables Width-Varying on page 600.

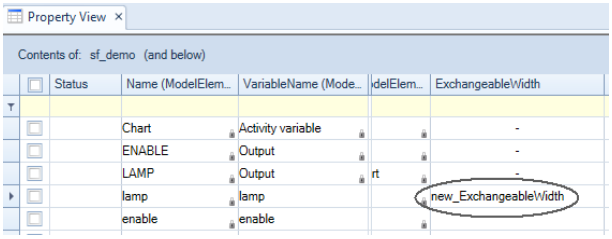
Method 2

To make a Stateflow variable width-varianted without DD variable

1 Open the TargetLink Property Manager.

2 If an ExchangeableWidth property column does not exist, add it via the Column Chooser. For more information, refer to Column Chooser Dialog (📖 TargetLink Tool and Utility Reference).

3 For the specific model element (here: lamp), reference an ExchangeableWidth DD object, for example:



Result

You have made the Stateflow variable width-varying. Its width is defined by a width macro (default or user-defined) so that the width can take several values at code compilation time, for example (variable: **`Out`**, width macro: **`NumOfCyl`**):

```
Int16 Out[NumOfCyl];
for(i = 0; i < NumOfCyl; i++)
{
   Out[i] = ... ;
}
```

Related topics

Basics

Modifying Multiple Properties at Once via the Property Manager (📖 TargetLink Preparation and Simulation Guide)

HowTos

How to Define the Width Values of Width-Varying Variables..... 598

References

Column Chooser Dialog (📖 TargetLink Tool and Utility Reference)

How to Terminate Width Propagation

Objective

To terminate width propagation on purpose at a certain block to avoid propagation from a subsystem onto the surrounding model, for example.

> **Tip**
>
> Propagation stops at blocks that do not support loop generation. For an overview of these blocks (*propagation end points*), refer to Details on Variable Vector Width Implementation on page 592.

Precondition

The block at which the propagation must stop, references a DD variable for its block output.

Method

To terminate width propagation

1. Create an ExchangeableWidth object specified as follows:
   - The **Propagation** property is set to `off`.
   - The other property settings are identical to the settings of the **ExchangeableWidth** object that causes the width propagation, including the reference to the same width macro.
2. Let the DD output variable of the block reference the **ExchangeableWidth** object whose **Propagation** property is set to `off`.

Result

Width propagation stops at this block. The block itself is width-varying.

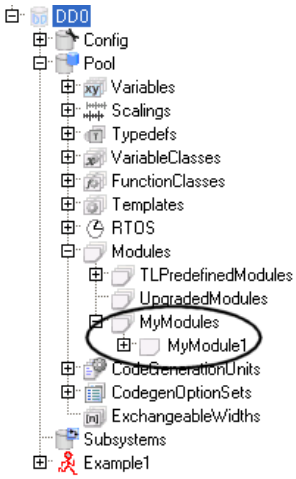
How to Define Width Macros in Custom Header Files

Objective

To have the width macro defined in a module other than the default (`tl_defines.h`).

Precondition

The module must exist in the Data Dictionary. For example:



Method

To define a width macro in a custom header file

1. Right-click the EXCHANGEABLE_WIDTH_MACRO variable class in the Data Dictionary.

   The context menu opens.
2. Select Duplicate.

   The Copy_of_EXCHANGEABLE_WIDTH_MACRO variable class is created.
3. Rename the new variable class.
4. Change the ModuleRef property of the new variable class:

   Click the Browse button (...) in the Value column.

   The Select object dialog opens and displays all the available modules.
5. Select a module and click OK.

Result

The width macro is defined in the selected module after production code has been generated.

How to Test Width-Varying Code

Objective

To check the validity of the code generated for width-varying variables.

Method

To test width-varying code

1. Generate code for each width variant. The result of each code generation run comprises a variant-specific code part (width context) and a generic (reusable) code part. For example,

Width context	`...`
	`#define NumOfCyl 8`
	`...`
		`...`
		`#define NumOfCyl 12`
		`...`

Reusable code part	`Int16 Out[NumOfCyl];`
	`Int16 In[NumOfCyl];`
	
	`for(i = 0; i < NumOfCyl; i++)`
	`{`
	`  Out[i] = In[i] ... ;`
	`}`
		`Int16 Out[NumOfCyl];`
		`Int16 In[NumOfCyl];`
		
		`for(i = 0; i < NumOfCyl; i++)`
		`{`
		`  Out[i] = In[i] ... ;`
		`}`

2. Compare the width-contexts. They differ only in the macro definitions.
3. Compare the reusable code parts. They must not differ.
4. Check the result of the width propagation by inspecting the report on the propagation failures, refer to How to Generate a Report on the Propagation Failures on page 606.

Note

For blocks of width-varying libraries, width propagation must not reach into the surrounding model part.

Result

You have checked the validity of the code.

Related topics

HowTos

How to Generate a Report on the Propagation Failures..... 606

How to Generate a Report on the Propagation Failures

Objective

To generate a report on the terminated propagation of ExchangeableWidth objects and the reasons.

Prerequisite

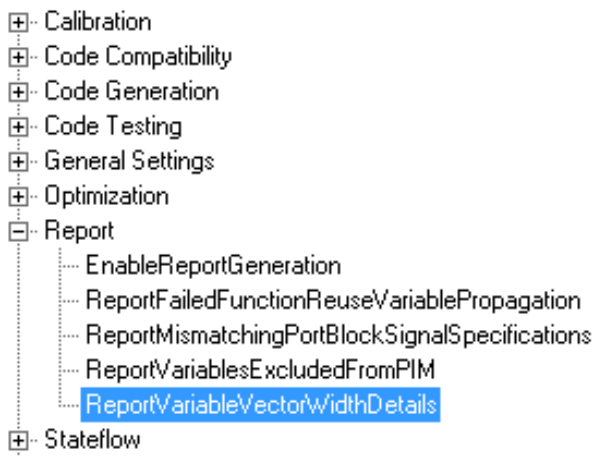
The EnableReportGeneration code generator option is selected.

Method

To generate a report on the propagation failures

1 Open the Code Generator Options Dialog.

2 Browse the marked code generator option:



3 Select the Generate a report for failed ExchangeableWidth object propagation checkbox.

Result

When code is generated, a report on variable vector width is generated. This report describes the blocks and ports at which the propagation of ExchangeableWidth objects terminates and the reasons.

Related topics

Basics

Basics on Optimizing the Production Code 498

References

EnableReportGeneration (📖 TargetLink Model Element Reference)
ReportVariableVectorWidthDetails (📖 TargetLink Model Element Reference)

Examples of Width-Varying Variables

Introduction

Simple models providing variable vector widths will show you the principle of modeling variable vector widths as well as the generated production code. These examples help you as a starting point for developing your own width-varying blocks/subsystems and libraries. In addition, an example shows you that using width macros improves code readability.

Where to go from here

Information in this section

Example of Width-Varying Subsystems..... 607
Example of a Width-Varying Library..... 610
Example of a Width Macro Referenced in the Model..... 611
Example of Improved Code Readability due to Width Macros..... 614

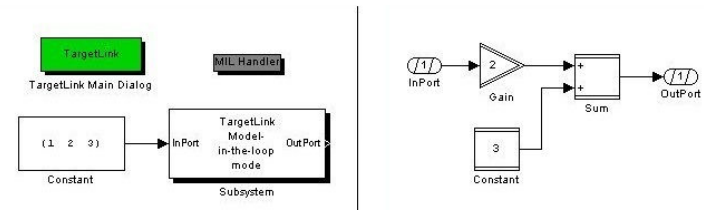
Example of Width-Varying Subsystems

Introduction

Using a subsystem in different width contexts.

Subsystem allows uniform elements

Suppose a model is designed as the following:



The Constant parameter of the Simulink Constant block is a vector that consists of three elements (as shown) and another time consists of six elements. The TargetLink **Subsystem** subsystem is width-varying.

TargetLink generates the following production code for the TargetLink subsystem when the active width (variant) is 3:

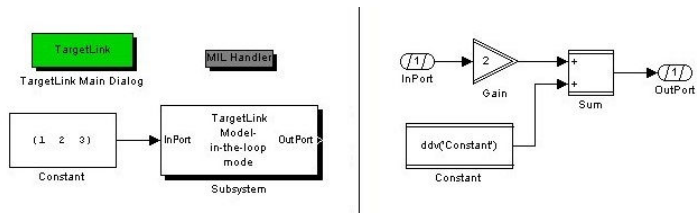
```
#define WIDTH 3
Int16 Out[WIDTH];
Int16 In[WIDTH];
for(i = 0; i < WIDTH ; i++)
{
   Out[i] = (In[i] * 2) + 3;
}
```

Specifications The following specifications are necessary:

Where?	Specification
Block dialog	The InPort block in the TargetLink subsystem references a DD variable for its block output, e.g., In.
Data Dictionary	In references an ExchangeableWidth object tha tdefines the width variants (3 and 6). The variant items that are related to the width variants must belong to an CodeVariantItems object in the variant configuration (see the `/Config/VariantConfigs` area in the DD).
Data Dictionary	The ExchangeableWidth object references a DD variable (width macro), e.g., WIDTH, and its Propagation property is set to **`default`** , or **`On`**.
Block dialog	For the other blocks in the subsystem, the Uniform elements checkbox on the Output page is selected. They do not need to reference a DD variable.

Subsystem has vectorized parameters

Suppose a model is designed as the following:



The Constant parameter of the Simulink Constant block is a vector that of three elements (as shown), and another time consists of six elements. The TargetLink Subsystem subsystem is width-varying. The TargetLink Constant block references the Constant DD variable (`ddv('Constant')`) whose Value property is varianted to `[1 2 3]` and `[1 2 3 4 5 6]`.

TargetLink generates the following production code for the TargetLink subsystem when the active width (variant) is 3:

```
#define WIDTH 3
Int16 Out[WIDTH];
Int16 In[WIDTH];
#IF WIDTH == 3
Int16 C[WIDTH] = {1 2 3};
#ELIF WIDTH == 6
Int16 C[WIDTH] = {1 2 3 4 5 6};
#ENDIF
for(i = 0; i < WIDTH; i++)
{
   Out[i] = (In[i] * 2) + C[i];
}
```

Specifications The following specifications are necessary:

Where?	Specification
Block dialog	The InPort block in the TargetLink subsystem references a DD variable for its block output, e.g., In.
Block dialog	The Constant block in the TargetLink subsystem references a DD variable for its block output, e.g., Constant. The constant values are taken from the DD by using the `ddv('Constant')` command. The Value property of this DD variable has the following two code variants: [1 2 3] and [1 2 3 4 5 6].
Data Dictionary	The In variable references an ExchangeableWidth object that defines the width variants (3 and 6). The variant items that are related to the width variants must belong to an CodeVariantItems object in the variant configuration (see the `/Config/VariantConfigs` area in the DD).
Data Dictionary	The Constant variable references an ExchangeableWidth object which defines the width variants (3 and 6).
Data Dictionary	The ExchangeableWidth object references a DD variable (width macro), e.g., WIDTH, and its Propagation property is set to `default` , or `On`.
Block dialog	For the other blocks in the subsystem, the Uniform elements checkbox on the Output page is selected. They do not need to reference a DD variable.

Related topics

References

ddv (TargetLink API Reference)

Example of a Width-Varying Library

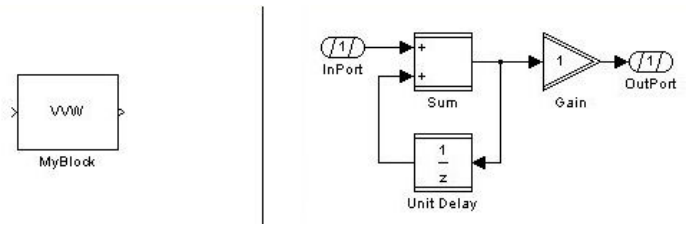
Introduction

Implementing a width-varying library allows for reusing its blocks in multiple width contexts. This implementation approach minimizes testing efforts.

> **Note**
>
> Use these libraries only in width contexts that the library has been specified and tested for.

Masked subsystem

The blocks of a width-varying library are usually masked subsystems, for example:



The MyBlock block becomes width-varying when its underlying subsystem is width-varying (refer to Example of Width-Varying Subsystems on page 607). In addition to a standard subsystem, width propagation must not exceed the subsystem's boundary. For the example above, width propagation must stop at the OutPort block (refer to How to Terminate Width Propagation on page 603), which requires an additional ExchangeableWidth object.

Specifications The following specifications are necessary:

> **Note**
>
> The library and the model must reference the same Data Dictionary project file.

Where?	Specification
Block dialog	The InPort block in the TargetLink subsystem references a DD variable for its block output, e.g., In.
Data Dictionary	In references an ExchangeableWidth1 object that defines the width variants (3 and 6). The variant items that are related to the width variants must belong to an CodeVariantItems object in the variant configuration (see the `/Config/VariantConfigs` area in the DD).
Data Dictionary	The ExchangeableWidth1 object references a DD variable (width macro), e.g., Width1, and its Propagation property is set to `default` , or `On`.
Block dialog	The OutPort block in the TargetLink subsystem references a DD variable for its block output, e.g., Out.

Where?	Specification
Data Dictionary	Out references an ExchangeableWidth2 object that defines the width variants (3 and 6).
Data Dictionary	The ExchangeableWidth2 object references the same width macro (Width1), but its Propagation property is set to Off.
Block dialog	For the other blocks in the subsystem, the Uniform elements checkbox on the Output page is selected. They do not need to reference a DD variable.

Note

Check whether the code generated for a width-varying block is identical for all its width variants.

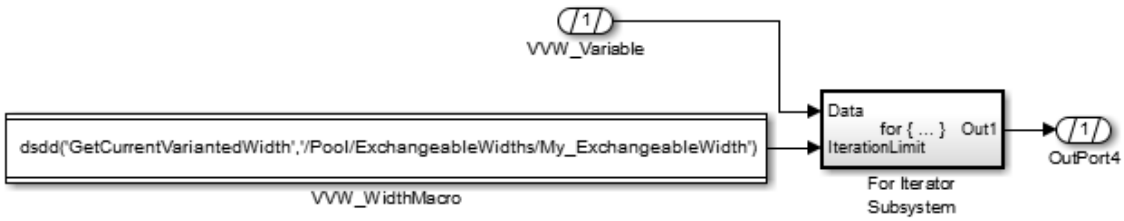
Example of a Width Macro Referenced in the Model

Introduction

Making the active width value of a width-varying DD variable explicitly available in the model.

Example

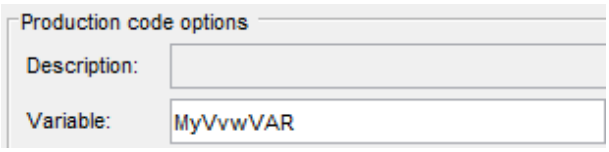
Suppose a subsystem in a TargetLink model is designed as the following:



The VVW_Variable inport block references a width-varying DD variable whose active width is used as the constant parameter in the VVW_WidthMacro block. The example's relevant model parts and DD objects as well as their specifications are described in the following paragraphs.

VVW_Variable block

For the above mentioned example, the VVW_Variable inport block references the width-varying MyVvwVAR DD variable.



MyVvwVAR is specified in the DD as follows:

DD object path: `/Pool/Variables/MyVvwVAR`

Property Value List		
Property	Value	Variant
Description	"" ...	default
Class	<default> ...	default
Type	<Int32> ...	default
Scaling	<VOID_SCALING> ...	default
Width	...	default
Value	[1 1 1 1] ...	default
[variant]	[-1 -1 -1 -1 -1] ...	VariantItem1
Min	...	default
Max	...	default
Address	"" ...	default
NameTemplate	"$S_$B" ...	default
ExchangeableWidthRef	<My_ExchangeableWidth> ...	

The My_ExchangeableWidth DD object defining the possible widths is referenced, because this is required for width-varying variables, refer to How to Define the Width Values of Width-Varying Variables on page 598.

Note

The Value property only relates to initial values, it does not define the possible widths.

My_ExchangeableWidth DD object

For the above mentioned example, the My_ExchangeableWidth is specified in the DD as follows:

DD object path: `/Pool/ExchangeableWidths/My_ExchangeableWidth`

Property Value List		
Property	Value	Variant
Description	"" ...	
Width	4 ...	default
[variant]	5 ...	VariantItem1
Propagation	on	
ConsiderLoopUnrollThresho	off	
MacroRef	<VvwMacros/MyVvwMACRO> ...	

This DD object references the MyVvwMACRO DD variable (width macro).

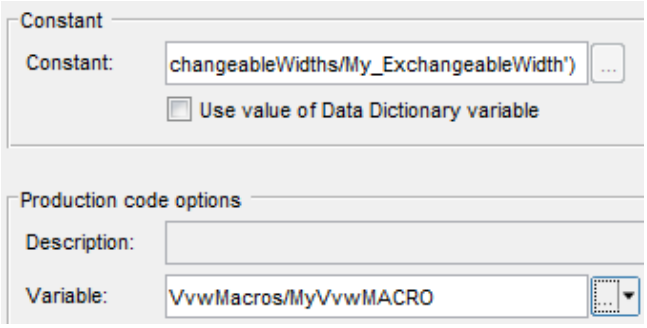
Note

The width values are specified in the ExchangeableWidth object that references the width macro.

VVW_WidthMacro block

For the above mentioned example, the VVW_WidthMacro block is a Constant block type. Its output equals the width value of the width-varying MyVvwVAR DD variable that is active at code generation time. This output is used as external input for the IterationLimit port of the For Iterator Subsystem block.

The VVW_WidthMacro block references the MyVvwMACRO DD variable, which is a width macro. For the Constant value, the VVW_WidthMacro block references the My_ExchangeableWidth DD object.



The Constant value is referenced by using a `dsdd` command:

```
dsdd('GetCurrentVariantedWidth','/Pool/ExchangeableWidths/My_Exchange
ableWidth')
```

This command guarantees that the actual width values of block parameter and width macro match at code generation time.

Tip

You can also use, a global MATLAB variable to specify the Constant value, for example.

MyVvwMACRO is specified in the DD as follows:

DD object path: `/Pool/Variables/VvwMacros/MyVvwMACRO`

Property Value List

Property	Value	Variant
Description	""	default
Class	<EXCHANGEABLE_WIDTH_MACRO>	default
Type	<Int32>	default
Scaling	<VOID_SCALING>	default
Width		default
Value		default
Min		default
Max		default
Address	""	default
NameTemplate	"$D"	default

For defining width macros, refer to How to Define Width Macros on page 596.

Related topics

Basics

Details on Variable Vector Width Implementation..... 592

HowTos

How to Define the Width Values of Width-Varying Variables..... 598
How to Define Width Macros..... 596

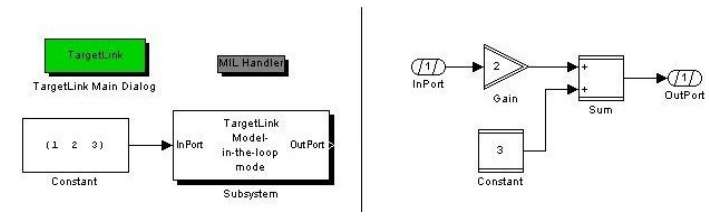
Example of Improved Code Readability due to Width Macros

Introduction

Using width macros instead of numbers for specifying vector widths increases readability of the generated code. Although only one vector width is specified, i.e., not multiple, using a width macro can be beneficial.

Example

Suppose a model is designed as the following:



The Constant parameter of the Simulink Constant block is a vector consisting of three elements, i.e., one vector width. The vector width is represented by a width macro in the generated code.

TargetLink generates the following production code for the TargetLink subsystem:

LoopUnrollThreshold == 5

With Width Macro	Without Width Macro

`#define WIDTH 3`
`Int16 Out[WIDTH];`
`Int16 In[WIDTH];`
`Out[0] = (In[0] * 2) + 3;`
`Out[1] = (In[1] * 2) + 3;`
`Out[2] = (In[2] * 2) + 3;`
	`Int16 Out[3];`
	`Int16 In[3];`
	`Out[0] = (In[0] * 2) + 3;`
	`Out[1] = (In[1] * 2) + 3;`
	`Out[2] = (In[2] * 2) + 3;`

LoopUnrollThreshold == 2	
With Width Macro	**Without Width Macro**

`#define WIDTH 3`
`Int16 Out[WIDTH];`
`Int16 In[WIDTH];`
`for(i = 0; i < WIDTH; i++)`
`{`
`  Out[i] = (In[i] * 2) + 3;`
`}`
	`Int16 Out[3];`
	`Int16 In[3];`
	`for(i = 0; i < 3; i++)`
	`{`
	`  Out[i] = (In[i] * 2) + 3;`
	`}`

Specifications The following specifications are necessary:

Where?	Specification
Block dialog	The InPort block in the TargetLink subsystem references a DD variable for its block output, e.g., In.
Data Dictionary	In references an ExchangeableWidth object that defines the one width variant.
Data Dictionary	The ExchangeableWidth object references a DD variable (width macro), e.g., WIDTH. Its ConsiderLoopUnrollThreshold property is set to **On**.
Block dialog	For the other blocks in the subsystem, the Uniform elements checkbox on the Output page is selected. They do not reference any DD variables.

Related topics

Basics

Basics on Optimizing the Production Code 498
Basics on Processing Vectors and Matrices (TargetLink Preparation and Simulation Guide)

References

LoopUnrollThreshold (TargetLink Model Element Reference)

Specifying Conditional Control Flows

Introduction

TargetLink lets you use preprocessor directives to create optional control flows to be considered during code compilation. This makes it easier to develop programs and compile them in different execution environments.

Where to go from here

Information in this section

Basics on Conditional Code Compilation via Preprocessor Directives..... 616

Details on Handling Objects in Preprocessor Directives..... 619

Details on Handling Logical Instructions in Preprocessor Directives..... 620

How to Specify Conditional Control Flows via Enabled Subsystems..... 621

How to Specify Conditional Control Flows via #If-#Else Statements..... 623

How to Include External Macros for Generating Conditional Control Flows..... 625

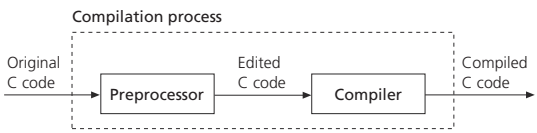
Information in other sections

Modeling Control Flows (📖 TargetLink Preparation and Simulation Guide)

Basics on Conditional Code Compilation via Preprocessor Directives

Conditional code compilation

A compiler usually does not process original C code but the code output of the preceding preprocessor. A preprocessor performs specific actions and deletes or substitutes specific C code sections according to conditions you defined.



Preprocessor directives

Preprocessor directives define how the preprocessor edits C code before the file containing the C code is compiled. They enable you to design conditional control flows in a TargetLink model.

Preprocessor directives always begin with the # sign in the original C code. For example:

```
...
#define PI   3.1416
main()
   int radius=1;
   float circ;
   circ = PI * (r*r);
...
```

The preprocessor edits the code as follows:

```
...
main()
   int radius=1;
   float circ;
   circ = 3.1416 * (r*r);
...
```

Note

A preprocessor treats code like text and performs textual substitutions.

Implementation You can implement preprocessor directives for conditional code compilation in a TargetLink model as follows:

- Via Simulink enabled subsystems
- Via #If-#Else statements with the following blocks:
  - TargetLink Preprocessor IF block
  - Simulink If block in combination with If Action subsystems
- Via Stateflow charts

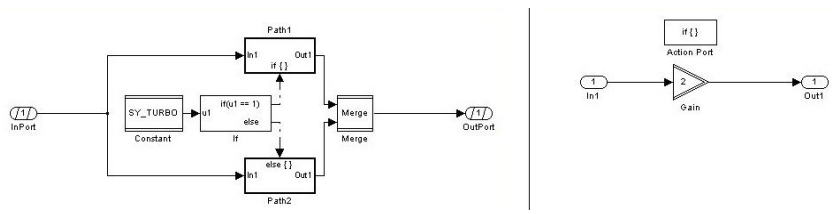
Note

A TargetLink Preprocessor IF block always results in preprocessor directives. A Simulink If block, however, results in preprocessor directives if specific preconditions are fulfilled, and otherwise generates common If code.

Implementation examples

There are various implementation methods that result in the same preprocessor directives.

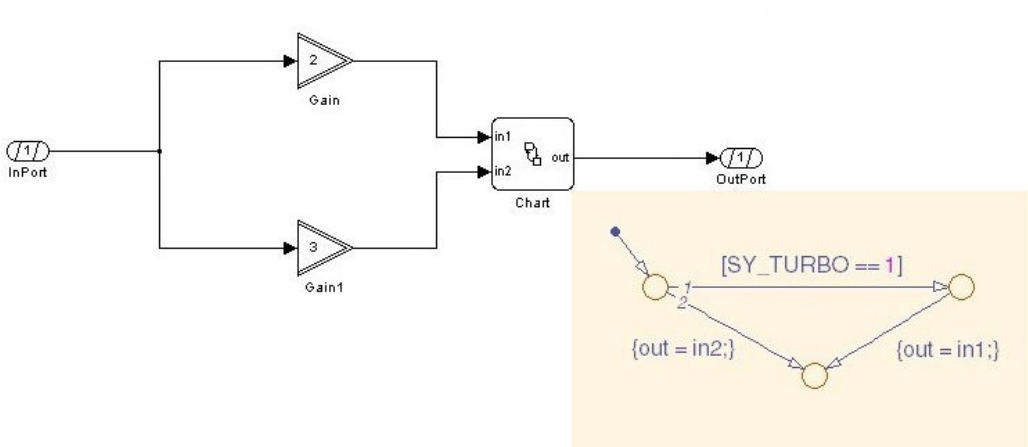
Simulink If block You can combine a Simulink If block with If Action subsystems. The following illustration shows the content of the Path1 If Action subsystem:



TargetLink generates the following production code:

```
#if SY_TURBO == 1
   CAL Int16 Ex1_Gain1 = 2;
#endif
#if !(SY_TURBO == 1)
   CAL Int16 Ex1_Gain2 = 3;
#endif
...
void example1_sl(void)
{
#if SY_TURBO == 1
   /* update of variable(s) associated with example1_sl/Path1/Gain
   # combined # TargetLink outport: example1_sl/OutPort
   # combined # Gain: example1_sl/Pfad1/Gain */
   Se1sl1_OutPort = Se1sl1_InPort * Ex1_Gain1;
#else
   /* update of variable(s) associated with example1_sl/Path2/Gain
   # combined # TargetLink outport: example1_sl/OutPort
   # combined # Gain: example1_sl/Pfad2/Gain */
   Se1sl1_OutPort = Se1sl1_InPort * Ex1_Gain2;
#endif
}
```

Stateflow chart You can use a Stateflow chart:



Note

With default settings, Stateflow generates If queries, not preprocessor #If statements. For information on generating preprocessor statements, refer to Basics on Substituting IF Queries with Preprocessor IF Statements (📖 TargetLink Preparation and Simulation Guide).

TargetLink generates the following production code:

```
#if SY_TURBO == 1
   CAL Int16 Ex1_Gain1 = 2;
#endif
#if !(SY_TURBO == 1)
   CAL Int16 Ex1_Gain2 = 3;
#endif
...
void example1_sf(void)
{
/* Begin execution of chart example1_sf/Chart */
#if SY_TURBO == 1
   /* # combined # TargetLink outport: example1_sf/OutPort
   # combined # Gain: example1_sf/Gain */
   Se1sf1_OutPort = Se1sf1_InPort * Ex1_Gain1;
#else
   /* # combined # TargetLink outport: example1_sf/OutPort
   # combined # Gain: example1_sf/Gain1 */
   Se1sf1_OutPort = Se1sf1_InPort * Ex1_Gain2;
#endif
/* End execution of chart example1_sf/Chart */
}
```

Related topics

Basics

Basics on Variant Handling in the TargetLink Data Dictionary..... 564

HowTos

How to Specify Conditional Control Flows via #If-#Else Statements..... 623
How to Specify Conditional Control Flows via Enabled Subsystems..... 621

Details on Handling Objects in Preprocessor Directives

Object handling

TargetLink handles objects (variables, functions) enclosed in preprocessor directives in such a way that memory consumption is minimized.

Example

Suppose an enabled subsystem is used for conditional compilation.

```
#ifdef SWITCH
   ss_enable( );
#endif /* SWITCH */
```

ss_enable( ) uses a global variable:

```
GLOBAL MyGlobVar;
```

```
void ss_enable( )
{
   ...
   x = MyGlbVar + y; /* Usage of MyGlbVar */
   ...
}
```

TargetLink shifts the definitions of ss_enable and MyGlbvar into the preprocessor directive, which reduces memory consumption:

```
#ifdef SWITCH
GLOBAL MyGlobVar;
#endif /* SWITCH */
#ifdef SWITCH
void ss_enable( )
{
   ...
   x = MyGlbVar + y; /* Usage of MyGlbVar */
   ...
}
#endif /* SWITCH */
```

Related topics

Basics

Details on Handling Logical Instructions in Preprocessor Directives..... 620

Details on Handling Logical Instructions in Preprocessor Directives

Handling auxiliary macros

TargetLink uses auxiliary macros for implementing logical instructions (AND, OR), which guarantees that all types of preprocessors understand the resulting production code.

All auxiliary macros are generated into the `tl_aux_defines.h file`, which keeps the C files lean. Using templates in the TargetLink Data Dictionary, you can adapt `tl_aux_defines.h` (`/Pool/Templates/Modules`) and the auxiliary macros themselves (`/Pool/Templates/Variables`) to your needs.

Example

Suppose, a model fragment represents the following logical OR instruction:

```
#if SWITCH_1 || SWITCH_2
   ss_enable();
#endif /* SWITCH_1 || SWITCH_2 */
```

As not all preprocessors understand such a parenthesis notation, TargetLink generates the following code:

```
#if SWITCH_1
   #define AUX
#endif
#if SWITCH_2
   #define AUX
#endif
#ifdef AUX
   ss_enable();
#endif /* AUX */
```

Example

Suppose, a model fragment represents the following logical AND instruction:

```
#if SWITCH_1 & SWITCH_2
void ss_B()
{
   ...
}
#endif
```

As not all preprocessors understand such a parenthesis notation, TargetLink generates the following code:

```
#if SWITCH_1
   #if SWITCH_2
      #define AUX
   #endif
#endif
#ifdef AUX
void ss_B()
{
   ...
}
#endif
```

How to Specify Conditional Control Flows via Enabled Subsystems

Objective

You must specify conditional control flows via an enabled subsystem to control whether a specific code section is included during compilation.

Method

To specify conditional control flows via enabled subsystems

1. From the Simulink block library, insert an Enabled Subsystem block in your TargetLink subsystem.
2. Design the control algorithm that represents the relevant code section inside the Enabled Subsystem.

3 Specify the Enable and Out blocks in the enabled subsystem as follows:
   - The **States when enabling** property of the Enable block must be set to **`held`**.
   - The **Output when disabled** property of the Out block(s) must be set to **`held`**.
4 Connect a TargetLink **Constant** block to the subsystem's Enable port. The output variable of the block must reference the appropriate variable class from the **TargetLink Data Dictionary**.
5 In the **TargetLink Data Dictionary**, specify the variable class of the output variable of the TargetLink **Constant** block as follows:

Property	Value
Storage	`default`
Scope	`global`
Macro	on
UsePreprocessorIf	on
OmitCastOfMacroValue	on

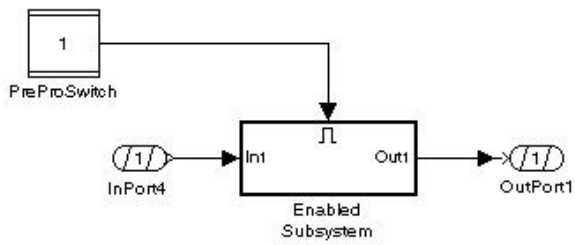
6 Set the data type of the output variable of the TargetLink **Constant** block either to **`Float`** or to an unscaled integer (LSB = 2^0, Offset = 0)
7 Set the output variable of the TargetLink **Constant** block to 0 or 1. If the output is set to 1, the subsystem is enabled, and the residing code is considered during compilation.

Result

You have specified a conditional control flow via an enabled subsystem.

Example

Suppose, you have implemented a conditional control flow as follows (subsystem `ss_root`):



The model results in the following code:

```
...
#if Sa1_PreProSwitch
   /* call of function: ss_root/Enabled
      Subsystem */
   Sa2_Enabled_Subsystem();
#endif
...
```

Related topics

Basics

Basics on Conditional Code Compilation via Preprocessor Directives..... 616
Basics on Controlling Subsystem Execution..... 334
Details on Executing Enabled Subsystems..... 342

HowTos

How to Specify Conditional Control Flows via #If-#Else Statements..... 623

How to Specify Conditional Control Flows via #If-#Else Statements

Objectives

You must specify #If-#Else statements (preprocessor directives) to control which alternative code section is used during compilation.

Possible methods:

There are two ways of implementing #If-#Else statements as preprocessor directives:

- You can use a TargetLink Preprocessor IF block. Refer to Method 1.
- You can use a Simulink If block. Refer to Method 2.

Note

The Simulink If block does not support complex logic (AND and/or OR logic).

Method 1

To specify conditional control flows via #If-#Else statements (via TargetLink Preprocessor IF block)

1. Insert a TargetLink Preprocessor IF block in your TargetLink subsystem.
2. Specify the #If-#Else statements and the block properties, for example, the number of inports and whether an Else outport exists.
3. Connect each outport of the TargetLink Preprocessor IF block to a Simulink IF Action Subsystem.
4. Design the control algorithm that represents the relevant code section inside the Simulink IF Action Subsystem.
5. Specify the Action and Out blocks in the IF Action Subsystem as follows:
   - The **States when execution is resumed** property of the Action block must be set to **`held`**.
   - The **Output when disabled** property of the Out block(s) must be set to **`held`**.

6 Connect TargetLink Constant blocks to all the inports of the TargetLink Preprocessor IF block.

7 Specify the variable class of the output variable of the TargetLink Constant block as follows:
   - Set the Macro property to **On**.

8 Set the data type of the output variable of the TargetLink Constant block either to **Float** or to an unscaled integer (LSB = 2^0, Offset = 0)

9 Specify the output variable of the TargetLink Constant blocks according to your needs.

Method 2

To specify conditional control flows via #If-#Else statements (via Simulink If block)

1 From the Simulink block library, insert an If block in your TargetLink subsystem.

2 Specify the number of inports and whether an Else outport exists.

3 From the Simulink block library, insert IF Action Subsystems in your TargetLink subsystem and connect each outport of the Simulink If block to an IF Action Subsystem.

4 Design the control algorithm that represents the relevant code section inside the Simulink IF Action Subsystem.

5 Specify the Action and Out blocks in the IF Action Subsystem as follows:
   - The States when execution is resumed property of the Action block must be set to **held**.
   - The Output when disabled property of the Out block(s) must be set to **held**.

6 Connect TargetLink Constant blocks to all the inports of the Simulink If block.

7 Specify the variable class of the output variable of the TargetLink Constant block as follows:
   - Set the Macro property to **On**.
   - Set the UsePreprocessorIf property to **On**.
   - Set the OmitCastOfMacroValue property to **On**.

8 Set the data type of the output variable of the TargetLink Constant block either to **Float** or to an unscaled integer (LSB = 2^0, Offset = 0).

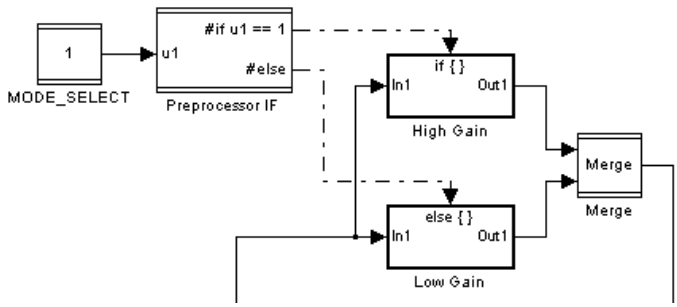
9 Specify the output variable of the TargetLink Constant blocks according to your needs.

Result

You have specified a conditional control flow via #If-#Else statements.

Example

The following example model contains the High Gain and Low Gain subsystems, which are connected to the outports of the TargetLink Preprocessor IF block.



The model results in the following code:

```
...
#if MODE_SELECT == 1
   /* Outport: controller/High Gain/Out1 */
   Sa1_Merge = Sa1_e >> 1;
#else
   /* Outport: controller/Low Gain/Out1 */
   Sa1_Merge = Sa1_e >> 2;
#endif
...
```

Related topics

Basics

Basics on Conditional Code Compilation via Preprocessor Directives..... 616

HowTos

How to Specify Conditional Control Flows via Enabled Subsystems..... 621

References

Preprocessor IF Block (📖 TargetLink Model Element Reference)

How to Include External Macros for Generating Conditional Control Flows

DD module

To generate conditional control flows you must include external macros by using a module in the TargetLink Data Dictionary.

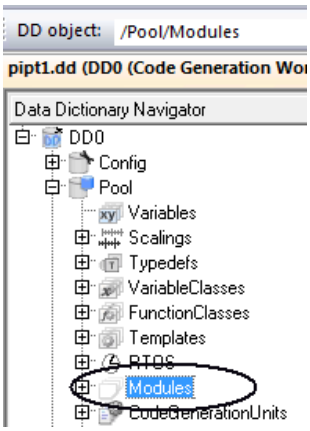
Preconditions

The macros must be specified in a macro definition file (`*.h, *.c`), and the location of the file must be known to the compiler.

Method

To include external macros for generating conditional control flows

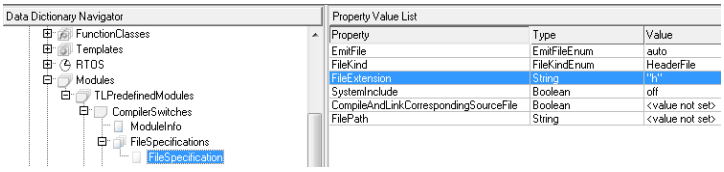
1 In the DD object tree, right-click the `/Pool/Modules` object and select Create Module.



2 Name the module, for example, **`CompilerSwitches`**.

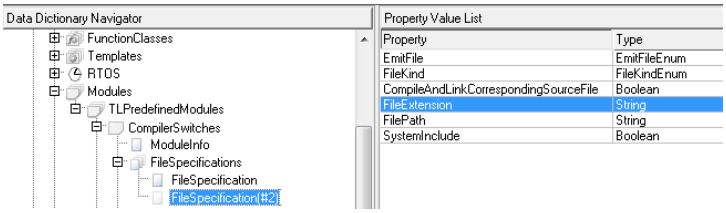
3 Right-click the `/Pool/Modules/CompilerSwitches/FileSpecifications` object and select Create FileSpecification to specify the header file.

4 Specify the header file as follows:



5 Right-click the `/Pool/Modules/CompilerSwitches/FileSpecifications` object and select Create FileSpecification to specify the source file.

6 Specify the source file as follows:



7 In the DD object tree, right-click the `/Pool/VariableClasses` object and select Create VariableClass.

8 Name the variable class, for example, **`IFDEF_MACRO_EXT_DEFINE`**.

9 Specify the variable class as shown below:

Property	Value
UseName	`off`
Optimization	MOVABLE
ModuleInfo	`CompilerSwitches`

Result

You have included an external macro definition file which can be used for generating conditional control flows.

Note

It is not possible to use the **Storage** property of a variable class to declare a macro external.

Related topics

References

Preprocessor IF Block (📖 TargetLink Model Element Reference)

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..... 630
A..... 631
B..... 634
C..... 635
D..... 638
E..... 640
F..... 641
G..... 642
I..... 642
L..... 644
M..... 645
N..... 647
O..... 648
P..... 649
R..... 650
S..... 652
T..... 654
U..... 656
V..... 656
W..... 657

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 on page 298.

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.

Symbols

$CHART 84
$E 84
$THIS 84

A

abstract interface 447
accessing
  function classes 231
  implicit functions 232
  variables 101
  variables via memory address 101
active page pointer 104
actual function argument 80
adapting
  function class template
    default settings 235
adding
  external code 358
  Function block to an atomic subsystem 227
applying
  filter rules 205
atomic
  subsystem 227
avoiding
  function inlining 228
  scaling adjustment code 320

B

backward obsolete parameter removal 528
base type 136
  cut 136
  name
    long type 136
    short type 136
  renaming 136
benefits
  model referencing 413
bus
  mapping bus to explicit struct 479
  mapping bus to implicit struct 468
  sorting components in implicit structs 475
  specifying bus elements individually 491
bypassing
  external code
    SIL 365

C

C standard compliance
  customizing 61
calibratable variables
  collecting 104
call-by-reference parameter 115
call-by-value parameter 115
caller function
  actual function argument 80
capable
  of reentrance 300
cascaded saturation
  code optimization 534
categorizing
  variable classes 90
CGU 49
changing
  struct name 166
code
  files
    generating 99
  optimization 497
    ANSI C compliant code 536
    cascaded saturation 534
    control flow 543
    entire TargetLink system 498
    function interface 528
    interblock optimization 543
    moving code to conditional branches 543
    non-ANSI C code 537
    optimization modules for specific microcontroller/compiler combinations 536
    saturation 534
    TargetLink blocks 534
    via range propagation 551
code block 227
code file
  including via DD module 379
code generation
  customizing 178
  for selected subsystems 406
Code Generator
  configuring 49
Code Generator options 49
code readability
  by using width macros 614
code variants
  specifying 575
CodeGenerationUnit 49
CodeGenerationUnit objects 49
coefficient vector
  offset 140
collecting
  calibratable variables 104
  variables during code generation 104
  variables in separate modules 99
commands
  dsstartup.m 19
Common Program Data folder 16
CommonProgramDataFolder 16
company-specific coding style 136
  variable name 136
conditional code compilation 561
  introduction 562
  preprocessor directives for conditional compilation 616
  specifying conditional control flows via enabled subsystems 621
  specifying conditional control flows via if/else statements 623
conditional control flows 616
control flow 334
  data-based 334
  implementing 334
control flows
  conditional 616
controlling
  function calls from outside TargetLink 336
creating
  a separate C function 227
  a variable class 93
  a variable class group 93
  Data Dictionary project file 209
  readable global variables 101
  user-defined function 236
  writable global variables 101
customization
  TargetLink 38
customizing
  C standard compliance 61
  code generation 178
  MATLAB start-up 19
  preferences 22
  TargetLink environment 22

D

data
  sharing
    data store semantics 340
  types
    user-defined 139
data pages 104
data variants
  activating at ECU run time 583
  coding style 579
  specifying 581
data-based control flow 334
DataItemMapping object 447
default
  behavior
    TargetLink InPort block 112
    TargetLink OutPort block 112
  values
    of variable class properties 93
defining
  the behavior of implicit functions 234
  usertypes 139
  variable class 88
delayed data transfer 342
  input semantic
    latched 342
    not latched 342
developer
  team 406
Discrete Filter block
  offset 140
Discrete Transfer Function block
  offset 140
Documents folder 16
DocumentsFolder 16
dsdd_compare_optionset 49
dsdd_export_optionset 49

dsdd_import_optionset 49
dsstartup.m 19

E

efficiency enhancement
  allocating variables to memory sections 89
  calibration system 89
  external
    user-supplied sources 89
  function interface 89
  re-using variables 89
efficient interface 447
eliminating
  variables 96
embedding
  external code 358
  external functions 360
    bypassing external code for SIL 365
enabled subsystem 335
  conditional code compilation 621
  executing 342
ensuring
  system interface implementation 368
ERASABLE 508
evaluating
  look-up tables 232
example
  inlining a function 355
excluding
  specific subsystems from code generation 406
executing
  enabled subsystems 342
  triggered subsystems 342
external
  stored parameters 109
  user-supplied sources 89
external code
  adding 358
  bypassing external code for SIL 365
  embedding 358
  integrating 358

F

FCN_REF_PARAM 115
FCN_RETURN 115
filter
  rules
    applying 205
    specifying 205
    template 205
flexible parameter 447
flexible parameters
  modeling 456
floating-point CPU
  linear interpolation 541
  look-up tables 541
  table search 541
forcing
  system interface implementation 368
formal function argument 80
forward obsolete parameter removal 528
function
  argument 80
    actual 80
    formal 80
  class objects
    EXTERN_FCN 231
    GLOBAL_FCN 231
    GLOBAL_MACRO 231
    INLINE_FCN 231
    INTERRUPT_FCN 231
    predefined 231
    STATIC_FCN 231
  classes
    accessing 231
    defining 230
    implicit 232
    predefined 231
    selection 233
  dependencies 238
  inlining 228
    Function block 228
  interface
    code optimization 528
    via global variables 242
    via parameters 241
  interface style 241
  interfaces 241
    generated using global variables 241
  return value 114
  reuse 297
    restrictions 302
  scope reduction 531
function code 447
function reuse
  basics 298
  process steps 301
Function System Signatures 252
function-call event 340

G

generating
  active page pointer 104
  code files 99
  data pages 104
  separate modules 99
  switch function 104
  switch variable 104
global
  interface variables 112
global macro 447

H

hardware triggering 342
hook scripts 178

I

if/else statements
  conditional code compilation 623
immediate data transfer 342
implementation variable 447
implementing
  external code 358
  externally function-call triggered subsystems 336
  function-call-triggered subsystems 340
implicit
  function classes 232
  functions 232
    defining the behavior 234
implicit struct
  flat 174
including
  models in other models 413
including code files
  via DD modules 379
incremental
  code generation 406
indirect reuse
  propagation variable 303
initializing
  initialization function 236
  the model 232
initiating
  generation of data pages 104
inlining
  a function 355
  function 228
  inline functions
    INLINE_FCN 238
  threshold 350
inport
  default behavior 112
input
  argument
    of generated function 114
  semantic
    latched 342
    not latched 342
instance
  of a system 300
instance-specific variables 300
  function arguments 311
  leaf struct components (direct reuse) 308
  pointers in reuse structs (indirect reuse) 309
integrating
  production code
    into an existing ECU software environment 109
integration model 447
interblock optimization 543
interface
  global variables 112
invariant scaling 320

K

keeping
  variables in a separate file 99

L

legacy code
  bypassing
    SIL 365
libraries
  becoming width-varianted 610
linear
  interpolation
    on a floating-point CPU 541
Local Program Data folder 16
LocalProgramDataFolder 16
logging
  within a reused function 302
logical instructions
  conditional code compilation 620
look-up tables
  on floating-point CPU's 541

M

macros
  naming
    $E, $THIS, and $CHART 84
making
  a parameter calibratable 101
  block outputs observable 96
mapping
  RDI object to an implementation variable 460
mapping code 447
MATLAB
  customizing start-up 19
MERGEABLE 511
model initialization 120
  via restart function 120
model referencing
  adapting an existing model 422, 430
  basics 413
  benefits 413
  building from scratch 419
  converting an existing model 422
  data-based control flow 415
  generating production code 424
  logging 414, 416
  restoring model reference 442
  simulating 414, 416
modeling
  flexible parameters 456
models
  mapping to production code 381
modular unit 447
module ownership
  introduction 388
modules
  generating 99
  reuse of existing modules 297
MOVABLE 505, 527
multiple
  use of functions 297

N

naming
  name macros 78
    $E, $THIS, and $CHART 84
    function argument 80
  variables automatically 78
  variables manually 78

O

objects
  conditional code compilation 619
offset
  coefficient vector 140
  Discrete Filter block 140
  Discrete State Space block 140
  Discrete Transfer Function block 140
  state space matrix 140
  unsupported 140
optimization
  block outputs followed by signal line splits 519
  direct assignments 518
  ERASABLE 508
  MERGEABLE 511
  MOVABLE 505, 527
  SCOPE_REDUCIBLE 511
  SIDE_EFFECT_FREE 525
  struct function arguments 518
optimizing
  the control flow 543
optimizing structs
  new 521
OptionSet 49
OptionSet object 49
outport
  default behavior 112

P

page switching 104
parameter
  specifying as function parameter 115
parameters
  calibratable 101
  call-by-reference 115
  call-by-value 115
  externally stored 109
  function interface 89
  passing 242
  passing mechanism 241
  via function interface 89
parent model 413
passing
  commands to the linker 187, 193
  signals to the code 241
performing
  consistency checks 406
pointer to struct 164
preconfigured
  variable classes 90
predefined
  function class objects 231
  function classes 231
PreLook-Up Index Search block
  LSB of the fraction 140
preprocessor directives for conditional compilation 616
  handling of logical instructions 620
  handling of objects 619
  implementation possibilities 617
propagation
  of Constant blocks 528
propagation port 303
propagation variable 303

R

range
  code optimization propagation 551
reading
  block output values
    via a calibration tool 96
reentrance 300
referencing
  reference parameters (call-by-reference) 241
removing
  variables from instance-specific structs 312
renaming
  base type 136
ReplaceableDataItem object 447
replacing
  name macros 78
resettable
  states 335
  subsystem 335
restarting
  function 236
restrictions
  function reuse 302
reusable function definition 303
reusable system definition 303
reuse of
  functions 297
  variables 89
root model 413
running
  multiple instances of TargetLink 31
  TargetLink in parallel 31

S

saturation
  code optimization 534
scaling
  invariances 320
scaling adjustment code
  avoiding 320
scope reduction
  functions 531
SCOPE_REDUCIBLE 511
search path
  TargetLink 33, 36
selecting

function classes 233
setting
environment variable during startup 31
preferences 22
settings
template 205
SIDE_EFFECT_FREE 525
signals
passing to the code 241
software
triggering 342
Specify
subsystem's signature 260
TL Function block data 258
specifying
block parameter as function parameter 115
filter rules 205
global communication 337
input arguments of functions 114
subsystem interface 236
the function interface 115
type qualifiers 122
user-defined function 236
speed up
the code generation time 406
start-up
customizing MATLAB 19
state
state space matrix
offset 140
Stateflow variables
becoming width varianted 601
states
resettable 335
static functions
STATIC_FCN 238
storing
parameters externally 109
struct
changing the name 166
name 176
pointer 164
struct name 176
subsystems
atomic 227
becoming width-varianted 607
control execution 334
enabled 335
externally function-call-triggered 336
function-call triggered 334
resettable 335
triggered 335
user-defined 228
virtual 227
supported
name macro 78
switch function 104
switch variable 104
switch variable for data variants
creating 573
Synchronizing
Function System Signatures 252

T

table
search
on a floating-point CPU 541
TargetLink
cloning targets/compilers 44
DD-related customization files 38
defining search path 33, 36
exporting preferences 26
importing preferences 26
returning search path 35
TargetLink Preferences Editor 22
templates
consisting of 205
filter rules 205
settings 205
TL Function block
synchronize 258
triggered subsystems 335
executing 342
type
of function input argument 114

U

user-defined
code editor 25
data type 139
function 227
subsystem 228
user-defined data types
vectorized variables 140
usertypes
struct 139

V

value
parameters (call-by-value) 241
variable class
defining 88
variable propagation 303
variables
classes
categorizing 90
creating 93
default values 93
definition 88
externally stored parameters 109
group creating 93
preconfigured 90
separate codefile 99
collecting during code generation 104
global interface variables 112
names
company-specific coding style 136
re-use of 89
width variants 600
width-varianted 586
variant coding 561, 564
activating a variant configuration at code generation time 576
activating data variants at ECU run time 583
coding styles for data variants 579
creating a switch variable for data variants 573
creating code variants 575
creating variant configurations 570
creating variant items 568
creating width macros 596
creating width variants 598
defining variant configurations 570
defining width macros externally 604
introduction 562
making libraries width-varianted 610
making Stateflow variables width varianted 601
making subsystems width-varianted 607
principle of width-varianted variables 586
specifying code variants 575
specifying data variants 581
terminating width propagation 603
testing width-varying variables 605
width variants 600
variant configuration
activating at code generation time 576
variant configurations
creating 570
variant items
creating 568
vectorized variables
user-defined data types 140
virtual subsystem 227

W

width macros
creating 596
external definition 604
improving code readability 614
making libraries width-varianted 610
making subsystems width-varianted 607
referenced in the model 611
terminating width propagation 603
width variants
creating 598
width-varianted variables 586
implementation details 592
specification details 589
testing validity 605
working
on a subsystem individually 406