TargetLink

Interoperation and Exchange 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

Contents

About This Guide

Contents

This guide shows you how to use TargetLink as part of a tool chain, for example, exchanging data with other tools or using TargetLink via command line. It outlines in detail, for example, how to annotate the code, generate reports, integrate TargetLink code in existing ECU code, generate virtual ECUs, use the TargetLink MATLAB script interface (API), work with A2L files, and localize production code by using different character sets.

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 TargetLink users and their colleagues that make use of TargetLink artifacts such as code files, A2L files or reports. 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.

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.

Including Requirement Information in Code and Documentation

Introduction

You can add requirement information to your TargetLink model via DD **RequirementInfo** objects or via the requirement management interface of the Simulink Requirements™ software from MathWorks®.

If a TargetLink model contains model elements that are linked to requirements, TargetLink can include information on these requirements in the generated code and in the generated documentation.

This traceability between requirement, model, and code is helpful if you want to verify whether a requirement is correctly implemented in the software, for example, by inspecting the generated code.

Where to go from here

Information in this section

Introduction

Basics on Requirement Information in the Generated Code

Requirement management

The first step in the development of a controller is usually the definition of requirements. These requirements can contain specifications about functionality, characteristics, quality or performance. They can be stored in requirement management systems such as IBM® Rational® DOORS® or in other file formats.

Requirement information

TargetLink can include requirement information in the model as comments into production code and in generated documentation. This is possible via DD RequirementInfo objects or via the requirement management interface of Simulink Requirements software from MathWorks.

Examples of requirement information comments in production code

During code generation, TargetLink tries to associate the requirement information of each model element to code fragments. If possible, the requirement information is included as comments in the generated code.

The comments contain the requirement description that you provided, e.g., the name of the document that contains the requirement, the location of the requirement in that document, and a user tag. In the following example, a requirement is linked to the `Controller/sPI` Sum block:

```
void controller(void)
{
...
   /* Sum: controller/sPI
      # combined # Gain: controller/Kp
      # combined # Gain: controller/Ki

      Requirements:
      controller/sPI
      Description: Requirement for sPI
      Document    : C:\Documents\RequirementsDocument.doc
      Location    : RequirementID_2
      User Tag    : none */
   Sa1_sPI = (Int16) (((UInt16) Sa1_e) + ((UInt16) (Int16) ((((Int32)
Sa1_si) * 13107) >> 16)));
...
}
...
```

The following code shows requirement information comments that result from one of the following:

- Simulink-based requirement information that is stored at the subsystem
- A DD RequirementInfo object referenced at the Function block contained in the subsystem

```
/**************************************************************************************************\
 ***  FUNCTION:
 ***      Sa2_Linearization
 ***
 ***  DESCRIPTION:
 ***      Linearization of measured position signal
 ***
 ***  PARAMETERS:
 ***      Type                 Name                 Description
 ***      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 ***
 ***  RETURNS:
 ***      static void
 ***
 ***  SETTINGS:
 ***
 ***  REQUIREMENTS:
 ***      controller/Linearization
 ***      Description: Requirement for Linearization
 ***      Document    : C:\Documents\RequirementsDocument.doc
 ***      Location    : RequirementID_1
 ***      User Tag    : none
 ***
\**************************************************************************************************/
static void Sa2_Linearization(void)
{
...
}
...
```

Note

By using the model-linked code view feature of TargetLink, you can navigate between model elements and their generated code parts. However, requirement information and their corresponding code comments are not directly navigateable/clickable.

Associating requirements with code fragments

TargetLink can associate requirements with code fragments only in certain situations:

- The requirement is referenced at calculating blocks, e.g., Gain or Unit Delay blocks.
- The requirement is referenced at a subsystem block (Simulink-based requirements only).
- The requirement is referenced at one of the following Stateflow objects.
  - Stateflow chart
  - Stateflow data object properties
  - Stateflow subchart (Simulink-based requirements only)
  - Stateflow transitions (Simulink-based requirements only)
  - Stateflow states (Simulink-based requirements only)
  - Stateflow graphical functions (Simulink-based requirements only)
  - Stateflow box (Simulink-based requirements only)

Specifics

- At the Function block, TargetLink can associate only DD RequirementInfo objects requirements with code fragments. Simulink requirements cannot be associated.
- Requirements at TargetLink InPort blocks of atomic subsystems are included in the production code at the location, where the variable for the InPort block is updated. For subsystems configured for incremental code generation, this causes the requirement to be included in the production code of the surrounding system. For the TargetLink subsystem, this implies that the requirement cannot be found in the production code. In this case, a warning message occurs.
- If subsystems or charts are reused multiple times, you can link every instance of them to different requirements. TargetLink includes the requirement information in the function header comment of the generated function. However, only the requirements of those instances that are part of the current code generation unit are included. Requirement information of all other instances are dropped without displaying a warning or note.

  To avoid the loss of requirement information of a reused system, ensure that all instances of the reused system are linked to the same requirements.

Controlling creation of requirement information comments

Controlling requirement information comments You can control the creation of requirement information comments in the production code via the RequirementInfoAsCodeComment Code Generator option.

By default, the option is `off` and TargetLink does not generate requirement information comments into production code.

Controlling messages TargetLink displays a message if it cannot create an requirement information comment for a model element that has requirement information.

You can control whether this message is displayed as a warning or a note via the EmitMissingRequirementsWarningsAsNotes Code Generator option.

By default, the option is `off` and TargetLink displays a warning.

Note

It is your responsibility to verify that all specified requirements are linked to the model. TargetLink only checks if linked requirements are included as comments in the code.

Documenting requirement information

You can control whether and how TargetLink adds the requirement information to its documentation.

Call the **`tldoc('TargetLink Code Generation Units', docFid, propertyName, propertyValue, ...)`** API function with its PrintRequirementInfo and RequirementInfoWithScreenShot properties set as required. By default, both are on.

For further details, refer to Generating Documentation on Model Characteristics on page 23.

Tip

Requirements that are linked to inports of incremental or referenced systems are not documented in the section that describes the incremental or referenced system. They are documented in the section of the system in which the incremental or referenced system is embedded.

Effects on code optimization

If you use code optimization (= optimization option is enabled), the handling of requirement comments in the generated code is affected as follows:

- When TargetLink optimizes the production code, expressions might be simplified and statements might be removed. If requirements are linked to code fragments that are deleted during optimization, the requirement information might not be included as code comments in the generated code. TargetLink generates messages for these missing links.
- When the code for several model elements is optimized into a single statement, all comments, including requirement information comments, are merged at that statement.
- When function inlining is performed, the requirements linked to the function are commented in the calling function at the beginning of the inlined code.

  When an inlined function terminates with a return statement, the requirement information comment of the return statement is assigned to the originating assignment statement.

  For more information on function inlining, refer to Controlling Function Inlining (📖 TargetLink Customization and Optimization Guide).

Related topics

Basics

Adding Requirements via DD RequirementInfo Objects.. 18
Adding Requirements via Simulink's Requirements Software.. 21

References

EmitMissingRequirementsWarningsAsNotes (📖 TargetLink Model Element Reference)
RequirementInfoAsCodeComment (📖 TargetLink Model Element Reference)
tldoc('TargetLink Code Generation Units', docFid, propertyName, propertyValue, ...) (📖 TargetLink API Reference)

Adding Requirements via DD RequirementInfo Objects

Basics on Using DD-Based Requirement Information

Requirement information in the DD

The Data Dictionary provides the `/Pool/RequirementInfos` subtree to store requirement information. You can add DD RequirementInfo objects, which hold the requirement information. These objects act as a proxy for your requirements management system.

Each DD RequirementInfo object lets you define a requirement information via its properties:

Property	Description
Description	Lets you describe the requirement. The value is displayed in the Description field on the Requirements page of the model element's dialog and appears as the `Description` part of a requirement information comment in production code.
Document	Lets you specify the document the requirement is defined in. The value appears as the `Document` part of a requirement information comment in production code.
Location	Lets you specify the location of the requirement within the document. The value appears as the `Location` part of a requirement information comment in production code.

Referencing RequirementInfo objects

You can reference DD RequirementInfo objects at the following model elements:

Model Element	Access
TargetLink blocks	Open the block dialog and select the Requirements page.
Stateflow Charts	Open this object's dialog via the Property Manager and select the Requirements page.

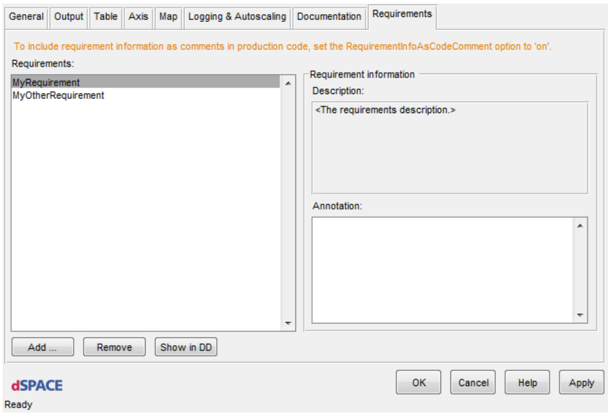
On the Requirements page of each model element's dialog, you can add references to DD RequirementInfo objects. Via the Show in DD button, you can navigate to the selected DD RequirementInfo object in the DD object tree.

Mapping relations You can reference the same DD RequirementInfo object at several model elements and vice versa, i.e., n-n mappings are possible.

Annotations You can specify an annotation for each reference of a DD RequirementInfo object via the Annotation edit field of the model element's dialog. Each reference to the same DD RequirementInfo object has its own annotation. Annotations appear as the `User Tag` part of requirement information comments in production code.

Example

The following block dialog references the DD RequirementInfo object named MyRequirement:



To include requirement information as comments in production code, set the RequirementInfoAsCodeComment option to 'on'.

Requirements:

MyRequirement
MyOtherRequirement

Requirement information

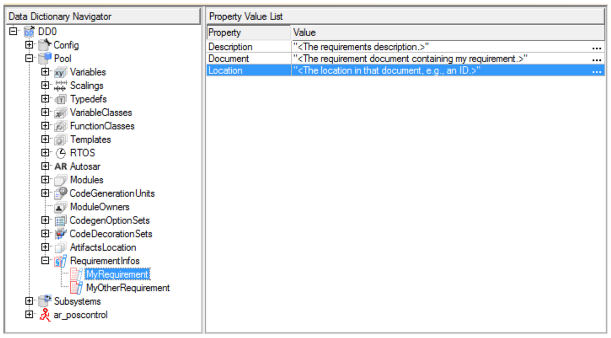
Description:

<The requirements description.>

Annotation:

Ready

Representation in the DD The following screenshot shows MyRequirement in the Data Dictionary:



Property Value List

Property	Value
Description	"<The requirements description.>"
Document	"<The requirement document containing my requirement.>"
Location	"<The location in that document, e.g., an ID.>"

Representation in production code The following code example shows the requirement information comment resulting from the DD RequirementInfo object:

```
/*****************************************************************************************
***************\
...
Requirements:
<Subsystem>/<ModelElement>
Description: <The requirements description.>
Document   : <The requirement document containing my requirement.>
Location   : <The location in that document, e.g., an ID.>
User Tag   : <Some annotation that qualifies MyRequirement.>
...
\*****************************************************************************************
***************/
```

Managing requirement information via the API

TargetLink provides the **`tlRequirementInfo()`** API function. Use this API function instead of **`tl_get()`** or **`tl_set()`** to pragmatically manage requirement information at model elements.

Tip

To generate a list of TargetLink blocks that reference to DD **RequirementInfo** objects, you can use the API:

```
hBlockList = tl_find('<System>', 'HasRequirementInfos', 1)
```

Related topics

Basics

Basics on Requirement Information in the Generated Code.. 14
Modifying Multiple Properties at Once via the Property Manager (📖 TargetLink Preparation and Simulation Guide)

References

Property Manager (📖 TargetLink Tool and Utility Reference)
tl_get (📖 TargetLink API Reference)
tl_set (📖 TargetLink API Reference)
tlRequirementInfo (📖 TargetLink API Reference)

Adding Requirements via Simulink's Requirements Software

Introduction

TargetLink supports the requirement management interface of Simulink Requirements software from MathWorks.

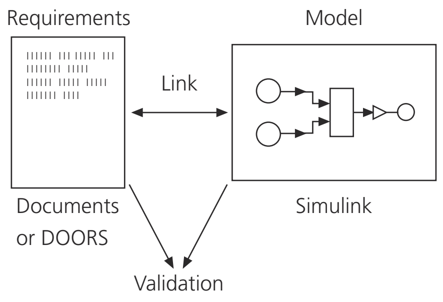
TargetLink blocks can be linked to software requirements, which are stored in requirement documents or requirement management tools, in the same way as to Simulink and Stateflow objects. If a TargetLink model contains objects that are linked to requirements, TargetLink can include information on these requirements in the generated code and in the generated documentation.

This traceability between requirement, model, and code is helpful if you want to verify whether a requirement is correctly implemented in the software, for example, by inspecting the generated code.

Basics on Requirement Information Related to Simulink Requirements

Requirement management

By using the Simulink Requirements software, you can link the requirements to Simulink and Stateflow objects.



You can link requirements to the following Simulink/Stateflow objects:

- Simulink models and subsystems
- Simulink blocks
- Stateflow charts and states
- Stateflow transitions, boxes, and graphical functions

The link information contains a description for the requirement, the name of the document that contains the requirement, the location of the requirement in that document, and a user tag (for user-specific key words).

Requirement support by TargetLink

If the Simulink Requirements software is installed on your host PC, TargetLink supports its requirement management interface as follows:

- Enhancing a Simulink model to a TargetLink model does not change any requirements that are already linked to the Simulink model or its objects.

- You also can link your requirements directly to your TargetLink subsystems and blocks. This is useful if you build a TargetLink model from scratch or if you are working in the Modeling Only operation mode.

Note

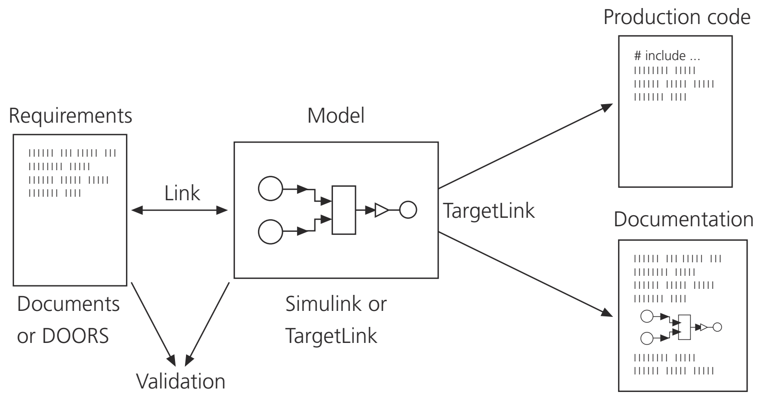
TargetLink ignores requirements linked to a TargetLink subsystem via the subsystem's context menu. Therefore, you must not use a subsystem's context menu to link a requirement to it.

To link a requirement to a TargetLink subsystem, open it and use the commands in its menu.

For more information on linking requirements, refer to the *Simulink Requirements User's Guide* provided by MathWorks.

Requirement information in TargetLink output files

If your TargetLink model contains objects that are linked to requirements, TargetLink can include information on these requirements in the generated code and in the generated documentation. As a precondition, Simulink's Requirements software must be installed on your host PC.



This traceability between requirement, model, and code makes it easy for you to verify whether a requirement is correctly implemented in the software, for example, by inspecting the generated code.

Generating Documentation on Model Characteristics

Where to go from here

Information in this section

Basics on Documentation Generation

Documenting the generated code

In the final phase of developing production code with TargetLink, you may want to document the generated code and the Simulink model including Stateflow charts that the production code has been generated from.

Suppose you want to integrate the generated code into the ECU environment. You need to know the following:

- How to call the functions from the RTOS kernel (function prototype, sample time, etc.) that were generated by TargetLink
- Scaling parameters and range information of integer variables
- Calibratable and displayable variables in the code

In addition, the following aspects may be useful:

- Properties of the Simulink model
- Code Generator options
- Screen copies of simulation results
- Statistical data
- Version and date information

Predefined M scripts

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

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

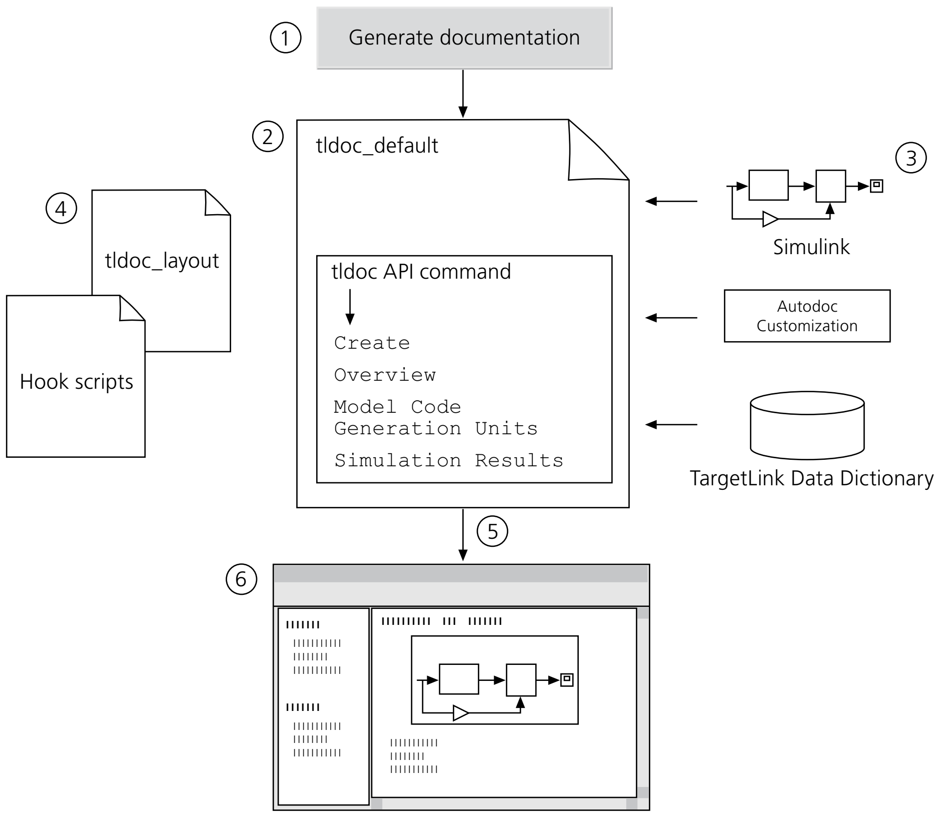
You can call these scripts in the MATLAB Command Window or specify them in the Script file edit field on the Tools page of the TargetLink Main Dialog Block.

Note

Do not modify these files. For customizing the generation of documentation, use TargetLink's customization file templates. For details, refer to Basics on Customizing the Generated Documentation on page 29.

Process

The document generation process is started via the Tools page of the TargetLink Main Dialog. The following illustration shows the standard process and information flow during document generation:



1. The M file (by default, `tldoc_default`) that controls the document generation process is run.
2. The actions of the `tldoc` API function (`Create`, `Overview`, and so on) are executed as defined in the M file.
3. The information included in the documentation is retrieved from the Simulink model and the TargetLink Data Dictionary. In addition, you can configure the documentation generation for specific parts of the model with the Autodoc Customization block. Refer to Basics on Customizing the Generated Documentation on page 29.
4. The documentation layout is modified as defined in `tldoc_layout`. You can also customize the documentation generation further by using your own hook scripts. Refer to Basics on Customizing the Generated Documentation on page 29.
5. All documentation files (text and images) are created and saved in the folder defined in the M file that controls the document generation process (by default, in the `doc` subfolder of the folder for production code files defined via the DD ProjectFolder and FolderStructure objects.
6. The generated documentation appears on screen.

Structure of the generated documentation

Documentation overview Depending on the settings, the generated documentation consists of up to five chapters:

- Overview
- Code Generation Units
  - Model Code Generation Units
  - Data Dictionary Code Generation Units
- Simulation Results

The information included in the documentation is retrieved from the Simulink model and the TargetLink Data Dictionary. The generated documentation contains hyperlinks that help you navigate through large documents and jump to other files, like the generated C code. You can customize each chapter by changing the properties in the M file that controls the document generation process.

Overview chapter This chapter shows an overview of the properties of the model and the settings of the TargetLink Main Dialog. It is generated when the `Overview` action is invoked in the M file that controls the document generation process.

If you want to turn off the screenshot of the model or include the author's name and department, you can change the values of the `Overview` action's properties accordingly. For detailed information on the properties of the `Overview` action, refer to tldoc('Overview', docFid, propertyName, propertyValue, ...).

TargetLink Code Generation Units This chapter shows a detailed description of each TargetLink code generation unit in the model (and in the Data Dictionary, if applicable), including date and version information, settings of the code generator, function prototypes, scaling data, lists of calibratable and displayable variables, etc. It is generated when the `TargetLink Code Generation Units` action is invoked in the M file that controls the document generation process.

If you want to exclude certain functions from the documentation or the general information or statistical data, you can change the values of the `TargetLink Code Generation Units` action's properties accordingly. For detailed information on the properties of the `TargetLink Code Generation Units` action, refer to tldoc('TargetLink Code Generation Units', docFid, propertyName, propertyValue, ...).

Simulation Results chapter This chapter shows the simulation results that are currently stored by the TargetLink Data Server (TLDS). Simulations are displayed via symbolic names and comments, which can be assigned on the Simulation page of the TargetLink Main Dialog. Screenshots of simulated signal histories can also be added. This chapter is generated when the `Simulation Results` action is invoked in the M file that controls the document generation process.

If you want to exclude certain simulations from the documentation or generate a separate plot for each signal, you can change the values of the `Simulation Results` action's properties according to your requirements. For detailed information on the properties of the `Simulation Results` action, refer to tldoc('Simulation Results', docFid, propertyName, propertyValue, ...) (📖 TargetLink API Reference).

Related topics

Basics

Basics on the TargetLink Data Dictionary (📖 TargetLink Data Dictionary Basic Concepts Guide)

References

API Functions (📖 TargetLink API Reference)
Files Related to Documentation Generation Customization (📖 TargetLink File Reference)
TargetLink Main Dialog Block (📖 TargetLink Model Element Reference)
tl_autodoc_hook (📖 TargetLink File Reference)
tldoc (📖 TargetLink API Reference)
tldoc('Overview', docFid, propertyName, propertyValue, ...) (📖 TargetLink API Reference)
tldoc('Simulation Results', docFid, propertyName, propertyValue, ...) (📖 TargetLink API Reference)
tldoc('TargetLink Code Generation Units', docFid, propertyName, propertyValue, ...) (📖 TargetLink API Reference)

How to Generate the Documentation

Objective

TargetLink's Generate documentation tool generates documentation that is always up-to-date with the generated code. By default, the document is generated in HTML format and can be viewed with a standard Web browser. You can also convert the generated HTML documentation to a PDF. For more information, refer to the tldoc('Convert', propertyName, propertyValue, ...).

Preconditions

The following preconditions must be fulfilled before you can perform the subsequent instructions.

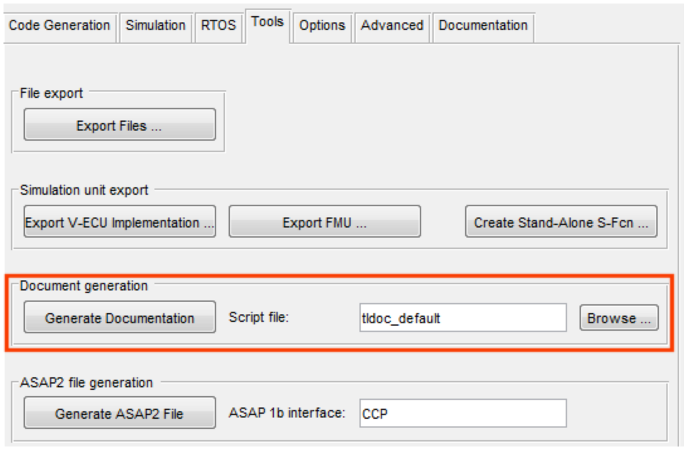
- Production code is generated for the code generation units you want to document.
- The TargetLink Data Dictionary project file with the description of the production code generated for the code generation units (in DD Subsystem objects) is loaded.
- The Simulink model that contains the code generation units is open.

Method

To generate the documentation

1 Open the TargetLink Main Dialog, and select the Tools page.

2 In the Document generation group box, specify the M file (customization file) that you want to use to control the document generation process.



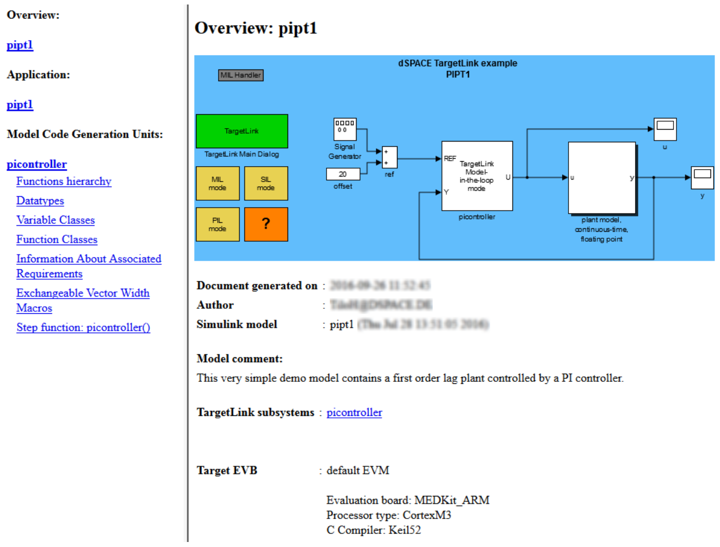
Script file: tldoc_default

ASAP 1b interface: CCP

3 Click the Generate Documentation button to start the generation process. As an alternative, you can also call the script in the MATLAB Command Window. This method is also recommended if you want to document the DD-based code generation units only. In this case a Simulink model is not necessary.

Result

The following illustration shows part of a typical document generated by `tldoc_default`.



Overview:

pipt1

Application:

pipt1

Model Code Generation Units:

picontroller
- Functions hierarchy
- Datatypes
- Variable Classes
- Function Classes
- Information About Associated Requirements
- Exchangeable Vector Width Macros
- Step function: picontroller()

Overview: pipt1

Document generated on : [redacted]
Author : [redacted]
Simulink model : pipt1 [redacted]

Model comment:
This very simple demo model contains a first order lag plant controlled by a PI controller.

TargetLink subsystems : picontroller

Target EVB : default EVM

Evaluation board: MEDKit_ARM
Processor type: CortexM3
C Compiler: Keil52

Customizing the generated documentation

You can customize the documentation generation, for example to add user-defined contents. Refer to Basics on Customizing the Generated Documentation on page 29.

Related API function

For information on how to generate documentation via the command line, refer to tldoc.

Related topics

References

API Functions (📖 TargetLink API Reference)
tldoc (📖 TargetLink API Reference)
tldoc('Convert', propertyName, propertyValue, ...) (📖 TargetLink API Reference)

Basics on Customizing the Generated Documentation

Customizing documentation generation via templates

You can derive customization files from their templates to customize the documentation process. The following templates are available:

Template	Description
`tldoc_html_customized`	Template file for customizing the HTML documentation generation.
`tldoc_layout`	Template file for customizing the language and table layout of generated documentation.
`tldoc_pdf_customized`	Template file for customizing the PDF documentation generation.

Using the templates ensures that all preconditions for the documentation generation are met and that the information gathered by each tldoc call is added to the appropriate documentation file.

In the derived customization file, change the calls to **`tldoc`** as required.

Deriving customization files from their 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 (📖 TargetLink Customization and Optimization Guide).

Autodoc Customization block

The block lets you customize documentation generation further. It lets you add additional descriptions to the standard chapters of the documentation or lets you insert additional elements as a new chapter. You can even define the position the additional elements should be placed to. In addition, you can instruct TargetLink's Documentation Generator to consider only

Autodoc Customization blocks with certain tags during documentation generation.

Inserting screenshots By default, TargetLink includes a screenshot of the root level of a subsystem or chart if all of the following conditions are fulfilled:

- A function was generated for the subsystem or chart.
- The function was not inlined.

If you want to include screenshots for other subsystems or charts, refer to How to Include User Defined Information in the Documentation on page 32.

Inserting User-Defined Information You can insert your own sections in the generated documentation. You can format these additional sections by using text elements (embedded commands). For example, you can make selected sentences bold, italic or underline. You can also add your own graphics and hyperlinks to the section. In addition, text elements lets you add the content of text or HTML files to your documentation. Refer to How to Include User Defined Information in the Documentation on page 32. For a complete list of text elements, refer to the Parameters Page.

Generating documentation for specific functions You can generate documentation separately for specific functions by placing an Autodoc Customization block in the corresponding subsystems the functions have been generated from. In the derived customization file, you must then set the `AutoDocFunctionSelection` property to `on`. For details, refer to How to Document Specific Functions on page 34.

Filtering blocks using tags You can instruct TargetLink's Documentation Generator to consider only Autodoc Customization blocks with certain tags during documentation generation. This allows you to create documentation with different contents for different use cases, for example, for specific customers. For details, refer to How to Filter Autodoc Customization Blocks by Using Tags on page 35.

Incremental documentation generation

As a part of the incremental code generation feature, you can also generate the documentation incrementally. By default, TargetLink generates new documentation for all code generation units (CGUs) that belong to the specified model hierarchy: TargetLink subsystems, subsystems configured for incremental code generation and referenced models. That means, already generated and existing documentation for some of the specified code generation units is not taken into account: In each documentation run for each specified code generation unit new documentation is (re-)generated.

With activated incremental documentation generation, TargetLink generates the documentation 'bottom-up': For a specified code generation unit, TargetLink does not generate the documentation for the inner incremental code generation units again, but embeds the existing documentation into the currently generated one. However, if no documentation exits, TargetLink generates the missing documentation of the incremental code generation units including all further incremental code generation units within its hierarchy. This means you can flexibly choose the way that better matches your modular development style, top-down or bottom-up. The following ways apply:

- You can generate *one* documentation as a final step at the end.
- You can collect already generated and finalized modular/incremental documentations at the end to create the whole documentation.

You can specify the code generation units to be documented incrementally in the derived customization file.

- For a description on how to specify incremental documentation generation for code generation units, refer to How to Specify Incremental Documentation Generation on page 37.
- For further information on incremental code generation, refer to Basics on Incremental Code Generation (📖 TargetLink Customization and Optimization Guide).
- A demo model also shows incremental code generation. Refer to MODEL_REFERENCING (📖 TargetLink Demo Models).

Further customization via hook scripts

You can customize the documentation generation further by using your own hook scripts which can be invoked at various points during documentation generation.

- You can add your own text and images to each of the chapters via the **`tl_autodoc_hook`** (📖 TargetLink File Reference) hook script, derived from its template.
- You can control Autodoc Customization blocks, for example you can change the order those blocks are considered. Refer to tl_post_autodoc_filter_hook (📖 TargetLink File Reference).

For basic information on using hook scripts, refer to Basics on Using Hook Scripts (📖 TargetLink Customization and Optimization Guide).

Customizing the appearance of tables

You can customize the appearance of tables. You may want to alter the column widths in tables or hide columns that are not required in the documentation. Fort details, refer to How to Customize the Appearance of Tables in the Documentation on page 38.

Changing the Documentation Language

You can add custom languages to the documentation by providing custom language strings. For details, refer to How to Change the Documentation Language by Providing Custom Language Strings on page 41.

Demo model

You can inspect a demo model that also shows how to customize the generated documentation. Refer to AUTODOC_GENERATION (📖 TargetLink Demo Models).

Related topics

Basics

Deriving Customization Files From Their Templates (📖 TargetLink Customization and Optimization Guide)

References

Autodoc Customization Block (📖 TargetLink Model Element Reference)
Create Customization Files Dialog Description (📖 TargetLink Tool and Utility Reference)
Files Related to Documentation Generation Customization (📖 TargetLink File Reference)
tlCustomizationFiles (📖 TargetLink API Reference)
tldoc (📖 TargetLink API Reference)

How to Include User Defined Information in the Documentation

Objective

You may want to add additional contents to the standard chapters of the documentation or include some information as a new chapter.

Adding external files

You can add external files to the documentation, for example, a picture (as supported by your browser).

TargetLink Modeling Only restriction

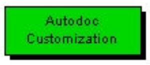
If you use the Modeling Only operation mode and have no license for code generation or other modules of the TargetLink Base Suite, you cannot generate documentation (only in Full-Featured operation mode). However, you can configure it. For details, refer to Overview of the TargetLink Operation Modes (📖 TargetLink Blockset Guide).

Method

To include user defined information in the documentation

1 Open the TargetLink library tllib.

2 Insert an Autodoc Customization block into one of the following subsystems according to your needs:



- If you want to add a screenshot of a virtual subsystem, insert the block into this subsystem.
- If you want to add additional information to a function, insert the block into the subsystem which the function has been generated from.

- If you want to add additional information to the TargetLink subsystem chapter, insert the block into the desired TargetLink subsystem.
- If you want to add additional information to the Overview chapter, insert the block on the root level of your model.

  You can now open the block dialog to make the necessary settings.

3. Double-click the inserted block to open the block dialog.
4. In the **Caption** edit field, enter the text that is used as the title for the chapter that you want to add.
5. In the next edit field, you can enter additional information, for example, a description of the function.

   The text entered here appears below the title specified in step 4. You can use specific commands to format the text, and to add hyperlinks, files, and graphics. For example, to add a graphic click the **Add Text Element** button and select the **Insert Image File** embedded command. You can also manually type the command as follows: `$insertimage:<filepath> $endinsertImage:`.

   - For a description of all embedded commands, refer to the **Parameters Page**.
6. To add the user-defined information in the documentation as a new chapter, select the **Add as a new chapter** checkbox.
7. To include a screenshot of the subsystem or Stateflow (sub-)chart, select the **Add screen copy of Simulink subsystem or SF Chart/Subchart** checkbox.
8. In the **Documentation placement** list, select the location where the description or screenshot is to be included.
9. Click **OK** to save the settings.
10. Repeat steps 3 to 10 for each comment or screenshot that you want to add to the documentation.

Note

If you select a chart in the **Associate with SFChart/Subchart** list, documentation is generated only for the associated Stateflow chart.

Result

If you now generate the documentation, the specified information is included.

Related topics

HowTos

References

Autodoc Customization Block (📖 TargetLink Model Element Reference)

How to Document Specific Functions

Objective

At some stage of the development process you may want to generate documentation for specific functions only or exclude a function that is not complete. You can also include functions generated from Stateflow charts.

Precondition

The following preconditions must be fulfilled:

- The model that you want to document is open.
- You derived a customization file (you should not use the `tldoc_default` script) as described in Basics on Customizing the Generated Documentation on page 29 and specified it on the Tools page of the TargetLink Main Dialog Block.

Method

To document specific functions

1. In the MATLAB Command Window, type
   `edit <name of your customization file>`.
   The customization file is displayed in the editor.
2. Scroll to the **`TargetLink Code Generation Units`** section, and set the `AutoDocFunctionSelection` property to `on`.
   This ensures that documentation is generated *only* for functions belonging to subsystems containing an Autodoc Customization block.
3. Save and close the M file.
   The customization file that controls the document generation process is now ready. You now have to insert a TargetLink Autodoc Customization block into the subsystem of the function that you want to document and specify some settings.
4. Open the TargetLink block library tllib.

5 If the function that you want to document has been generated from a Simulink subsystem, insert an **Autodoc Customization** block into the subsystem, and open the block dialog. Proceed with step 7.

  Or

  If the function that you want to document has been generated from a Stateflow chart, insert **Autodoc Customization** block into the subsystem where the chart is located, and open the block dialog. Proceed with step 6.

6 From the **Associate with SF Chart/SubChart** drop-down list select the Stateflow chart or subchart which the function has been generated for.

7 If you want to add a description of the function or a comment, or include a screenshot in the documentation, refer to How to Include User Defined Information in the Documentation on page 32. Otherwise, clear the relevant checkboxes and remove any text in the edit fields.

8 Click **OK** to save the settings.

9 Repeat steps 5 to 8 for each function that you want to include in the documentation.

Result

If you generate the documentation, the functions are included in the documentation.

Related topics

HowTos

How to Generate the Documentation.. 27
How to Include User Defined Information in the Documentation.. 32

References

Autodoc Customization Block (📖 TargetLink Model Element Reference)
TargetLink Main Dialog Block (📖 TargetLink Model Element Reference)
tldoc('TargetLink Code Generation Units', docFid, propertyName, propertyValue, ...) (📖 TargetLink API Reference)

How to Filter Autodoc Customization Blocks by Using Tags

Objective

You can filter the **Autodoc Customization** blocks by using tags and define filters in the derived customization file for them. Only **Autodoc Customization** blocks with tags equal to the one specified in the filter are taken into account by TargetLink's documentation generator.This is useful if you want to create different (e.g. customer-specific, or internal and external) documentation sets.

Precondition

The following preconditions must be fulfilled:

- The model that you want to document is open.
- You derived a customization file (you should not use the `tldoc_default` script) as described in Basics on Customizing the Generated Documentation on page 29 and specified it on the Tools page of the TargetLink Main Dialog Block.

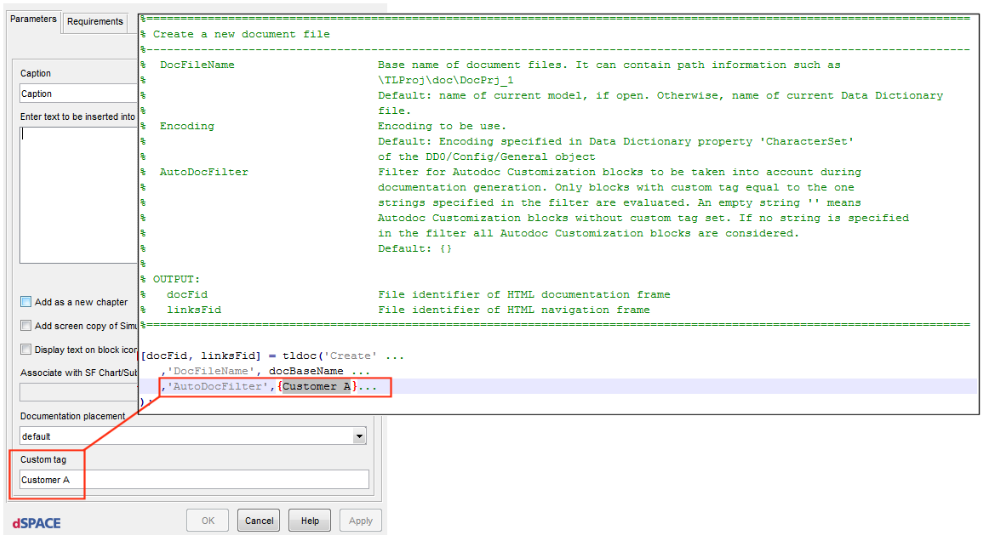
Method

To filter the documentation generation

1. Place an Autodoc Customization block you want to add a filter tag to in the subsystem.
2. Specify a tag in the Custom tag field on the Parameters page of block, for example, `Customer A`, and configure the Autodoc Customization block as required for this customer, for example by adding an additional explanation.
3. Repeat steps 1 and 2 with another Autodoc Customization block in a subsystem for `Customer B`. Configure the Autodoc Customization block as required for `Customer B`.
4. In the MATLAB Command Window, type

   `edit <name of your customization file>`.

   The customization file is displayed in the editor.
5. Scroll to the **`Create a new document file`** section, and enter a tag string in the `AutoDocFilter` property as shown in the example below:



```
%=====================================================================================================
% Create a new document file
%-----------------------------------------------------------------------------------------------------
%  DocFileName                    Base name of document files. It can contain path information such as
%                                 \TLProj\doc\DocPrj_1
%                                 Default: name of current model, if open. Otherwise, name of current Data Dictionary
%                                 file.
%  Encoding                       Encoding to be use.
%                                 Default: Encoding specified in Data Dictionary property 'CharacterSet'
%                                 of the DD0/Config/General object
%  AutoDocFilter                  Filter for Autodoc Customization blocks to be taken into account during
%                                 documentation generation. Only blocks with custom tag equal to the one
%                                 strings specified in the filter are evaluated. An empty string '' means
%                                 Autodoc Customization blocks without custom tag set. If no string is specified
%                                 in the filter all Autodoc Customization blocks are considered.
%                                 Default: {}
%
% OUTPUT:
%   docFid                        File identifier of HTML documentation frame
%   linksFid                      File identifier of HTML navigation frame
%=====================================================================================================

[docFid, linksFid] = tldoc('Create' ...
   ,'DocFileName', docBaseName ...
   ,'AutoDocFilter',{Customer A}...
);
```

Caption: Caption
Enter text to be inserted into
Add as a new chapter
Add screen copy of Simu
Display text on block icon
Associate with SF Chart/Sub
Documentation placement: default
Custom tag: Customer A

If done, a regular string comparison retrieves all blocks with the specified tag(s) (exact match). An empty string `''` means Autodoc Customization blocks without a custom tag. If no string is specified in the `AutoDocFilter` filter, all Autodoc Customization blocks are considered.

Tip

You can use multiple Autodoc Customization blocks with different tags in a subsystem. In your customization scripts, you can then list the tags in the `AutoDocFilter` property as required for specific customers/use cases. For example,
`'AutoDocFilter', {'Customer A','CommonInfo'}`

For a detailed description of the properties, refer to the documentation of the customization file.

Result

You have filtered the documentation generation using tags.

Related topics

References

TargetLink Main Dialog Block (📖 TargetLink Model Element Reference)

How to Specify Incremental Documentation Generation

Objective

As a part of the incremental code generation feature, you can also generate documentation for selected code generation units incrementally.

Preconditions

- The code generation units must be specified for incremental code generation. Refer to Basics on Incremental Code Generation (📖 TargetLink Customization and Optimization Guide).
- You derived a customization file (you should not use the `tldoc_default` script) as described in Basics on Customizing the Generated Documentation on page 29 and specified it on the Tools page of the TargetLink Main Dialog Block.

Method

To specify incremental documentation generation

1 In the MATLAB Command Window, type
`edit <name of your customization file>`.
The customization file is displayed in the editor.

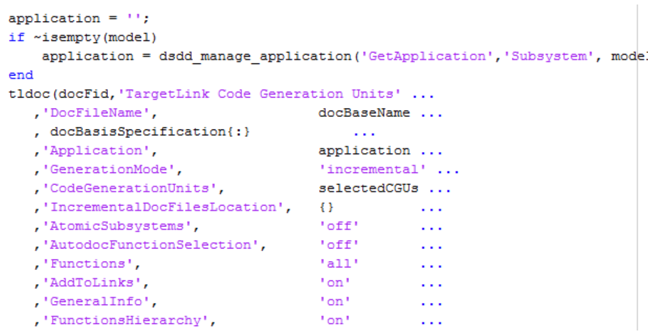
2 Scroll to the **`TargetLink Code Generation Units`** section, and set the `GenerationMode` property to `'incremental'`.

3. Specify the names of the code generation units to be documented incrementally in the `CodeGenerationUnits` property. Alternatively, if set to `'all'` (default), all code generation units in the specified Data Dictionary/model are considered.

   If you have specified code generation units for incremental documentation generation, the following rules apply:

   - If the specified code generation units contain nested incremental code generation units, an existing documentation is embedded into the generated one for them.
   - If such documentation does not exist yet, it is created.

4. Optionally, you can specify the path of the existing documentation file with the `IncrementalDocFilesLocation` property. By default, the empty cell array uses the following path: `<currentDocDirectory>\<cguName>\<cguName>_main.html`.



```
application = '';
if ~isempty(model)
    application = dsdd_manage_application('GetApplication','Subsystem', model);
end
tldoc(docFid,'TargetLink Code Generation Units' ...
   ,'DocFileName',                     docBaseName ...
   , docBasisSpecification{:}              ...
   ,'Application',                     application ...
   ,'GenerationMode',                  'incremental' ...
   ,'CodeGenerationUnits',             selectedCGUs ...
   ,'IncrementalDocFilesLocation',     {}          ...
   ,'AtomicSubsystems',                'off'       ...
   ,'AutodocFunctionSelection',        'off'       ...
   ,'Functions',                       'all'       ...
   ,'AddToLinks',                      'on'        ...
   ,'GeneralInfo',                     'on'        ...
   ,'FunctionsHierarchy',              'on'        ...
```

   For a description of all properties available, refer to the documentation in the script.

Result

You have specified incremental documentation generation for selected code generation units.

Related topics

References

TargetLink Main Dialog Block (📖 TargetLink Model Element Reference)

How to Customize the Appearance of Tables in the Documentation

Objective

Tables in the documentation are normally optimized for browser viewing. If you want a document that is optimized for printing, you may want to alter the column widths in tables. It is also possible to hide columns that are not required in the documentation.

Precondition

You derived the `tldoc_layout.m` file from its template.

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 (TargetLink Customization and Optimization Guide).

Method

To customize the appearance of tables in the documentation

1 At the MATLAB Command Window, type **`edit tldoc_layout`**.

2 Scroll to the function that contains the layout settings of the table that you want to customize.

For example, the table of function input signals is defined in the `i_GetFcnInputTable_<cfgName>` function. `cfgName` is the name of the configuration set specified via the `ConfigurationSet` property of the **`TargetLink Code Generation Units`** action.

3 Locate the column that you want to exclude.

Tip

You can identify a column via the value of the `fcnInputTable.column(x).content` property. For example, `fcnInputTable.column(1).content = 'TLDOC_VARIABLE'` represents the variable name column, which is the first column (`.column(1)`) in the table.

4 To exclude the column from the documentation, comment out the lines of code associated with the column (see the example below).

5 Alter the column numbers of the following columns so that they are consecutive.

If you want to exclude the second column (`column(2)`) in a table with 4 columns, you need to change the numbering for columns 3 to 4 as shown below:

```
fcnInputTable.column(1).content = 'TLDOC_VARIABLE';
fcnInputTable.column(1).format  = '%s';
fcnInputTable.column(1).width   = '40mm';
%fcnInputTable.column(2).content = 'TLDOC_TYPE';
%fcnInputTable.column(2).format  = '%s';
%fcnInputTable.column(2).width   = '15mm';
fcnInputTable.column(2).content = 'TLDOC_LSB';
fcnInputTable.column(2).format  = '%g';  % 2^%d for power of two
scaling
fcnInputTable.column(2).width   = '20mm';
fcnInputTable.column(3).content = 'TLDOC_OFFSET';
fcnInputTable.column(3).format  = '%g';
fcnInputTable.column(3).width   = '10mm';
```

6 To alter the width of a column, locate the desired column as described in step 3, and change the value of the `fcnInputTable.column(x).width` property.

> **Note**
>
> Changes to column widths apply only when the View Mode property in the appropriate tldoc_default chapter is set to Print. Otherwise the tables are optimized for browser viewing. For more detailed information, refer to tldoc (TargetLink API Reference).

7 Save and close the M file.

Result

You can now generate the documentation to confirm your changes.

Excluding tables from the documentation

It is also possible to hide complete tables that are not required in the documentation. To exclude a table from the documentation, append `<table>.column = [];` at the end of the function that contains the layout settings of the table that you want to hide, where `<table>` stands for the respective table, for example, `fcnInputTable` for the table of function input signals.

Related topics

Basics

Deriving Customization Files From Their Templates (TargetLink Customization and Optimization Guide)

HowTos

How to Generate the Documentation.. 27

References

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

How to Change the Documentation Language by Providing Custom Language Strings

Objective

When shipped, TargetLink supports documentation generation in German (`de`) or English (`en`). To provide custom languages, you have to provide custom language strings.

Precondition

You derived the `tldoc_layout.m` file from its template.

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 (📖 TargetLink Customization and Optimization Guide).

Method

To change the documentation language by providing custom language strings

1. At the MATLAB Command Window, type **`edit tldoc_layout`**.
2. Copy the contents of the default function `i_getstrings_uk` and paste it directly after the function.
3. Change `uk` in the function name to the desired language identifier, for example, `fr` for French.
4. Translate the strings in the new function accordingly.

   You now have to tell the document generation process to use the new language set.
5. Use the `tlCustomizationFiles` API function and select one of the customization files, for example, `tldoc_pdf_customization` and save it under a new name, for example `tldoc_pdf_french`.
6. Scroll to each of the actions **`TargetLink Code Generation Units`**, **`Overview`** and **`Simulation Results`**, and set the `Language` property to the language identifier (for example, search for `uk` and replace it by `fr`).
7. Save and close the M file.

Result

If you now generate the documentation, the text is written in the specified language.

Related topics

Basics

Deriving Customization Files From Their Templates (📖 TargetLink Customization and Optimization Guide)

HowTos

References

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

Using the TargetLink M-Script Interface (API)

TargetLink's API and MATLAB Command Window

When working with TargetLink, you normally use the TargetLink user interface to carry out tasks. However, you can also use TargetLink's API via the MATLAB Command Window to carry out tasks or to automate an entire process.

Where to go from here

Information in this section

Automating Tasks via TargetLink API

Command-line based application interface

The TargetLink API is a command-line based application interface which allows you to carry out or automate tasks.

Where to go from here

Information in this section

Basics on the TargetLink API

Supported tasks

Here are some examples of tasks that can be performed via TargetLink's command-line application interface:

- Access and modify properties of one or several TargetLink blocks or Stateflow objects
- Start TargetLink utilities or tools for model conversion, autoscaling, production code generation, and so on, without using TargetLink's graphical user interface

You can also write your own MATLAB scripts (M scripts) that contain API functions to automate repetitive tasks, such as production code generation.

Note

If you use TargetLink API functions via M script, you should always perform an error check. To do so, use the following commands at the end of the script:

```
if ds_error_check
   disp('There was an error');
end
```

For the automation of TargetLink via API, you can use the batch mode that suppresses dialogs interrupting the script processing. In batch mode, the default settings of the dialog windows are selected.

Online help

If you type `help <APIcommand>` in the MATLAB Command Window, you will get a concise description of the respective TargetLink or Data Dictionary API function. If you type `tl_online_help`, the start page of dSPACE Help opens.

Related topics

References

ds_error_get('BatchMode') (TargetLink API Reference)
ds_error_set (TargetLink API Reference)

Basics on Getting and Setting Block Properties

TargetLink blocks

TargetLink blocks are masked Simulink blocks whose mask variables contain scaling and other data necessary for simulation and for production code generation. For information on the block properties that can be accessed and changed via the TargetLink API, refer to The TargetLink Simulation Blocks and Their TargetLink Properties (TargetLink Model Element Reference).

Block properties

Block properties can be read and set via the block dialog, TargetLink's Property Manager, or API functions. Here only the API functions are dealt with. For more information on getting and setting block properties manually or via the Property Manager, refer to Modifying Block Properties via Block Dialogs (TargetLink Preparation and Simulation Guide) or Modifying Multiple Properties at Once via the Property Manager (TargetLink Preparation and Simulation Guide), respectively.

Accessing block properties

The `tl_get` and `tl_set` API functions provide access to the properties of the TargetLink blocks. First you need to get the block identifier. To do this you can use the `tl_get_blocks` and `tl_find` commands.

Getting and setting process

The typical process is as follows:

1. Get the TargetLink block identifier with `tl_get_blocks` or `tl_find`. Refer to How to Get TargetLink Block Identifiers on page 46.
2. Once the block identifier or the block name is known, you can process the block with `tl_set` and `tl_get`. Refer to How to Set Properties of TargetLink Blocks on page 48.

Example for Stateflow objects

A similar process applies to objects in Stateflow. For an example of this based on the `fuelsys` demo model, refer to Example of Getting and Setting Stateflow Object Properties on page 49.

Related topics

Basics

Modifying Block Properties via Block Dialogs (📖 TargetLink Preparation and Simulation Guide)
Modifying Multiple Properties at Once via the Property Manager (📖 TargetLink Preparation and Simulation Guide)

HowTos

Examples

References

tl_find (📖 TargetLink API Reference)
tl_get (📖 TargetLink API Reference)
tl_get_blocks (📖 TargetLink API Reference)
tl_set (📖 TargetLink API Reference)

How to Get TargetLink Block Identifiers

Objective

Before you can edit block properties, you need to know the block identifier (handle).

Block identifier

You can use the TargetLink API function `tl_get_blocks` for this purpose. This command returns the handles of the blocks that are located in a model. You can obtain the handles of all the TargetLink blocks in a model or a specific type of block.

Restrictions

The `tl_get_blocks` command works only for TargetLink blocks that originate from the TargetLink Blockset (tllib) or the TargetLink RTOS blockset (`tl_rtos_lib`).

Precondition

The model that you want to process is open.

Method

To get TargetLink block identifiers

1 In the MATLAB Command Window, type

```
hBlocks = tl_get_blocks(<system>)
```

Result

MATLAB returns a column vector of handles and stores the results in a workspace variable named `hBlocks`. You can now use this variable in the `tl_set` command to set the block properties.

Examples of getting handles

The following two examples show how to get the handles of all the TargetLink blocks or just specific blocks.

All blocks

If you want to get the handles of *all* the TargetLink blocks in the PIPT1 demo model, type

```
hBlocks = tl_get_blocks('pipt1','TargetLink')
```

MATLAB returns an array of handles like the one shown below:

```
hBlocks =
    16.0004
    17.0004
    18.0004
    ...
```

Specific blocks

If you want to get the handles of all the TargetLink *Gain* blocks in the PIPT1 demo model, type

```
hBlocks = tl_get_blocks('pipt1','Gain')
```

MATLAB returns an array of handles as shown below:

```
hBlocks =
    18.0004
    19.0004
```

Tip

You can also use the `tl_find` API function to get the handles of TargetLink blocks with specific properties.

Next steps

After you have finished these steps, you should proceed with How to Set Properties of TargetLink Blocks on page 48.

Related topics

HowTos

How to Set Properties of TargetLink Blocks........ 48

Examples

Example of Getting and Setting Stateflow Object Properties........ 49

References

tl_find (📖 TargetLink API Reference)

How to Set Properties of TargetLink Blocks

Objective

You can modify the properties of the TargetLink blocks with the `tl_set` command.

Preconditions

You have obtained the handles of the blocks whose properties you want to set and stored them in a workspace variable named `hBlocks`.

Method

To set properties of TargetLink blocks

1 In the MATLAB Command Window, type `tl_set(hBlocks,propertyName,propertyValue,...)`.

Result

The properties of the blocks specified by the `hBlocks` workspace variable are set to the specified values.

Example

You can set the variable class to GLOBAL and the data type to Int32 for all the Gain blocks in the PIPT1 demo model. If `hBlocks` is a workspace variable containing the handles of all the Gain blocks in the PIPT1 demo model, type

`tl_set(hBlocks,'gain.class','GLOBAL','gain.type','Int32')`.

MATLAB returns a vector of error flags that are zero for each element that is set successfully, as shown below:

```
ans =
   0
   0
```

Related topics

Basics

HowTos

Example of Getting and Setting Stateflow Object Properties

Accessing TargetLink data of Stateflow objects

Initially, Stateflow objects do not contain TargetLink data. TargetLink stores this information, such as variable classes or scaling factors, in the description string of the Stateflow objects. You can use either the Property Manager or the `tl_get` and `tl_set` API functions to access and modify this information. You can get the necessary Stateflow object handles via the `tl_get_sfobjects` API function. For details on these commands, refer to `tl_get`, `tl_set`, and `tl_get_sfobjects`. For information on modifying properties via the Property Manager, refer to Modifying Multiple Properties at Once via the Property Manager (TargetLink Preparation and Simulation Guide).

Example

Suppose you have a model called fuelsys and want to set the LSBs and data types of Stateflow variables with the TargetLink API.

Getting handles

To get the handles of the Stateflow constants, type `sfConstants = tl_get_sfobjects('fuelsys','SF Constant')`.

MATLAB returns an array of handles like the one shown below:

```
sfConstants =
    109
    110
    111
    …
```

Getting properties

There are a number of Stateflow objects in the model whose handles are now stored in `sfConstants`. If you want to check the current property values, carry out the following steps.

To get the Stateflow names, type `tl_get(sfConstants,'sfname')`.

MATLAB returns an array of Stateflow names as shown below:

```
ans =
    'DISABLED'
    'LOW'
    ’RICH’
    …
```

To get the TargetLink data types, type **`tl_get(sfConstants,'type')`**.

MATLAB returns an array of TargetLink data types as shown below:

```
ans =
    'UInt8'
    'UInt8'
    'UInt8'
    …
```

Modifying the scaling

In this case all the variables are floating-point and have to be scaled for implementation on a fixed-point controller. If you want to implement the o2_t_thresh variable as a UInt16 with LSB = 2^{-8}, you have to attach the scaling data. This is possible via the tl_set API function. To modify the scaling, type **`tl_set(sfConstants(1),'type','UInt16','LSB',2^-8)`**.

MATLAB returns an error flag that is zero to indicate that the properties were set successfully, as shown below:

```
ans =
    0
```

Confirming modifications

The TargetLink data type of the Stateflow o2_t_thresh variable is now set to UInt16, and the LSB value to 2^{-8}. To confirm that the LSB has changed, type **`tl_get(sfConstants(1),'LSB')`**.

MATLAB returns:

```
ans =
    0.0039
```

To confirm that the data type has changed, type **`tl_get(sfConstants(1),'type')`**.

MATLAB returns:

```
ans =
    UInt16
```

Related topics

Basics

Editing Property Values in the Property View (TargetLink Preparation and Simulation Guide)
Modifying Multiple Properties at Once via the Property Manager (TargetLink Preparation and Simulation Guide)

References

tl_get (TargetLink API Reference)
tl_get_sfobjects (TargetLink API Reference)
tl_set (TargetLink API Reference)

Automating Code Generation

Customizing the code generation process

TargetLink's API lets you customize the code generation process according to your requirements. Instead of manually performing all the code generation steps on the Code Generation page of the TargetLink Main Dialog, you can write M scripts to automate the process.

Where to go from here

Information in this section

Overview of the API Functions for Automatic Code Generation

List of API functions

TargetLink provides the following basic API functions to automate the code generation process:

API Function	Purpose
`tl_generate_code`	To generate code for the specified TargetLink subsystem(s).
`tl_compile_host`	To compile and link production code for the simulation S-function(s) and the host simulation application. In PIL simulation mode, this command is used to compile the S-function for communication with the target evaluation board (physical or virtual).
`tl_compile_target`	To compile code for the target simulation application.
`tl_download`	To download the target simulation application to the evaluation board via the specified download port.
`tl_set_sim_mode`	To switch the simulation mode of TargetLink subsystem(s). The possible simulation modes are: ▪ `TL_BLOCKS_HOST` (MIL simulation) ▪ `TL_CODE_HOST` (SIL simulation) ▪ `TL_CODE_TARGET` (PIL simulation)

The following commands call the basic commands described above:

API Function	Purpose
`tl_build_host` (TargetLink API Reference)	To generate and compile production code and to load the generated simulation application to the RAM of your development PC. This command calls `tl_generate_code`, `tl_compile_host`, and `tl_set_sim_mode`.

API Function	Purpose
`tl_build_target` (📖 TargetLink API Reference)	To generate and compile production code and to download the generated simulation application to the target evaluation board. This command calls all the basic commands listed above.

Related topics

Basics

Basics on Using Hook Scripts (📖 TargetLink Customization and Optimization Guide)

HowTos

References

Code Generation Hook Scripts (📖 TargetLink File Reference)
tl_build_host (📖 TargetLink API Reference)
tl_build_target (📖 TargetLink API 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_set_sim_mode (📖 TargetLink API Reference)

How to Generate Code via M Scripts

Objective

You can write your own M scripts which help to avoid errors during manual code generation.

Maintaining constraints

During development, it may be important to maintain the same constraints on the code generation process. If the code generation process is performed manually via the user interface, human errors, for example, forgetting to select a certain setting, may lead to different results. To avoid this, you can write an M script for each of the required code generation processes. Each time the M script is run, the same constraints are kept.

To further customize the M script, refer to Basics on Using Hook Scripts (📖 TargetLink Customization and Optimization Guide).

Preconditions

MATLAB is open, and the path is set to that of your model.

Method

To generate code via an M script

1 To create an M script, type **edit** in the MATLAB Command Window.

The MATLAB Editor opens.

2 In the MATLAB Editor, choose **Save as** from the **File** menu.

3 Enter a suitable file name, and then click **Save**.

The M script is saved in the same folder as your model. You can now write the commands for automating the production code generation process.

> **Tip**
>
> If you have more than one M script for controlling the production code generation process, it makes sense to give the M script a file name that identifies the process, for example, `pipt1_CodeGen_Standard.m`.

4 In the MATLAB Editor, type

```
open_system('<model_name>')
```

This command opens the specified model.

5 In the second line of code, use the tl_set command to set the code generation options. For example, type

```
tl_set('<model_name>','codeopt.cleancode',1,...
'codeopt.target','TOM C16x/Task86')
```

This command selects the clean code option and sets target optimization for the Infineon C16x with Tasking 8 compiler.

6 To generate code for all the TargetLink subsystems in the current model, type `tl_generate_code`.

You can also generate code for specific TargetLink subsystems only:

```
tl_generate_code('Model','<model_name>', ...
'TlSubsystems',{'tlSubsys1', ...,'TlSubsysN'});
```

7 To add a check for errors, type

```
if ds_error_check
   disp('There was an error');
end
```

For the PIPT1 demo model the script looks as follows:

```
open_system('pipt1')
tl_set('pipt1','codeopt.cleancode',1,...
   'codeopt.target','TOM C16x/Task86')
tl_generate_code
if ds_error_check
   disp('There was an error');
end
```

8 Save the script, and close the MATLAB Editor.

9 To run the M script and generate code, type the file name of the M script in the MATLAB Command Window, for example:

`pipt1_CodeGen_Standard`
The M script runs and generates code for the PIPT1 model.

Result	Production code and the associated files are stored in the same folder as your model. You can run the M script, or variants of it, from the MATLAB Command Window whenever you want to generate production code for your model.
Next steps	If you want to further customize the M script by embedding your own commands at specific points of the code generation, compilation and download process, refer to Basics on Using Hook Scripts (📖 TargetLink Customization and Optimization Guide).

Related topics

Basics

Automating Code Generation.. 52
Basics on Using Hook Scripts (📖 TargetLink Customization and Optimization Guide)

References

Code Generation Hook Scripts (📖 TargetLink File Reference)
ds_error_check (📖 TargetLink API Reference)
tl_generate_code (📖 TargetLink API Reference)
tl_set (📖 TargetLink API Reference)

Obtaining Model and Code Information from the TargetLink Data Dictionary

Model and code information

You may want to find specific information on your model, TargetLink subsystem or the generated production code. This information is stored in the TargetLink Data Dictionary.

Where to go from here

Information in this section

Basics on Information Stored in the TargetLink Data Dictionary

Model and code information

The TargetLink Data Dictionary enables you to find specific information on your model or the generated production code.

Accessing the TargetLink Data Dictionary

The TargetLink Data Dictionary (DD) is a central data container that holds all relevant information for code generation and calibration. You can access the information via the Data Dictionary Manager or via the TargetLink Data Dictionary MATLAB API. For more information on the TargetLink Data Dictionary, refer to Basics on the TargetLink Data Dictionary (📖 TargetLink Data Dictionary Basic Concepts Guide). Detailed information on the API is available in the 📖 TargetLink Data Dictionary Reference.

TargetLink Data Dictionary Structure

The TargetLink Data Dictionary has 4 main areas for the setting of configuration data, pool data for models (pre-code generation data), subsystem and application data (post-code generation data). For more detailed information on the 4 areas, refer to Basics on the Data Dictionary Structure (DD Object Tree) (📖 TargetLink Data Dictionary Basic Concepts Guide).

Online Help

For more information on the syntax of commands and the data model object, you can type `dsdd help <command/object>` in the MATLAB Command Window. MATLAB opens dSPACE Help at the desired place.

To understand how to use the MATLAB API to access information in the TargetLink Data Dictionary, refer to the following instructions. These represent example tasks that can be performed via API.

- How to Obtain a List of Generated C Modules for a Subsystem on page 57
- How to Obtain Version Information on Generated C Modules on page 58
- How to Obtain a List of Calibratable Variables in an Application on page 59

Related topics

Basics

Basics on the Data Dictionary Structure (DD Object Tree) (📖 TargetLink Data Dictionary Basic Concepts Guide)
Basics on the TargetLink Data Dictionary (📖 TargetLink Data Dictionary Basic Concepts Guide)

HowTos

How to Obtain a List of Generated C Modules for a Subsystem

Objective

If you want to automate a task that needs performing on all the generated C modules for a subsystem, you need to obtain a list of them first.

Data model objects

Since the C modules are child objects of a Subsystem object, you can use the generic **GetChildren** command, which allows you to specify the object identifier to be found.

Precondition

Code has been generated for the model.

Method

To obtain a list of generated C modules for a subsystem

1 At the MATLAB Command Window, type

```
hModules=dsdd('GetChildren',...
   '/Subsystems/<subsystem>','objectKind','module')
```

where <subsystem> represents the subsystem which the production code was generated for.

On executing this command, MATLAB returns the handles of all the C modules in the subsystem.

Result

The C module handles are now stored for reuse in the `hModules` workspace variable.

Example

The following example shows the command for the `picontroller` subsystem in the `PIPT1` demo model.

```
hModules=dsdd('GetChildren',...
   '/Subsystems/picontroller','objectKind','module')
```

MATLAB returns information similar to this:

```
hModules =
   596    21   556   714   654
```

There are five C modules in `PIPT1`.

Tip

You could also use the `Find` command to achieve the same result.

```
hModules=dsdd('Find',...
    '/Subsystems/<subsystem>','objectKind','module')
```

Next step

You can now use the **`hModules`** workspace variable to obtain information on a specific C module. Refer to the example in How to Obtain Version Information on Generated C Modules on page 58.

Related topics

HowTos

How to Obtain Version Information on Generated C Modules

Objective

The code generation process stores a variety of information on each C module, for example, a short description, the name of the person or the tool who created the module.

Module information

The module information is stored in the `ModuleInfo` object, which is a child of the `Module` object. The tool version information is stored in the `CreatorVersion` property. You can use the `GetModuleInfoCreatorVersion` function to retrieve this property.

Precondition

Code has been generated for the model.

Method

To obtain version information on a generated C module

1 At the MATLAB Command Window, type

```
dsdd('GetModuleInfoCreatorVersion',<objectidentifier>)
```

where `<objectidentifier>` represents the desired object, in this case a C module.

Result

MATLAB returns the version information on the specified C module.

Example

The following example uses the `hModules` workspace variable created in the previous instructions under How to Obtain a List of Generated C Modules for a Subsystem on page 57.

```
dsdd('GetModuleInfoCreatorVersion',hModules(2))
```

MATLAB returns information similar to this:

```
ans =TargetLink 5.1
```

This is the version information on the tool that generated the second module in `hModules`.

Related topics

HowTos

How to Obtain a List of Generated C Modules for a Subsystem.. 57

How to Obtain a List of Calibratable Variables in an Application

Objective

To perform a task on a specific set of variables, you can create a list of variables of a specific variable class, for example, calibratable variables.

Obtaining lists

As an application can contain more than one subsystem, you have to obtain a list of subsystems in the application first. You can use this list to produce a list of subsystem variables, which you can then filter for calibratable variables.

Data model objects

Applications are located at the root level of the TargetLink Data Dictionary and represented by `Application` objects. An application contains `SubsystemConfig` objects, which are children of the `Config` object. Each

`SubsystemConfig` object describes one `Subsystem` object. The variables in an application are contained in the `Subsystem` object.

Precondition

Code has been generated for the model.

Method

To obtain a list of calibratable variables in an application

1 At the MATLAB Command Window, type:

```
hSubsystemConfig=dsdd('GetChildren',...
   '/<application>/Config','objectkind','SubsystemConfig')
```

where <application> represents the root level name of the application.

On executing this command, MATLAB returns the handles of all the subsystems in the application. With this information you can now find the names of the subsystems by using the generic **`GetAttribute`** command.

2 At the MATLAB Command Window, type:

```
SubsystemNames=cell(size(hSubsystemConfig));
for i=1:length(hSubsystemConfig),
   SubsystemNames{i}=dsdd('GetAttribute',...
      hSubsystemConfig(i),'Name');
end
```

MATLAB returns an array of subsystem names. To reduce the number of variables found, the following instructions assume that you want to search only one subsystem. You now need to obtain a list of all the variables and count the variables in the array.

3 At the MATLAB Command Window, type:

```
hVariables=dsdd('Find',...
   '/Subsystems/<subsystem>','objectkind','variable');
```

where <subsystem> represents one of the subsystems found in the previous step.

MATLAB returns the variables in the subsystem. Now you need to filter all the calibratable variables. You can do this by checking whether each variable class is readwrite. For this purpose you need to use the **`GetClassTarget`** and **`GetInfo`** functions, which are functions of the `Variable` and `VariableClass` objects, respectively.

4 At the MATLAB Command Window, type:

```
for i=1:length(hVariables)
   % get associated VariableClass
   hClass=dsdd('GetClassTarget',hVariables(i));
   % get the info property - is it "readwrite" ?
   if strcmp(dsdd('GetInfo',hClass),'readwrite') == 0,
      % if no - mark the handle vector element
      hVariables(i)=-1;
   end
end
```

The handles of all the variables in `hVariables` that are not calibratable are set -1. You can now filter the list of variables to show the handles of the calibratable variables.

5 At the MATLAB Command Window, type

```
hCalVariables=hVariables(find(hVariables~=-1))
```

MATLAB returns the handles of all the calibratable variables in the chosen subsystem.

Result

You could now further process the calibratable variables stored in `hCalVariables`, for example, by changing the variable class.

Example

The following example uses the `PIPT1` demo model to demonstrate how to obtain a list of all the calibratable variables.

```
hSubsystemConfig=dsdd('GetChildren',...
   '/pipt1/Config','objectkind','SubsystemConfig')
```

MATLAB returns information similar to this:

```
hSubsystemConfig =
    87
```

where 87 is the handle of the subsystem. The `PIPT1` demo model has only one subsystem. Generally, your application will have more than one subsystem. Now you need to find the subsystem's name and store it in `SubsystemNames`.

```
SubsystemNames=cell(size(hSubsystemConfig));
for i=1:length(hSubsystemConfig),
   SubsystemNames{i}=dsdd('GetAttribute',...
      hSubsystemConfig(i),'Name');
end
```

You can use the index of the `SubsystemNames` array to refer to the name.

```
         SubsystemNames(1)
```

MATLAB returns:

```
ans =
    'picontroller'
```

You can now search for all the variables in the subsystem.

```
hVariables=dsdd('Find',...
   '/Subsystems/picontroller','objectkind','variable');
size(hVariables)
```

MATLAB returns:

```
ans =
     1    12
```

There are 12 variables in the picontroller subsystem.

```
for i=1:length(hVariables)
   % get associated VarableClass
   hClass=dsdd('GetClassTarget',hVariables(i));
   % get the info property - is it "readwrite" ?
   if strcmp(dsdd('GetInfo',hClass),'readwrite') == 0,
      % if no - mark the handle vector element
      hVariables(i)=-1;
   end
end
```

The handles of the variables that are not calibratable (readwrite) are set to -1.

```
hCalVariables=hVariables(find(hVariables~=-1))
```

MATLAB returns information similar to this:

```
hCalVariables =
   135   133
```

The handles of the calibratable variables in the picontroller subsystem are 135 and 133.

Interoperating with Other Tools

Where to go from here

Information in this section

Interoperating with Other dSPACE Tools for Virtual Validation

Introduction

TargetLink is part of dSPACE's tool chain for virtual validation.

Where to go from here

Information in this section

Basics on Interoperating with Other dSPACE Tools for Virtual Validation

Virtual validation

TargetLink is part of the dSPACE tool chain for virtual validation. TargetLink lets you generate V-ECU implementations (`*.vecu` file) from its subsystems.

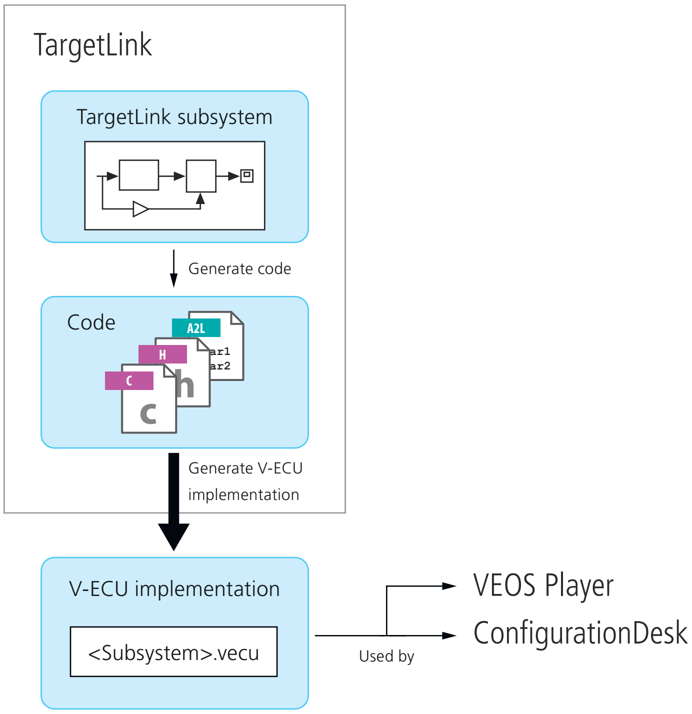
Virtual validation is used for *simulation systems* that comprise executable representations of ECUs (V-ECUs) as well as a representation of their environment and all their interconnections. Simulation systems can be executed on dSPACE simulation platforms. Two simulation platforms are available:

- VEOS for *offline* simulation
- SCALEXIO and MABX III for *real-time* simulation

For details on virtual validation, refer to Virtual Validation Overview.

Generating V-ECU implementations

TargetLink can generate and export V-ECU implementations to be integrated into a simulation system. This consists of two steps:



You can import the generated V-ECU implementation to the VEOS Player and/or to ConfigurationDesk and generate an executable application for it:

- An offline simulation application (OSA) (OSA file) for offline simulation on VEOS.
- A real-time application (RTA file) for real-time simulation on SCALEXIO or MABX III.

Handling V-ECU implementations

TargetLink provides two ways to handle V-ECU implementations:

- TargetLink V-ECU Manager dialog
- `tl_generate_vecu_implementation` API function

Embedding external code

You can embed external code in your subsystem to use it in V-ECU implementations. Refer to Embedding External Code (TargetLink Customization and Optimization Guide).

Limitations

For details on limitations concerning V-ECU implementation generation, refer to V-ECU Implementation Generation Limitations (TargetLink Limitation Reference).

Related topics

References

TargetLink V-ECU Manager (📖 TargetLink Tool and Utility Reference)
tl_generate_vecu_implementation (📖 TargetLink API Reference)

How to Generate V-ECU Implementations

Objective

To generate a V-ECU implementation (`*.vecu` file) for integration with ConfigurationDesk (for real-time simulation) or with VEOS Player (for offline simulation), using the TargetLink V-ECU Manager.

Precondition

The model from which you want to generate a V-ECU implementation must be open.

Method

To generate V-ECU implementations

1. On the Tools page of the TargetLink Main Dialog, click Export V-ECU Implementation.

   The TargetLink TargetLink V-ECU Manager opens.

2. In the Code generation units for V-ECU implementation group box, select the TargetLink subsystem for which you want to generate a V-ECU implementation.

   If required, select one or more DD CodeGenerationUnit objects or select Include subitems for code generation.

   If you select Include subitems for code generation, TargetLink generates production code for nested code generation units.

3. In the Configuration for V-ECU implementation generation group box, specify the destination directory for the generated V-ECU implementation.

4. If you need definitions of external global interface variables in the code, because the source file with the variable definitions is not included, select the Create definitions for external global interface variables checkbox.

5. Unless you already generated code or want to generate code again, perform the following step: From the Export V-ECU Implementation menu, select Generate Code.

   Wait for the code generation process to finish.

6. From the Export V-ECU Implementation menu, select Generate V-ECU implementation.

   TargetLink generates the V-ECU implementation file and copies it to the destination path.

Result

You generated a V-ECU implementation file.

Related topics

Basics

Integrating the Simulation System for SCALEXIO or a MicroAutoBox III (📖 Virtual Validation Overview)
Integrating the Simulation System with the VEOS Player (📖 Virtual Validation Overview)

References

TargetLink V-ECU Manager (📖 TargetLink Tool and Utility Reference)
tl_generate_code (📖 TargetLink API Reference)
tl_generate_vecu_implementation (📖 TargetLink API Reference)

Interoperating with FMI-Compliant Tools via FMUs

Introduction

TargetLink lets you export Functional Mock-up Units (FMUs) as described by the FMI standard.

This lets you use TargetLink production code with FMI-compliant tools.

Where to go from here

Information in this section

Definition of the FMI Standard and FMUs

Functional Mock-up Interface

Functional Mock-up Interface (FMI) is a tool-independent standard that supports both the co-simulation and the exchange of dynamic models by using archive files containing a combination of XML files and C functions provided in source and/or binary form. The FMI standard defines the interface to be implemented by a Functional Mock-up Unit (FMU).

FMI for Co-Simulation interface The FMI for Co-Simulation interface is designed both for coupling simulation tools (simulator coupling, tool coupling) and for coupling subsystem models that were exported by their simulators and their solvers as runnable code.

FMI for Model Exchange interface The FMI for Model Exchange interface defines an interface to the model of a dynamic system described by differential, algebraic, and discrete-time equations. Via the FMI for Model Exchange interface, these equations can be evaluated as needed in different simulation environments.

Note

TargetLink supports only the FMI for Co-Simulation interface, but not the FMI for Model Exchange interface. For detailed and up-to-date compatibility information on FMI support in TargetLink, refer to the following website: http://www.dspace.com/go/FMI-Compatibility.

Functional Mock-up Unit

A Functional Mock-up Unit (FMU) describes and implements the functionality of a model. It is an archive file with the file name extension `.fmu`. The FMU file contains:

- The functionality defined as a set of C functions provided either in source or in binary form
- The model description file (`modelDescription.xml`) with the description of the interface data
- Additional resources needed for simulation

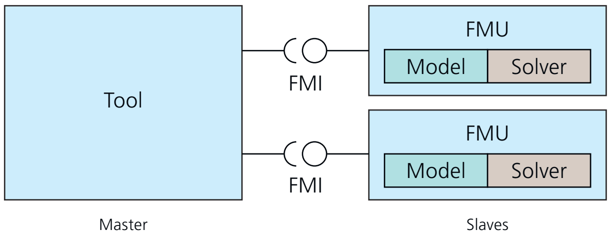
Note

For limitations that apply to TargetLink's FMU support, refer to FMU Generation Limitations (TargetLink Limitation Reference).

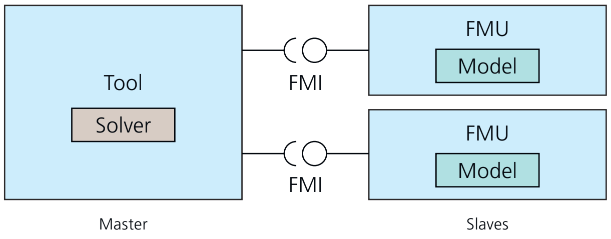
Correlation between the FMI standard and FMUs

The following illustrations show how FMUs correlate with the FMI standard.

FMI for Co-Simulation interface:



FMI for Model Exchange interface:



Related topics

Basics

Basics on Exporting FMUs from TargetLink........ 70

Basics on Exporting FMUs from TargetLink

Functional Mock-up Interface

Functional Mock-up Interface (FMI) is a tool-independent standard that supports both the co-simulation and the exchange of dynamic models by using archive files containing a combination of XML files and C functions provided in source and/or binary form. The FMI standard defines the interface to be implemented by a Functional Mock-up Unit (FMU).

Exporting FMUs

By default, the source code of generated FMUs is compatible with the following platforms:

- If system files are not included: all 32-bit and 64-bit platforms (QNX, Windows and Linux)
- If system files are included: all 32-bit and 64-bit platforms (QNX, Windows and Linux) with little endian byte order

The following options are available when exporting FMUs from TargetLink:

- You can specify if the source code, binaries or both are included in the FMU.

  If specified, TargetLink compiles a binary for the following platforms:

  - Windows 32-bit
  - Windows 64-bit
  - Linux 64-bit

  Note

  For the compilation of the Linux 64-bit binary, you must provide a suitable compiler. Refer to Topic Settings (TargetLink Tool and Utility Reference).

- You can specify if system files, such as the fixed-point library, are included in the FMU.

Note

If you want to generate an FMU for another target, use the **`tl_generate_fmu`** API function and specify its `SimConfigPackageDir` property as required and perform the build process outside of TargetLink.

API - tl_generate_fmu

In contrast to the dialog, the **`tl_generate_fmu`** API function lets you specify a simulation configuration package for a target-compiler combination for which you want to generate an FMU. This can be done via the `SimConfigPackageDir` property.

TargetLink ensures that the platform-specific files, such as the `tl_basetypes.h` header file, that are included in the container are suitable for the target platform.

A binary can only be built for the following platforms: Win32, Win64 and Linux64.

FMUs exported by TargetLink

The FMUs exported by TargetLink fulfill the following conditions:

- They comply with the FMI 2.0 standard for co-simulation.
- They have a fixed step size.
- They have a a default start time equal to `0.0`.
- They do not define units.
- Their interface variables (inputs, outputs, parameters, and log variables of the FMU's root function) are of the `fmi2Real` data type.

  All TargetLink data types and access functions can be used at the TargetLink root step function interface. TargetLink provides wrapper code for the simulation to ensure FMI compliance.

SIL simulation behavior

To ensure consistent behavior across simulations in SIL simulation mode, all the model's variables have to be reset to their initialization values before each simulation run. Two approaches are possible:

Approach 1: restart function initialization The FMI standard requires this to be done by executing the `fmi2Reset` FMI API function.

To comply with this standard, initialize all your state variables in one or more restart functions.

TargetLink generates a C file named `FMU<SubsystemName>` that defines the `fmi2Reset` function as follows:

```
/* BEGIN: fmi2Reset */
fmi2Status fmi2Reset(fmi2Component c)
{
tlFmuDoInit();
   return fmi2OK;
}
/* END: fmi2Reset */
```

The `tlFmuDoInit()` function calls the restart functions belonging to the TargetLink subsystem. If no restart function exists, `tlFmuDoInit()` is empty.

Note

This implies that all states are initialized in restart functions.
If you do not want to change your production code in this way, refer to approach 2.

Approach 2: reloading the application after each simulation run If you do not want to change your production code to ensure FMI compliance, resetting your states to their initial value can be achieved by reloading the application to the simulation platform before each simulation run. This ensures that the application's startup code is run, and your states are initialized properly.

Limitations

For limitations on exporting an FMU with TargetLink, refer to FMU Generation Limitations (📖 TargetLink Limitation Reference).

Related topics

Basics

HowTos

Examples

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

References

FMU Generation Limitations (📖 TargetLink Limitation Reference)
TargetLink FMU Manager (📖 TargetLink Tool and Utility Reference)
tl_generate_fmu (📖 TargetLink API Reference)

How to Generate an FMU to use in an FMI-Compliant Tool

Objective

To generate a Functional Mock-up Unit (FMU) from a TargetLink subsystem to use production code in an FMI-compliant tool.

Restriction

By default, the **Export FMU** dialog on the **Tools** page of the TargetLink Main Dialog Block generates an FMU that is compatible with the following targets:

- If system files are not included: all 32-bit and 64-bit platforms (QNX, Windows and Linux)
- If system files are included: all 32-bit and 64-bit platforms (QNX, Windows and Linux) with little endian byte order

If you want to generate an FMU for another target, you have to generate the FMU via the API and specify a simulation configuration package via the `SimConfigPackageDir` property.

Precondition

The model containing the TargetLink subsystem for which you want to generate an FMU is open.

Method

To generate an FMU to use in an FMI-compliant tool

1. On the Tools page of the TargetLink Main Dialog, click Export FMU.
   The TargetLink FMU Manager dialog opens.
2. Select the TargetLink subsystem for which you want to generate an FMU.
3. Configure the FMU generation as required.
4. Click Export FMU.
   TargetLink generates code and exports an FMU to the destination directory.

Result

You generated an FMU from a TargetLink subsystem.

Related topics

Basics

Basics on Exporting FMUs from TargetLink.. 70
Definition of the FMI Standard and FMUs.. 68

References

FMU Generation Limitations (📖 TargetLink Limitation Reference)
TargetLink FMU Manager (📖 TargetLink Tool and Utility Reference)
TargetLink Main Dialog Block (📖 TargetLink Model Element Reference)
tl_generate_fmu (📖 TargetLink API Reference)

Interoperating with dSPACE ECU Interface Software for On-Target Bypassing

Where to go from here

Information in this section

Basics on Modeling with RTI Bypass Blocks and Generating Code for On-Target Bypassing

On-target bypassing with TargetLink

In TargetLink, you can use the RTI Bypass Blockset for modeling. The RTI Bypass Blockset can be used to configure ECU interfaces and implement new ECU functions for bypassing or for tests. You can then use the TargetLink Code Generator for code generation to add or replace control function directly on the target hardware (on-target bypassing). This is done by using the free resources on the target ECU. On-target bypassing is also often called on-target prototyping, because you can develop and replace ECU functions without the use of additional (external) hardware.

Dynamically integrating a TargetLink model into a V-ECU You can also use the RTI Bypass Blockset to dynamically integrate a TargetLink model into a V-ECU executed in a VEOS offline simulation and SCALEXIO or MicroAutoBox III real-time application. Refer to:

- Dynamically Integrating a Simulink Model into a V-ECU Running on VEOS (📖 Virtual Validation Overview)
- Dynamically Integrating a Simulink Model into a V-ECU Running on a SCALEXIO System or on a MicroAutoBox III (📖 Virtual Validation Overview)

On-target bypassing at a glance

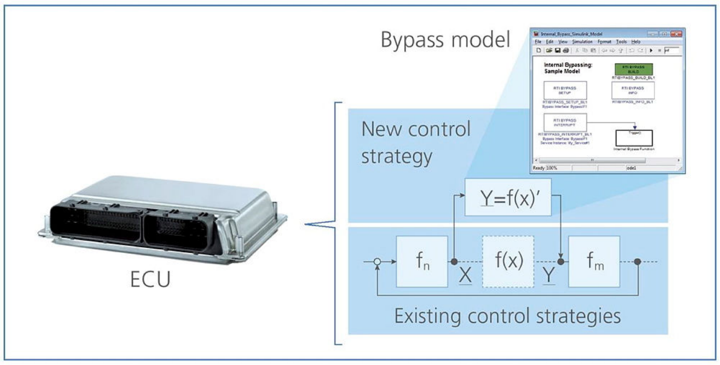
On-target bypassing is an effective way to develop ECU control functions under the following conditions:

- The ECU's available computational power, RAM and flash memory is sufficient.
- No additional sensor signals are required.

The following benefits apply if you develop and replace ECU functions directly on the target hardware:

- Reduced development costs because no additional hardware is needed.
- The input and output values do not have to be exchanged via interface hardware between the ECU and a prototyping hardware, i.e., there are no additional latencies.
- Improved safety aspects if the ECU hardware is already cleared for use.
- Because the function is already executed on the production hardware during development, it is guaranteed to comply with the resource limitations under production conditions.

The following illustration shows how bypassing is used to develop individual parts of the ECU software, e.g., a single control function, directly on the ECU:



Use cases and target audience

The following use cases apply when modeling with the RTI Bypass Blockset in TargetLink for on-target bypassing:

- You want to model with TargetLink blocks and RTI bypass blocks and generate bypass code.
- You want to model with TargetLink blocks and RTI bypass blocks and want to reuse the original A2L variables in the new control function: You have to define measurement and calibration variables from the ECU A2L files in the Data Dictionary.
- You want to model with Simulink® blocks and RTI bypass blocks and want to reuse the original A2L variables. You must specify them via Simulink.Signals and Simulink.Parameters in the new control function: The A2L variables for reuse are automatically created in the TargetLink Data Dictionary.

Target audience for on-target bypassing:

- TargetLink users who want to use existing or newly developed TargetLink models for prototyping.
- Function developers who want to perform prototyping on the real ECU.

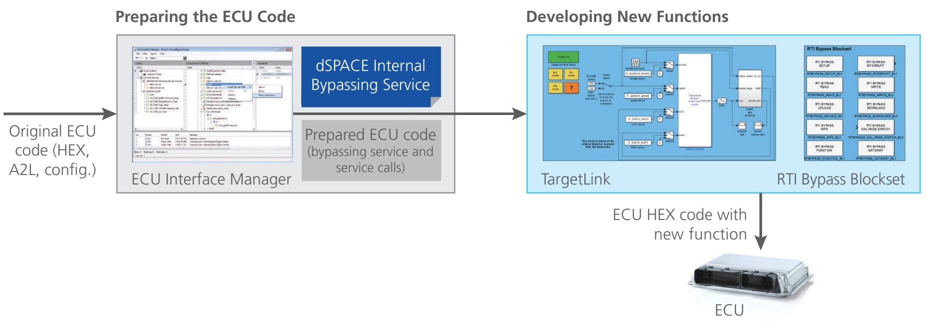
Tool chain overview

In combination with the RTI Bypass Blockset and a custom or a dSPACE flash tool, TargetLink implements new functions on the ECU (binary/hex code).

The following illustration shows the basic concepts of on-target bypassing:

1. You have to prepare the ECU software only once by integrating the internal bypassing service and the service calls.

2. You can model the bypass function by using the RTI Bypass Blockset with TargetLink and configure the interface to the ECU application.


Preparing the ECU Code
Developing New Functions
Original ECU code (HEX, A2L, config.)
dSPACE Internal Bypassing Service
Prepared ECU code (bypassing service and service calls)
ECU Interface Manager
TargetLink
RTI Bypass Blockset
ECU HEX code with new function
ECU

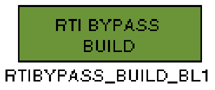
For an overview of the ECU interfacing tool chain, refer to 🕮 ECU Interfacing Overview.

The following table lists the relevant user documentation of the products involved in the tool chain:

Tool	Task	Further documentation
ECU Interface Manager	To add the dSPACE Internal Bypassing Service and the required interfaces to the ECU hex code as preparation for on-target bypassing.	Introduction to the ECU Interface Manager (🕮 ECU Interface Manager Manual).
dSPACE Internal Bypassing Service	Driver software integrated by ECU source code modification or ECU binary code modification via the ECU Interface Manager providing service points.	Introduction to the dSPACE Internal Bypassing Service Implementation (🕮 dSPACE Internal Bypassing Service Implementation).
TargetLink	To model new functions and automatically generate production code (C code) directly from the MATLAB®/Simulink®/Stateflow® development environment.	Also refer to Limitations When Using TargetLink with the RTI Bypass Blockset on page 81.
RTI Bypass Blockset	To easily connect new functions with existing ECU software.	General Information on the RTI Bypass Blockset (🕮 RTI Bypass Blockset Reference).
Target compiler (third- party tool)	To transfer C code to object code for the processor families Infineon TriCore™, Renesas V850™, and NXP MPC5xxx (formerly Freescale).	For details, refer to General Information on the RTI Bypass Blockset (🕮 RTI Bypass Blockset Reference) and to the corresponding third party documentation.

Starting code generation with the RTI BYPASS BUILD block

To start TargetLink code generation for the bypassing function by using the RTI BYPASS BUILD block.



If you use TargetLink-specific methods, for example, via the TargetLink Main Dialog, the build process aborts with an error.

For a description of the RTI BYPASS BUILD block, refer to RTI BYPASS BUILD Block (🕮 RTI Bypass Blockset Reference).

Using a TargetLink model with the RTI Bypass Blockset for code generation

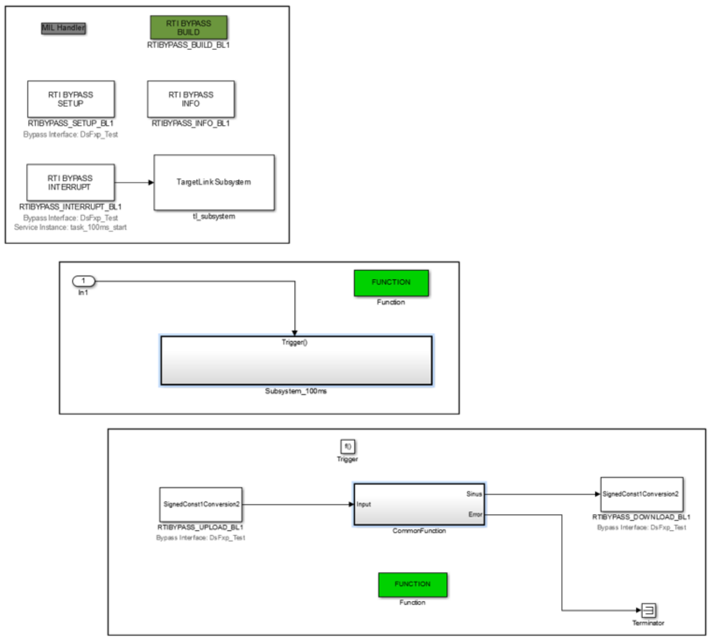
TargetLink lets you reuse ECU-software internal-measurement and calibration variables from the existing ECU A2L files. To do this, you must define Variable objects in the `/Pool/Variables` tree of the TargetLink Data Dictionary. The variable class must be either `BYPASSING_EXTERN_CAL` or `BYPASSING_EXTERN_DISP`, depending on whether you want to calibrate or measure signals. The variables must have exactly the same name, data type, LSB, and offset as in the original ECU A2L file variable. Otherwise, the build process aborts with an error.

Note

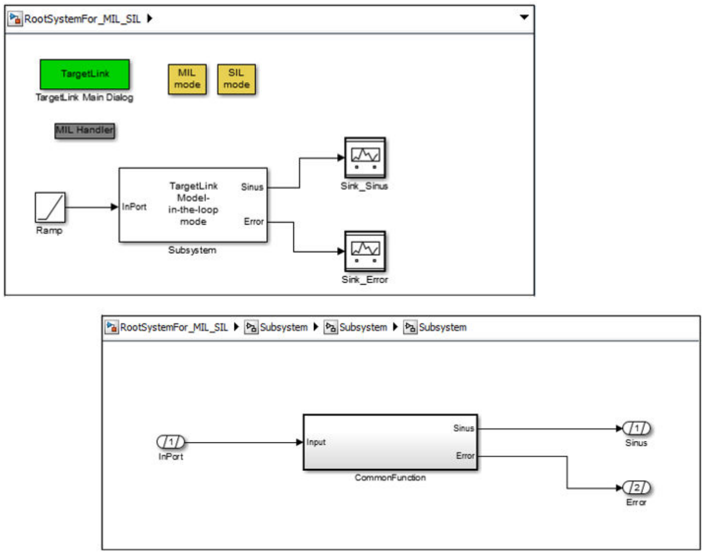
- For a list of the supported RTI bypass blocks, refer to Supported blocks of the RTI Bypass Blockset.
- With certain RTI bypass blocks, MIL or SIL/PIL simulation is not possible. However, a modeling guideline is available to help you to perform MIL/SIL/PIL simulations anyway. Refer to Modeling guideline to perform simulations.
- If you want to use TargetLink models with RTI bypass blocks, some limitations apply. Refer to Limitations When Using TargetLink with the RTI Bypass Blockset on page 81.

Modeling guideline to perform simulations

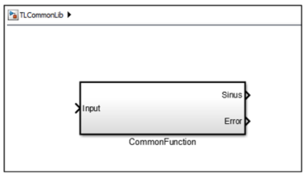
If you use the RTI Bypass Blockset in TargetLink, the sources for input and output signals are always RTI Bypass Read, RTI Bypass Write, RTI Bypass Upload, or RTI Bypass Download blocks. The blocks must be placed in the TargetLink subsystem. With these RTI blocks, MIL or SIL/PIL simulation is not possible. However, the following modeling guideline helps you perform MIL/SIL/PIL simulations anyway. Refer to the following example:



The TargetLink model for the simulation looks like this:



The CommonFunction subsystem is identical in both models, and is a library in this example.



Model referencing

If you start the RTI Bypassing build via an RTI BYPASS BUILD block, additional properties are written to the root model (`Hardware Implementation`). This might lead to a mismatch of the root model and the model-referencing subsystem and, as a consequence, to a Simulink initialization error. You can prevent this by using the `tl_refmodel_to_subsystem.m` API function before calling the RTI Bypassing build, which removes the model referencing.

If you have to make additional changes to the CommonFunction subsystem, you must use `tl_subsystem_to_refmodel.m` to create a referenced model, which can then be used for MIL/SIL/PIL simulations.

Removing TargetLink simulation frames

Optionally, you can remove TargetLink simulation frames to omit intermediate levels of the model (the frame is required only for SIL/PIL simulation, which is not used here). This can improve the handling of the model. Refer to

- How to Remove TargetLink Simulation Frames (TargetLink Main Dialog) (📖 TargetLink Preparation and Simulation Guide)
- How to Remove TargetLink Simulation Frames (API command) (📖 TargetLink Blockset Guide)

Using a Simulink model with the RTI Bypass Blockset for code generation

A pure Simulink model is automatically prepared for TargetLink by using Simulink signals and parameters for calibration and measurement. The TargetLink System preparation tool runs in the background. Refer to Basics on Preparing Simulink Systems for TargetLink Code Generation (📖 TargetLink Preparation and Simulation Guide).

You can reuse the original variables from the A2L file. They are automatically created in the `/Pool/Variables/Tools/Bypassing` tree. The class is either `BYPASSING_EXTERN_CAL` or `BYPASSING_EXTERN_DISP`, depending on whether you want to calibrate or measure signals. Those variables are automatically created if you select the Map Simulink Signals to ECU internal variables and the Map Simulink Parameters to ECU internal variables checkboxes in an RTI Bypass Setup block. For details, refer to Build Page (RTIBYPASS_SETUP_BLx for INTERNAL) (📖 RTI Bypass Blockset Reference).

Variable creation rules

CAL variables After model preparation, TargetLink searches for the `ExportedGlobal` Simulink property in TargetLink subsystems. Refer to the following variable creation rules:

- If the Simulink parameter is specified in the original target A2L file, the variable description (Type, Scaling, and Offset) is imported into the Data Dictionary and a VariableRef to the DD Variable object is set in the source block.
- If the Simulink parameter is a new calibration variable, the blocks' variable class is set to BYPASSING_CAL and the blocks' variable name is set to the Simulink parameter name.

DISP variables After model preparation, TargetLink searches for the `ExportedGlobal` Simulink property in TargetLink subsystems. If the source block of the signal line is a TargetLink block that lets you specify an output variable, the following rules apply:

- If the Simulink signal is specified in the original target A2L file as a measurement variable, the variable description (Type, Scaling, and Offset) is imported into the Data Dictionary and a VariableRef to the DD Variable object is set in the source block.
- If the Simulink parameter is a new measurement variable, the blocks' variable class is set to BYPASSING_DISP and the blocks' variable name is set to the Simulink signal name.

Also refer to Limitations When Using TargetLink with the RTI Bypass Blockset on page 81.

Supported blocks of the RTI Bypass Blockset

The following blocks of the RTI Bypass Blockset are supported:

Blocks to be used only in TargetLink subsystems:

- RTI Bypass Read/Write (ECU Synchronous Read/Write access)
- RTI Bypass Upload/Download (ECU Asynchronous Read/Write access)
- RTI Bypass Function

Blocks to be used in and outside of TargetLink subsystems:

- RTI Bypass Info
- RTI Bypass Setup
- RTI Bypass Build

Blocks to be used only outside of TargetLink subsystems:

- RTI Bypass Interrupt

Note

Some limitations apply when working with the RTI Bypass Blockset. Refer to

- Limitations of the RTI Bypass Blockset (📖 RTI Bypass Blockset Reference)
- Limitations When Using TargetLink with the RTI Bypass Blockset on page 81

Supported platforms

TargetLink supports the following platforms for bypassing code and A2L file generation:

- Generic TRICOREBypassing/Gnu
- Generic MPC5XXXBypassing/Gnu
- Generic V850XBypassing/Gnu
- Generic X86Bypassing/Gnu
- Generic ARMBEBypassing/Gnu
- Generic ARMLEBypassing/Gnu

You can activate the platform in the TargetLink Preferences. Refer to Basics on Using the Preferences Editor (📖 TargetLink Customization and Optimization Guide) and Topic Navigator (📖 TargetLink Tool and Utility Reference).

File generation

The following rules apply to bypass file generation:

- During the on-target build process, the ECU A2L file is copied and extended by the additional measurement and calibration variables from the TargetLink or Simulink model.
- You can reuse the measurement and calibration variables from the original target A2L file.

The bypass code generation delivers the following intermediate files:

- The source and header files
- If applicable, the `DsFxp.lib` TargetLink Fixed-Point Library
- An A2L file fragment
- A makefile fragment

The RTI Bypass Blockset uses the generated files to create the final output of the build process:

- An A2L file for the new ECU application
- A new HEX code image to be flashed on the ECU

For more information on the build process, refer to the documentation of the RTI Bypass Blockset: Build Page (RTIBYPASS_SETUP_BLx for INTERNAL) (📖 RTI Bypass Blockset Reference).

File path

All generated files for bypassing are located in `.\<ModelName>_<Platform>_TL\TL`.

Note

TargetLink deletes all old files from the specified directory before a new file export is performed.

Limitations

Some limitations apply when working with TargetLink and RTI Bypass Blockset to perform on-target bypassing. Refer to Limitations When Using TargetLink with the RTI Bypass Blockset on page 81.

Related topics

Basics

Basics on Preparing Simulink Systems for TargetLink Code Generation (📖 TargetLink Preparation and Simulation Guide)
Basics on Using the Preferences Editor (📖 TargetLink Customization and Optimization Guide)
Dynamically Integrating a Simulink Model into a V-ECU Running on a SCALEXIO System or on a MicroAutoBox III (📖 Virtual Validation Overview)
Dynamically Integrating a Simulink Model into a V-ECU Running on VEOS (📖 Virtual Validation Overview)
General Information on the RTI Bypass Blockset (📖 RTI Bypass Blockset Reference)
Implementing On-Target ECU Interfacing with RTI Bypass Blockset (📖 ECU Interfacing Overview)
Introduction to the dSPACE Internal Bypassing Service Implementation (📖 dSPACE Internal Bypassing Service Implementation)
Limitations When Using TargetLink with the RTI Bypass Blockset.. 81

References

RTI BYPASS BUILD Block (📖 RTI Bypass Blockset Reference)
Topic Navigator (📖 TargetLink Tool and Utility Reference)

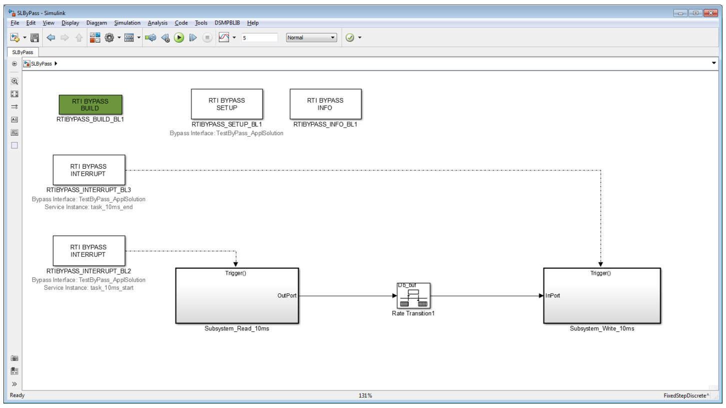
Limitations When Using TargetLink with the RTI Bypass Blockset

AUTOSAR and RTOS blocks

AUTOSAR and RTOS blocks are not supported when modeling with blocks from the RTI Bypass Blockset. TargetLink aborts the code generation with an error

	message when blocks from the RTI Bypass Blockset are used in code generation modes AUTOSAR or RTOS.
Regenerate code for a TargetLink model created via model preparation	When working with a prepared model, reused original A2L variables are created. If you continue working with the TargetLink model all Simulink objects are ignored. A new rebuild with the TargetLink model deletes all DD objects below `/Pool/Variables/tools/Bypassing` (and does not create them again). As a result, the block references a Variable object that does not exist in the Data Dictionary. As a workaround, you can move all imported variables from `/Pool/Variables/tools/Bypassing` to `/Pool/Variables` manually before rebuilding again.
Calibration variables	TargetLink's Code Generator does not evaluate all settings of *TargetLink supported Simulink blocks* and can therefore not use/produce calibration variables. If the `ExportedGlobal` Simulink parameter is set at a TargetLink supported Simulink block, TargetLink cannot specify a calibration variable for bypassing and issues a warning.
Measurement variables	If the `ExportedGlobal` Simulink signal is set to a signal line and its source block does not allow you to specify an output variable, TargetLink cannot use/produce a measurement variable for bypassing and issues a warning. No A2L variable is created for this Simulink signal.
Multiple TargetLink subsystems in a model	You cannot use more than one TargetLink subsystem in a model when modeling with blocks from the RTI Bypass Blockset.
Subsystems triggered from outside containing data ports	TargetLink aborts the bypassing code generation with an error when a subsystem triggered by an RTI BYPASS INTERRUPT block containing a data port.

The following example shows the Subsystem_Read_10ms and Subsystem_Write_10ms subsystems. They are triggered from outside and contain data ports.



This modeling style leads to an error:

```
Error #15773: tl_byPassing_subsystem/Subsystem_Read_10ms. The
following subsystem is called from outside the TargetLink
subsystem and has one or more data ports:
tl_byPassing_subsystem/Subsystem_Read_10ms. This is not
supported in RTI Bypass code generation mode.
```

For further information, contact dSPACE Support (www.dspace.com/go/supportrequest).

Blocks from the RTI Bypass Blockset in referenced subsystems

You cannot use blocks from the RTI Bypass Blockset in referenced subsystems.

Blocks from the RTI Bypass Blockset in subsystems configured for incremental code generation

You cannot use blocks from the RTI Bypass Blockset in subsystems to be generated incrementally.

Using Restart or Init functions

You cannot call Restart or Init functions on the target ECU when working with blocks from the RTI Bypass Blockset. TargetLink issues an error message when the code generation has produced a Restart or an Init function.

Using RTI bypass interrupt blocks

RTI bypass interrupt blocks are not supported inside TargetLink subsystems. The build process aborts with an error.

Subsystems called from outside containing function parameters or return values	For bypassing, TargetLink does not support subsystems called from outside a TargetLink subsystem that contain function parameters or return values in the function signature.

Interoperating with SystemDesk via SWC Containers

Where to go from here

Information in this section

Exchanging Software Component Containers

Where to go from here

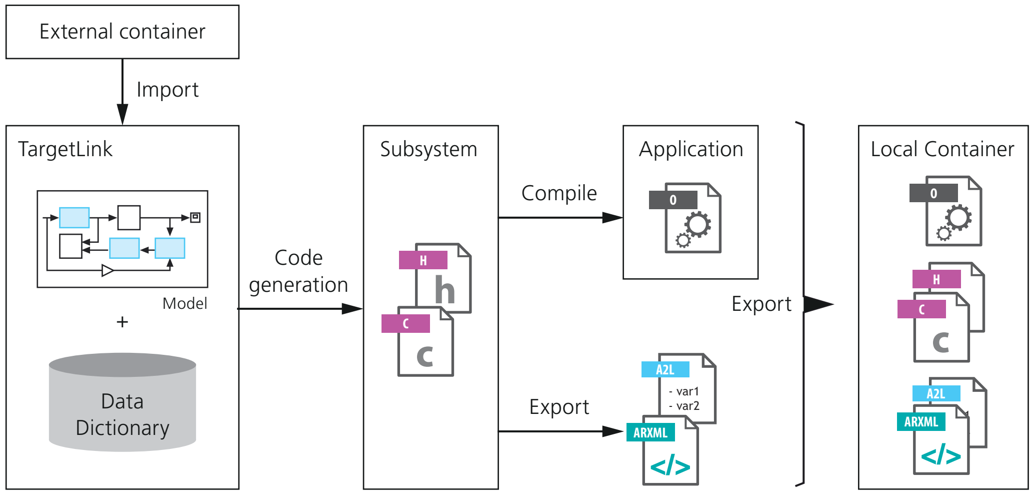
Information in this section

Basics on Exchanging Software Components

Exchanging containers

When exchanging software component containers between TargetLink and SystemDesk, several files are involved.

When exporting software components from a Data Dictionary, TargetLink creates a local container for each software component that is exported. Their contents are described by each container's catalog file (CTLG). All the containers belonging to a Data Dictionary are described in a container set file (CTS).



During import, the local container is either created or synchronized with the corresponding external container that belongs to SystemDesk, using a set of predefined rules.

In the dialog you can change the actions derived from these rules or adapt the rules themselves.

Methods of container exchange

You can use the following methods to exchange software component containers between SystemDesk and TargetLink:

Method	Description
Import from Container `dsdd('Import', 'Format','Container', …)`	Lets you import software component containers.
Export as Container `tl_export_container`	Lets you export software component containers.

Related topics

Basics

References

Export as Container (TargetLink Data Dictionary Manager Reference)
Import from Container (TargetLink Data Dictionary Manager Reference)
tl_export_container (TargetLink API Reference)

How to Import a Container from SystemDesk

Precondition

An external container that was exported from SystemDesk exists.

Method

To import a container from SystemDesk

1. From the TargetLink Data Dictionary Manager menu, select File – Import – from Container.

   TargetLink opens the Import Container dialog.
2. In the Import Container dialog, click the External Catalog File Browse button to select the container that you have exported from SystemDesk.

   The Select Catalog dialog opens.
3. In the Select Catalog dialog, click the ContainerSet Browse button to select the container set file (CTS).

   TargetLink displays the containers belonging to the container set.
4. Select the catalog file (CTLG) of the container that you want to import to the Data Dictionary. Click OK to close the dialog.
5. In the Import Container dialog, check the Import AUTOSAR Files checkbox if required.
6. In the Import Container dialog, select file operations from the Operation list box in the container grid.
7. In the Import Container dialog, click OK to close the dialog.

   TargetLink synchronizes the container files and imports any AUTOSAR files that are selected for import from the container.

Result

You imported a container from SystemDesk.

Related topics

Basics

Basics on Exchanging Software Components.......... 85
Customizing Container Exchange.......... 89
Exchanging Software Component Code.......... 90

References

Import from Container (TargetLink Data Dictionary Manager Reference)

How to Export Containers From TargetLink

Preconditions

You have implemented software components using TargetLink and generated code in the AUTOSAR code generation mode for them.

Note

When exporting containers, TargetLink performs an AUTOSAR file export with default options. You can specify your own default options for importing and exporting AUTOSAR files in the TargetLink Data Dictionary at the `/Pool/Autosar/Config/ImportExport` DD object.

Method

To export containers from TargetLink

1. From the TargetLink Data Dictionary Manager menu, select File – Export – as Container.

   TargetLink opens the Export Container dialog sequentially for the software components that are available in the open TargetLink Data Dictionary.
2. In each Export Container dialog, specify export options as required.
3. Click OK to close each dialog.

   TargetLink exports one catalog file (CTLG) for each software component to the `./_ComponentFiles/<Container>` subdirectory.

Result

You exported a container from TargetLink.

Related topics

Basics

Basics on Exchanging Software Components........ 85
Customizing Container Exchange........ 89
Exchanging Software Component Code........ 90

References

Export as Container (📖 TargetLink Data Dictionary Manager Reference)

Customizing Container Exchange

Customizing container export

You can customize container exchange.

Customizing via hook script

TargetLink provides a hook script that lets you add your own commands when exporting containers. This allows you to adapt container export to your company's development tool chain.

To customize container export, you can derive the **`tl_pre_containerexport_hook`** (📖 TargetLink File Reference) hook script from its template.

The hook script is called immediately before container export.

Providing a custom workflow definition (CTW) file

You can provide a custom workflow definition file to change workflow rules.

Note

It is recommended that only experienced users change the workflow rules. There is a special syntax for defining workflow rules.
For more information, refer to Advanced: Configuring Container Handling (📖 Container Management Manual).

1. Provide a copy of the CTW template via TargetLink's **`tlCustomizationFiles`** API function and save it to `<MyDir>`.
2. Change the CTW file's workflow rules as required.
3. In the Data Dictionary Manager, provide the path to `<MyDir>\<CTW file name>` as the value of the WorkflowDefinitionFile property contained in the `/Pool/Autosar/Config/ContainerExchange` option set or as the value of the **`tl_export_container`** API function's `WorkflowDefinitionFile` property.

Controlling A2L export

You can control whether an A2L file is generated and exported during container exchange via the `/Pool/Autosar/Config/ContainerExchange` option set's PerformA2LExportOnContainerExport property.

You can copy the option set from a DD workspace, based on the dsdd_master_autosar4.dd [System] template or create it via the Create AutosarOptionSet command from the context menu of the DD `/Pool/Autosar/Config` subtree.

Related topics

References

tl_export_container (📖 TargetLink API Reference)
tlCustomizationFiles (📖 TargetLink API Reference)

Exchanging Software Component Code

Where to go from here

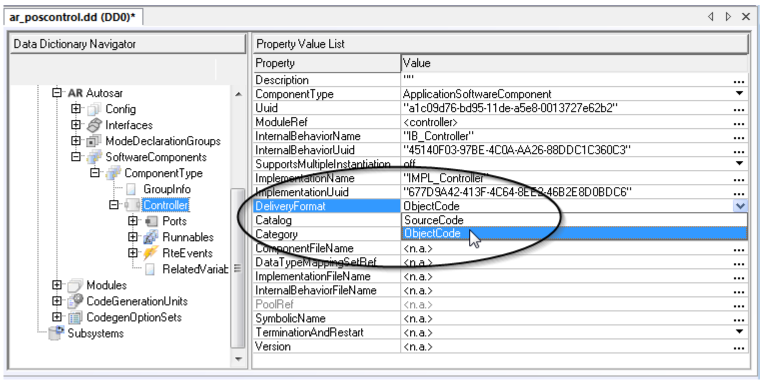
Information in this section

Basics of Exchanging Software Component Code

Delivery format of SWCs

TargetLink allows you to specify the delivery format of software components. This allows you to exchange software component code as source code or as object code in compatibility mode.

You can specify the delivery format at DD SoftwareComponent objects by setting the DeliveryFormat property to `SourceCode` or `ObjectCode`.



Property	Value
Description	""
ComponentType	ApplicationSoftwareComponent
Uuid	"a1c09d76-bd95-11de-a5e8-0013727e62b2"
ModuleRef	<controller>
InternalBehaviorName	"IB_Controller"
InternalBehaviorUuid	"45140F03-97BE-4C0A-AA26-88DDC1C360C3"
SupportsMultipleInstantiation	off
ImplementationName	"IMPL_Controller"
ImplementationUuid	"677D9A42-413F-4C64-8EE2-46B2E8D0BDC6"
DeliveryFormat	ObjectCode
Catalog	SourceCode
Category	ObjectCode
ComponentFileName	<n.a.>
DataTypeMappingSetRef	<n.a.>
ImplementationFileName	<n.a.>
InternalBehaviorFileName	<n.a.>
PoolRef	<n.a.>
SymbolicName	<n.a.>
TerminationAndRestart	<n.a.>
Version	<n.a.>

If you do not specify the DeliveryFormat property, TargetLink assumes the delivery format to be source code.

Note

You have to specify the delivery format of a software component before code generation.

Further information

For an example of exchanging software component code, refer to Example of Exchanging Software Component Object Code on page 91.

Example of Exchanging Software Component Object Code

Introduction

TargetLink allows you to exchange software component code in different delivery formats. This example shows the exchange of software component object code.

Precondition

You have opened your model.

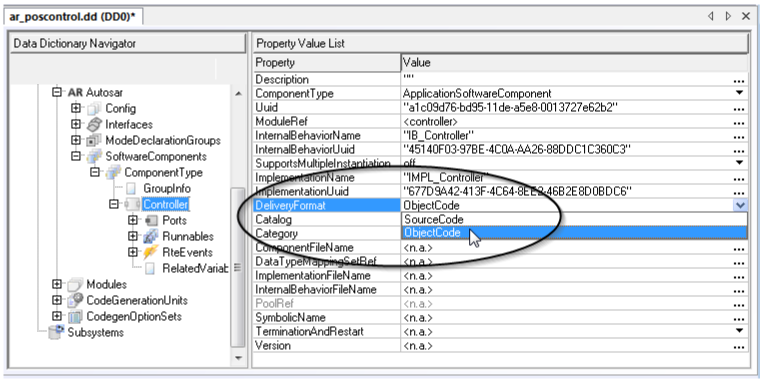
Method

To exchange SWC Code

1 In the Model window, click TargetLink - Data Dictionary Manager.

2 In the Data Dictionary Navigator pane select the DD SoftwareComponent object whose code you want to deliver.

3 Set the DD SoftwareComponent object's DeliveryFormat property to ObjectCode:

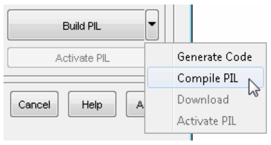


Property	Value
Description	""
ComponentType	ApplicationSoftwareComponent
Uuid	"a1c09d76-bd95-11de-a5e8-0013727e62b2"
ModuleRef	<controller>
InternalBehaviorName	"IB_Controller"
InternalBehaviorUuid	"45140F03-97BE-4C0A-AA26-88DDC1C360C3"
SupportsMultipleInstantiation	off
ImplementationName	"IMPL_Controller"
ImplementationUuid	"677D9A42-413F-4C64-8EE2-46B2E8D0BDC6"
DeliveryFormat	ObjectCode
Catalog	
Category	
ComponentFileName	<n.a.>
DataTypeMappingSetRef	<n.a.>
ImplementationFileName	<n.a.>
InternalBehaviorFileName	<n.a.>
PoolRef	<n.a.>
SymbolicName	<n.a.>
TerminationAndRestart	<n.a.>
Version	<n.a.>

4 In the TargetLink Main Dialog, select the appropriate target from the Code generation target settings list.

5 Click Generate Code and wait for the code generation process to finish.

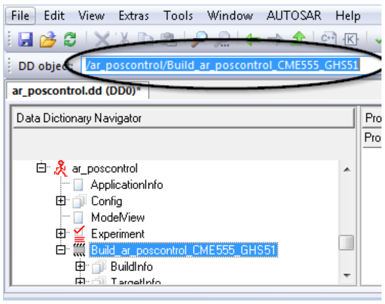
6 From the Build PIL menu button, select Compile PIL.



Wait for the compilation process to finish.

7 If you have built your software component for different target platforms, continue with step 8 on page 92 to specify a dedicated DD Build object. Otherwise continue with step 11 on page 92.

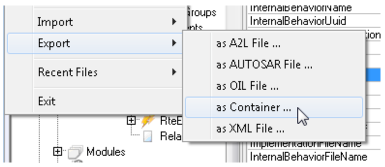
8 In the Data Dictionary Navigator pane, locate the Build object and copy its path to the Clipboard.


File Edit View Extras Tools Window AUTOSAR Help
DD object
/ar_poscontrol/Build_ar_poscontrol_CME555_GHS51
ar_poscontrol.dd (DD0)*
Data Dictionary Navigator
ar_poscontrol
ApplicationInfo
Config
ModelView
Experiment
Build_ar_poscontrol_CME555_GHS51
BuildInfo
TargetInfo

9 From the context menu of the DD `/Pool/Autosar/Config/ImportExport` object select Create Property.

10 Rename the property to `BuildObject` and paste the path you copied in step 8 into its Value field.

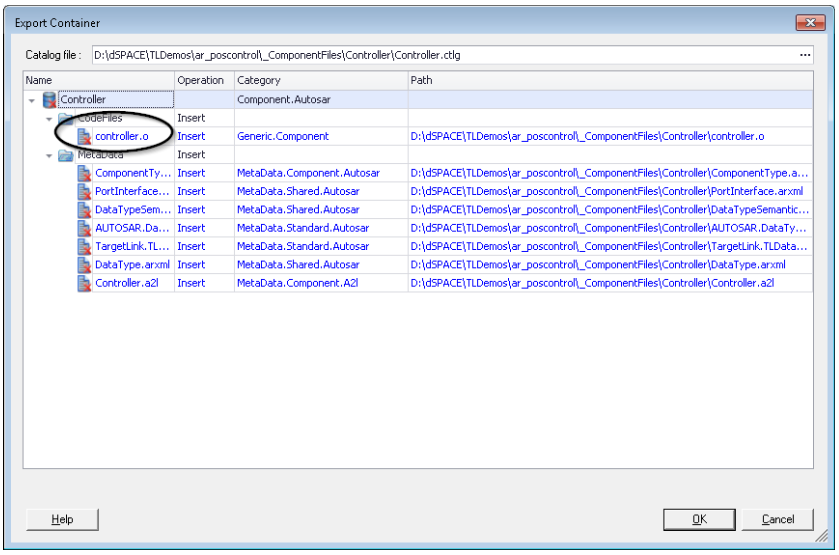
11 From the File menu of the Data Dictionary Manager, select Export - as Container...



The Export Container dialog opens.

12 Click OK to finish the container export.

TargetLink exports the container holding the SWC object code.



Catalog file : D:\dSPACE\TLDemos\ar_poscontrol_ComponentFiles\Controller\Controller.ctlg

Name	Operation	Category	Path
Controller		Component.Autosar	
CodeFiles	Insert		
controller.o	Insert	Generic.Component	D:\dSPACE\TLDemos\ar_poscontrol_ComponentFiles\Controller\controller.o
MetaData	Insert		
ComponentTy...	Insert	MetaData.Component.Autosar	D:\dSPACE\TLDemos\ar_poscontrol_ComponentFiles\Controller\ComponentType.a...
PortInterface...	Insert	MetaData.Shared.Autosar	D:\dSPACE\TLDemos\ar_poscontrol_ComponentFiles\Controller\PortInterface.arxml
DataTypeSem...	Insert	MetaData.Shared.Autosar	D:\dSPACE\TLDemos\ar_poscontrol_ComponentFiles\Controller\DataTypeSemantic...
AUTOSAR.Da...	Insert	MetaData.Standard.Autosar	D:\dSPACE\TLDemos\ar_poscontrol_ComponentFiles\Controller\AUTOSAR.DataTy...
TargetLink.TL...	Insert	MetaData.Standard.Autosar	D:\dSPACE\TLDemos\ar_poscontrol_ComponentFiles\Controller\TargetLink.TLData...
DataType.arxml	Insert	MetaData.Shared.Autosar	D:\dSPACE\TLDemos\ar_poscontrol_ComponentFiles\Controller\DataType.arxml
Controller.a2l	Insert	MetaData.Component.A2l	D:\dSPACE\TLDemos\ar_poscontrol_ComponentFiles\Controller\Controller.a2l

Result

You have exchanged software component object code.

Interoperating with ConfigurationDesk

Introduction

TargetLink lets you interoperate with ConfigurationDesk. Generally, this is possible with all of the following container files:

- Functional Mock-up Unit (FMU)
- Simulink implementation container (SIC)
- V-ECU implementation container (VECU)

However, SIC files offer the most possibilities and this guide focuses on them.

Where to go from here

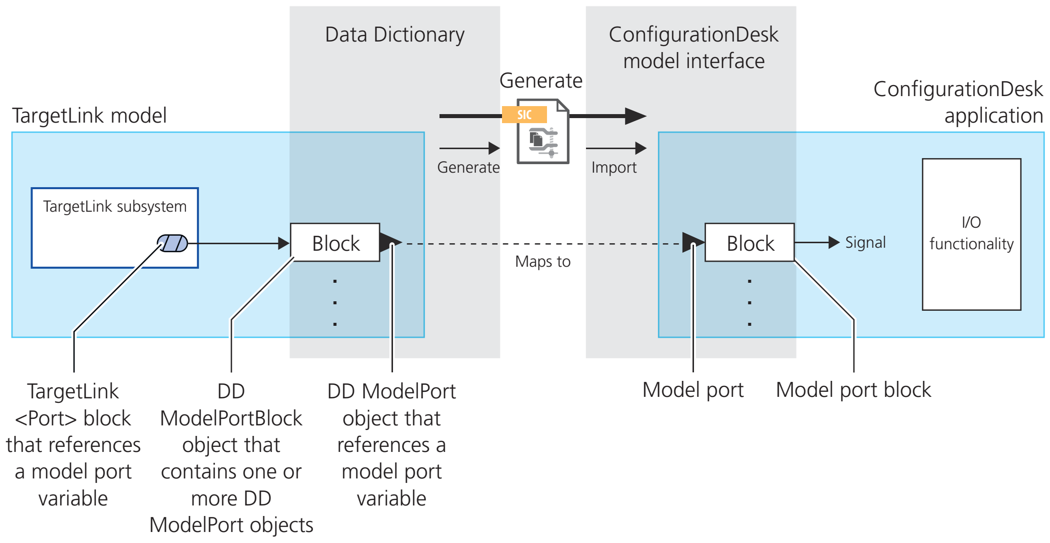
Information in this section

Basics on Interoperating with ConfigurationDesk via SIC Files

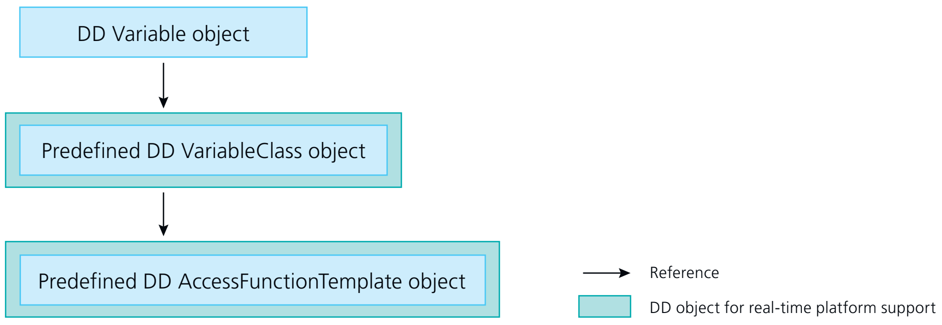
Introduction

You can use the production code generated from a TargetLink subsystem as a behavior model in ConfigurationDesk. This lets you simulate production code on the dSPACE real-time platforms supported by ConfigurationDesk. To do so, you model the interface of the behavior model, generate code, and export the resulting production code in a Simulink implementation container (SIC) that is generated by TargetLink.

Specifying the interface of the behavior model To use a TargetLink model as a behavior model in ConfigurationDesk, you have to specify the model's interface via model port variables.


Data Dictionary
ConfigurationDesk model interface
Generate
ConfigurationDesk application
TargetLink model
SIC
Generate
Import
TargetLink subsystem
Block
Maps to
Block
Signal
I/O functionality
TargetLink <Port> block that references a model port variable
DD ModelPortBlock object that contains one or more DD ModelPort objects
DD ModelPort object that references a model port variable
Model port
Model port block

Specifying model port variables You specify a model port variable by creating a DD Variable object that references a suitable DD VariableClass object.


DD Variable object
Predefined DD VariableClass object
Predefined DD AccessFunctionTemplate object
Reference
DD object for real-time platform support

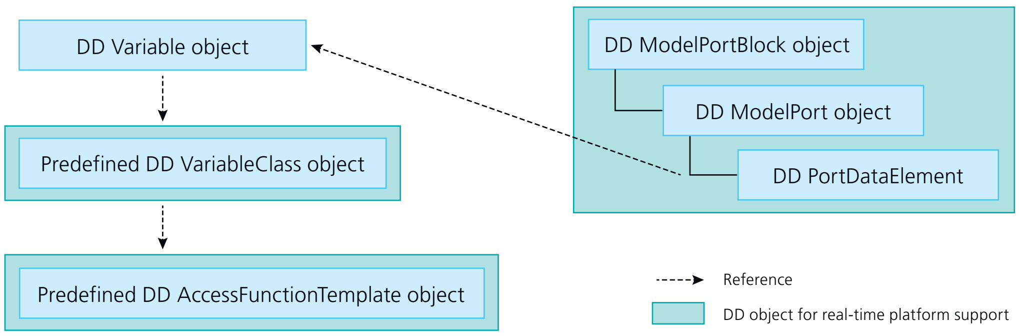
Which variable class to choose depends on the kind of model port that you want to generate to the SIC file.

Model Port in ConfigurationDesk	DD VariableClass Object
Data inport	`/Pool/VariableClasses/RealTimePlatformSupport/DATA_INPORT`[1]
Data outport	`/Pool/VariableClasses/RealTimePlatformSupport/DATA_OUTPORT`[1]

[1] Not available in Data Dictionaries that are based on the dsdd_master_basic.dd [System] DD template. Copy the DD VariableClass objects from a workspace that contains a Data Dictionary based on the dsdd_master_advanced DD template.

Grouping model port variables via ModelPortBlock objects You have to group model port variables of the same kind via DD ModelPortBlock objects:

- Add a DD ModelPort object to the ModelPortBlock object's subtree for each model port variable.
- Reference the model port variable at the ModelPort object via its PortDataElement child object's VariableRef property.

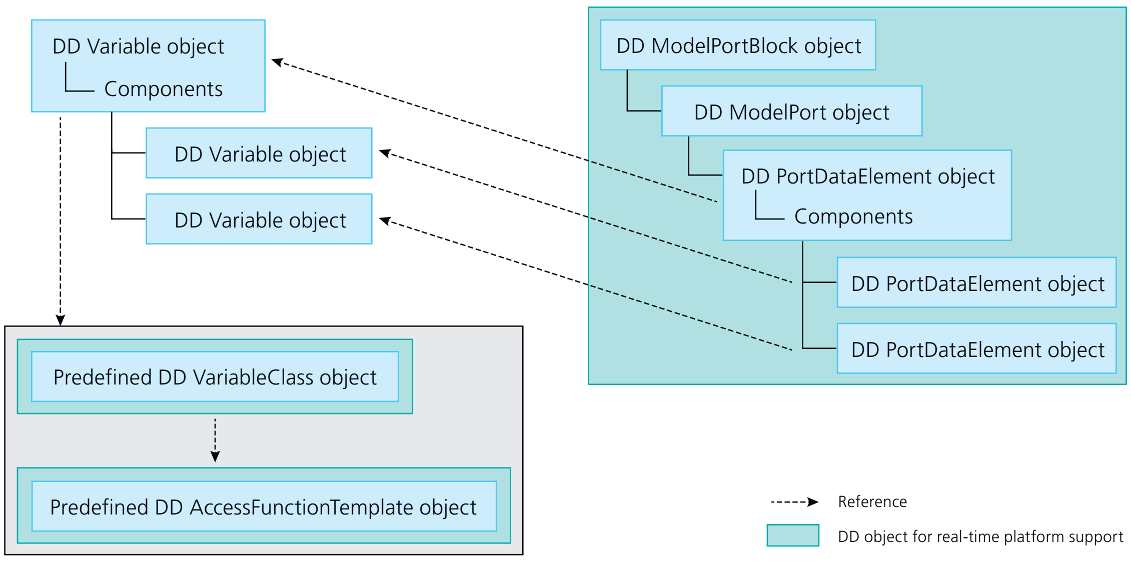

DD Variable object
Predefined DD VariableClass object
Predefined DD AccessFunctionTemplate object
DD ModelPortBlock object
DD ModelPort object
DD PortDataElement
Reference
DD object for real-time platform support

Each DD ModelPortBlock object results in a model port block in ConfigurationDesk. Each DD ModelPort object results in a model port in ConfigurationDesk. The following table shows how to specify the direction of the model port's data flow:

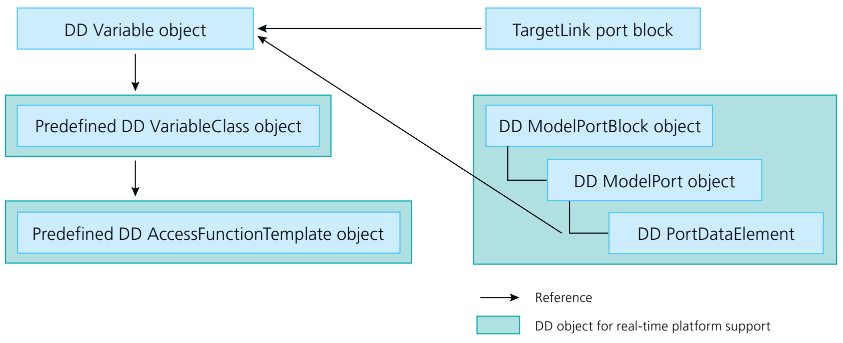
Model Port in ConfigurationDesk	DD ModelPort Object Kind
Data inport	The DD ModelPort object's Kind property is set to `Inport`
Data outport	The DD ModelPort object's Kind property is set to `Outport`

Uniquely identify each DD ModelPortBlock and each DD ModelPort object via their individual ID properties.

Creating DD ModelPort objects for structured model port variables To create a ModelPort object for structured model port variables, proceed as follows:


DD Variable object
Components
DD Variable object
DD Variable object
Predefined DD VariableClass object
Predefined DD AccessFunctionTemplate object
DD ModelPortBlock object
DD ModelPort object
DD PortDataElement object
Components
DD PortDataElement object
DD PortDataElement object
Reference
DD object for real-time platform support

Using model port variables in the model You use model port variables in the model at the behavior model's interface. To do so, you reference the model port variables at suitable TargetLink port blocks.


DD Variable object
TargetLink port block
Predefined DD VariableClass object
DD ModelPortBlock object
DD ModelPort object
Predefined DD AccessFunctionTemplate object
DD PortDataElement
Reference
DD object for real-time platform support

Model Port in ConfigurationDesk	TargetLink Port Block
Data inport	InPort or Bus Inport[1]
Data outport	OutPort or Bus Outport[2]

[1] For structured data inports.
[2] For structured data outports.

Note

TargetLink port blocks that reference model port variables can be placed in any atomic subsystem in the subsystem hierarchy if they are connected as follows:

- Each of the blocks must be connected to a Simulink port block on the root level of the TargetLink subsystem
- The signal lines must only contain Simulink port blocks.

Using the API You can use the following DD API functions to manipulate model-port-variable-related DD objects:

- `[errorCode] = dsdd('CreateSuitableModelPorts', … <objectIdentifier>[,<ModelPortBlockGroup>]);`
- `errorCode = dsdd('DeleteObsoleteModelPorts', … <objectIdentifier>);`
- `[errorCode] = dsdd('ReplaceID', … <objectIdentifier>[,'ApplyToObjectKind',<objectkind>] [,'ApplyToID',<applytoid>]);`

Logging

You can log model port variables in MIL simulation, SIL simulation, PIL simulation, and in real-time simulation.

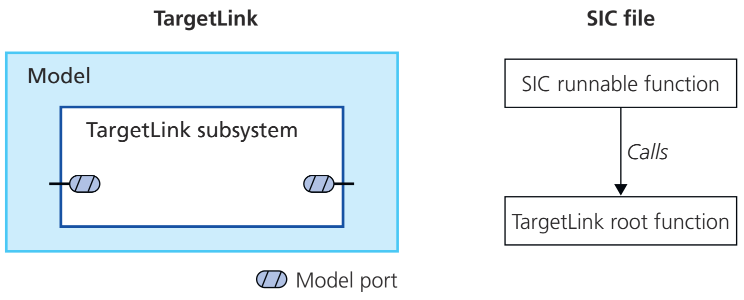
Real-time simulation For real-time simulation, TargetLink creates suitable auxiliary variables for you to log and display in dSPACE ControlDesk. They are named according to the following naming scheme:

Naming Scheme	
Model Port Variable	**Auxiliary Variable**
`<ModelPortVariableName>`	`Aux_<ModelPortVariableName>_<SubsystemID>`

Descriptions of these auxiliary variables are placed in the A2L or TRC file that is contained in the SIC file. When sorting a TRC according to the system hierarchy, the descriptions of the auxiliary variables are placed at the corresponding TargetLink port block.

Mapping of root step functions to SIC runnable functions

During code generation, TargetLink generates a root function for the TargetLink subsystem. This function is called by a corresponding SIC runnable function that also is generated by TargetLink. In ConfigurationDesk, this SIC runnable function is imported as a runnable function that can be assigned to a task.



Generating multiple SIC runnable functions in one SIC file You can also generate SIC files that contain multiple SIC runnable functions. Refer to Details on Generating SIC Files for ConfigurationDesk That Contain Several SIC Runnable Functions on page 100.

Model port variables in production code

In the production code, the model port variables are represented by special value copy AF. Depending on the RealTimePlatformSupport Code Generator option, their definitions are placed in different modules and additional code might be generated:

RealTimePlatformSupport[1] Code Generator option	**on**	**`off`**
Access Function Definitions	Simulation code, defined in a module called `<CGUName>_ioap`.[2]	Production code, defined via the access function's module specification.
Entry and Exit functions for Real-Time Simulation	Empty void functions (Simulation code), defined in a module called `<CGUName>_ioap`. These functions are required only to build the simulation application for SIL simulation and PIL simulation.[2]	-
	Example taken from the REAL_TIME_PLATFORM_SUPPORT demo model.	

	An entry function definition in the `MultipleRootFunctions_ioap.c` file:
	`void ap_Entry_Sb2_SysA_10ms(void)`
	`{`
	`}`
	The call of this entry function in the `MultipleRootFunctions.c` file:
	`void Sb2_SysA_10ms(void)`
	`{`
	`   …`
	`   ap_Entry_Sb2_SysA_10ms();`
	`   ap_GetAIn2_ModelPort(&Aux_AIn2_ModelPort_b);`
	`   …`
	`   ap_GetAIn1_ModelPort(&Aux_AIn1_ModelPort_b);`
	`   …`
	`   ap_Exit_Sb2_SysA_10ms();`
	`}`
		-

[1] By default, this Code Generator option is set to `off`. Enable it before generating code that you want to export to an SIC file. For your convenience, TargetLink assumes that the option is set to on when you are using the TargetLink SIC Manager.
[2] ConfigurationDesk generates the actual definitions when building the real-time application (RTA) for real-time simulation.

Generating an SIC file without specifying the interface

If you do not specify the interface of the model explicitly, TargetLink can still generate an SIC file if the characteristics of the block variables allow for it. However, you cannot influence the model port variables in this case. Therefore, it is best to model the interface as described in Specifying the interface of the behavior model on page 95.

Demo model

The REAL_TIME_PLATFORM_SUPPORT demo model shows all the aspects of preparing a TargetLink model for the generation of an SIC file.

Limitations

For limitations on generating SIC files with TargetLink, refer to SIC Generation Limitations (TargetLink Limitation Reference).

Related topics

Basics

References

Details on Generating SIC Files for ConfigurationDesk That Contain Several SIC Runnable Functions

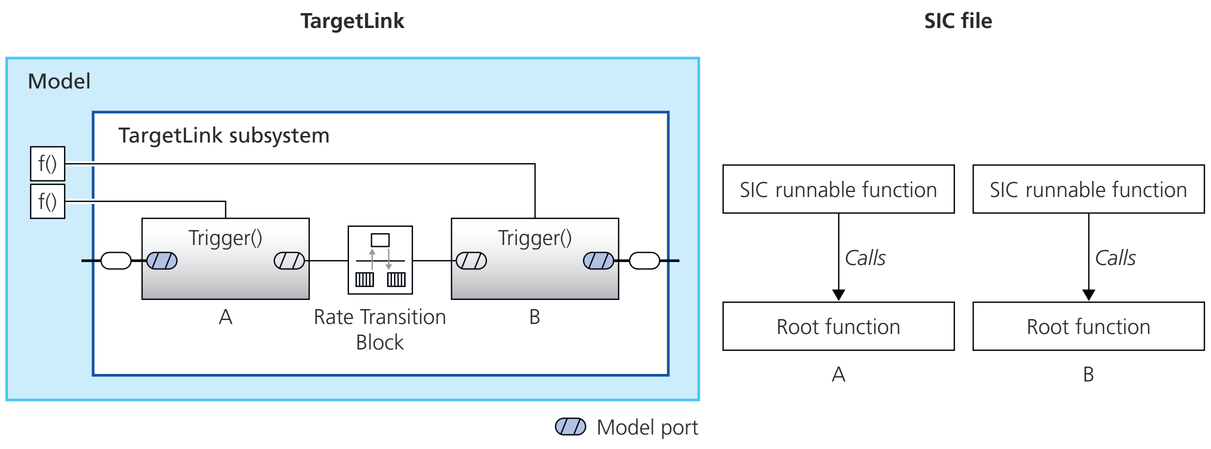
Introduction

In ConfigurationDesk, you can map the SIC runnable function contained in the behavior model imported from a Simulink implementation container (SIC) file to a task to determine how it is executed in the real-time application (RTA).

Several SIC runnable functions per SIC file

TargetLink lets you generate SIC files that contain multiple SIC runnable functions that you can assign to different tasks in ConfigurationDesk. This facilitates the execution of the different functions asynchronously or at different rates.

Modeling pattern for multiple SIC runnable functions The SIC file generated by TargetLink contains multiple SIC runnable functions if you use the following modeling pattern:



Each SIC runnable function is represented by an externally triggered function-call subsystem. The interface of the behavior model is modeled via TargetLink port blocks that reference model port variables.

Note

The SIC runnable functions generated from this modeling pattern are asynchronous functions. To execute them at different rates, you must assign them to different tasks that are triggered by events in ConfigurationDesk.

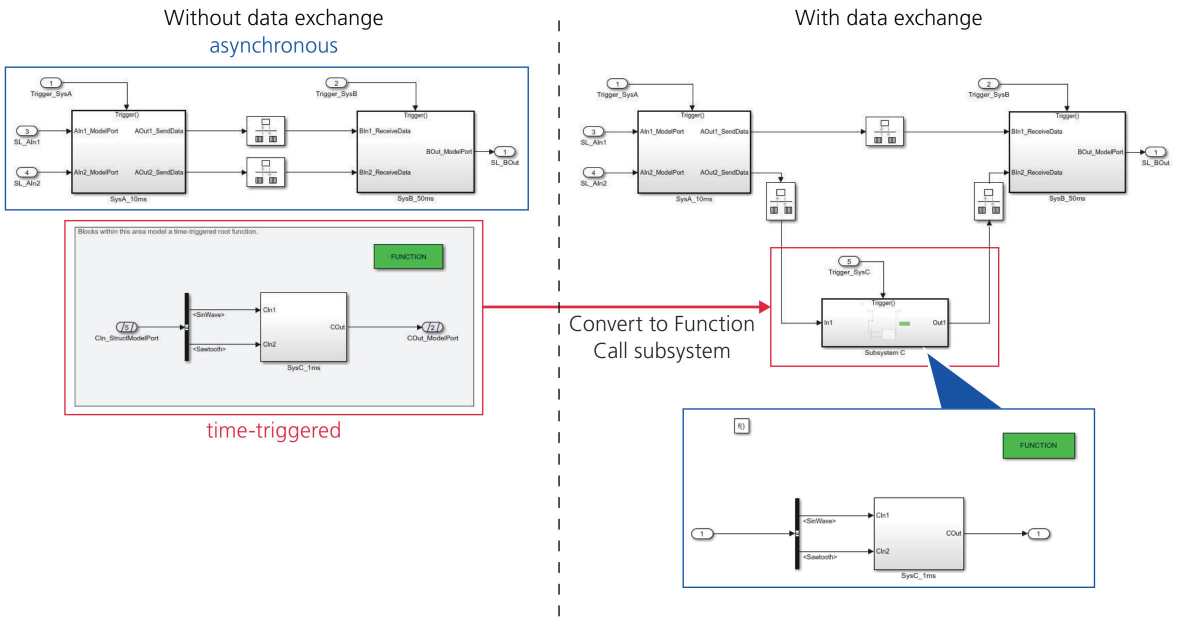
Modeling the data flow between SIC runnable functions If the different SIC runnable functions have to communicate, you have to model the correct data flow.

Ensuring data integrity The Simulink Rate-Transition block between the externally triggered function-call subsystems is used to enable data exchange in SIL simulation. Selecting its **Ensure data integrity during data transfer** checkbox instructs TargetLink to generate Simulation code that ensures data consistency via a buffer during real-time simulation.

Note

The simulation code generated by TargetLink to ensure data consistency works differently than the Simulink Rate Transition block. The code is placed in a simulation code module called `<CGUName>_rsfc`.

Communication between asynchronous and time-triggered SIC runnable functions You can model SIC runnable functions that are time-triggered or asynchronous in the same TargetLink subsystem. However, if time-triggered and asynchronous SIC runnable functions must exchange data, the blocks for which a time-triggered SIC runnable function will be generated must be moved into a Function Call subsystem as shown in the following example:



Limitations

For limitations on generating SIC files with TargetLink, refer to SIC Generation Limitations (📖 TargetLink Limitation Reference).

Related topics

Basics

References

Details on SIC Files Generated by TargetLink

Introduction

The Simulink implementation container (SIC) files generated by TargetLink have the following characteristics.

Variable description files

An SIC file contains a variable description file to facilitate experiments with measurement and calibration systems, such as ControlDesk. Two file formats are supported: A2L files and TRC files.

A2L file contained in the SIC The A2L file contained in the SIC describes the calibratable or measurable variables. Refer to Basics on Specifying the Variables to Export to A2L Files on page 120.

TRC file contained in the SIC By default, the TRC file contained in the SIC describes all global variables that are defined in the production code. The contents of the file can be sorted in different ways:

- By Simulink's subsystems hierarchy
- By TargetLink's function hierarchy

Note

During logging of state variables that are described in a variable description file, the value of the variables at $\mathtt{t}_{\mathtt{(n+1)}}$ is logged at $\mathtt{t}_{\mathtt{(n)}}$. This differs from SIC files created by the Model Interface Package for Simulink, which logs the value at $\mathtt{t}_{\mathtt{(n)}}$. This is the case because TargetLink's production code does not separate output code and update code.

Assigning multiple model implementations to the same application process in ConfigurationDesk

In ConfigurationDesk, the SIC files generated by TargetLink can be used as model implementations. These model implementations must be assigned to an application process. It is possible to assign one model implementation to one application process or multiple model implementations to the same application process. If you want to assign multiple model implementations to the same application process, note the following:

If the SIC files generated by TargetLink result from different TargetLink subsystems whose TRC files describe the same global variables, the addresses of these variables will not be updated and cannot be measured or calibrated in ControlDesk during simulation of the simulation application. In this case, ConfigurationDesk displays a conflict.

To avoid this conflict, ensure not to use the same global variables in SIC files resulting from different TargetLink subsystems, if you want to use these SIC files as model implementations that are assigned to the same application process in ConfigurationDesk. Usually, this pertains to the following kind of variables:

- Variables of global scope that are externally defined.

  For example, these definitions can occur in production code when you partitioned your model to generate code into different modules.
- Variables of global scope that are defined in stub code modules.

If using the same global variables cannot be avoided, but no corresponding conflicts are to be displayed in ConfigurationDesk, you can instruct TargetLink to not include them in the TRC files associated with each SIC file. This can be done via the `ExternalVariablesInTrcFile` and `StubCodeVariablesInTrcFile` properties of the `tlGenerateSic` API function.

Note

During the build process in ConfigurationDesk, ConfigurationDesk creates separate copies of global variables that are used by several model implementations, which are assigned to the same application process. This differs from a SIL/PIL simulation, where each global variable exists only once in the simulation application.

Example Suppose you generated two SIC files called A and B to use as model implementations that are assigned to the same application process.

The TargetLink subsystems used to generate A and B share some global variables. To resolve a conflict in ConfigurationDesk, you generated B with the `ExternalVariablesInTrcFile` and `StubCodeVariablesInTrcFile` properties of the `tlGenerateSic` API function set to `off`.

Accordingly, the TRC file of B does not contain descriptions of the global variables shared by A and B.

When experimenting with ControlDesk, you can access the global variables used by A but not those used by B, because ConfigurationDesk created separate copies during the build process.

Variable initialization and start-stop behavior

TargetLink generates SIC files in a way, that all variables that require an initial value are reset to their initial value when the real-time application (RTA) starts. Therefore, the variables must be initialized in at least one restart function.

If you do not want to explicitly specify which variables to initialize in a restart function, you can set the InitializeVariablesViaRestartFunction Code Generator option to `3 - Always`.

Fixed-Point Library files

If the production code requires macros or functions belonging to TargetLink's Fixed-Point Library, the required files are included in the SIC file.

Priority of SIC runnable functions

If the SIC file generated by TargetLink contains more than one SIC runnable function, their priority is determined as follows:

- The SIC runnable function generated for the TargetLink subsystem receives the highest priority.
- The priority of the SIC runnable functions generated from externally triggered function-call subsystems is derived from the order of the function-call trigger signals: the lower the number of the Inport block that carries the FcnCall signal, the higher the SIC runnable function's priority.

You can later change these priorities in ConfigurationDesk.

AUTOSAR code and stub Rte

If you generate code for an AUTOSAR software component and generate an SIC file for it, this file also contains the source files for a simulation Rte (stub Rte) to use on the real-time hardware.

Limitations

For limitations on generating SIC files with TargetLink, refer to SIC Generation Limitations (TargetLink Limitation Reference).

Related topics

Basics

Basics on Interoperating with ConfigurationDesk via SIC Files.. 94
SIC Generation Limitations (TargetLink Limitation Reference)

References

TargetLink SIC Manager (TargetLink Tool and Utility Reference)
tlGenerateSic (TargetLink API Reference)

How to Generate an SIC File for ConfigurationDesk

Objective

To export the production code of an existing TargetLink model as Simulink implementation container (SIC) files and use it as a behavior model in ConfigurationDesk.

Workflow

This workflow consists of the following parts:

- Specifying model port variables, refer to Part 1 on page 105.
- Modeling the interface, refer to Part 2 on page 107.
- Generating the SIC file, refer to Part 3 on page 108.

Part 1

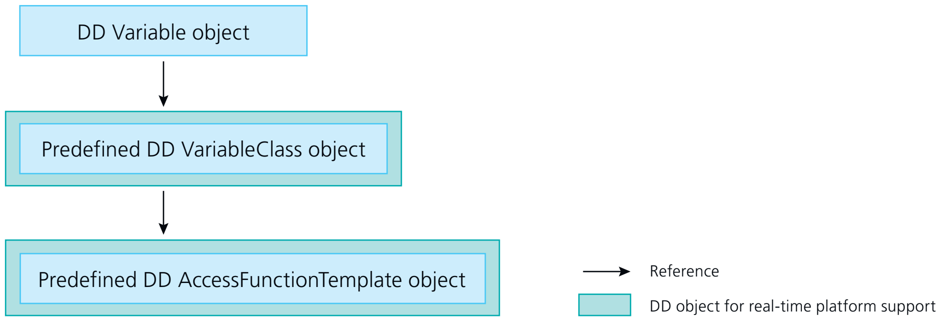
To specify model port variables

1 In the Data Dictionary Manager, create a model port variable for each model port that is required in ConfigurationDesk.

2 Specify the properties of each model port variable as shown in the following table:

Propery	Value
Class	`/Pool/VariableClasses/RealTimePlatformSupport/DATA_INPORT`[1] or `/Pool/VariableClasses/RealTimePlatformSupport/DATA_OUTPORT`[2]

[1] For data inports in ConfigurationDesk.
[2] For data outports in ConfigurationDesk.


DD Variable object
Predefined DD VariableClass object
Predefined DD AccessFunctionTemplate object
Reference
DD object for real-time platform support

3 Create the DD `/Pool/RealTimePlatformSupport` subtree.

4 To its ModelInterface child object, add a ModelPortBlock object to group several model port variables of the same kind in one model port block. Repeat this step as required.

5 To each ModelPortBlock object's subtree, add a DD ModelPort object for each model port variable that you want to group.

6 Specify each ModelPort object's Kind property as required.

Note

Make sure that all the ModelPort objects contained in the ModelPortBlock are of the same Kind.

7 At the VariableRef property of each PortDataElement object, reference exactly one model port variable.

8 Specify the ID properties of all the PortDataElement and ModelPortBlock objects you created. To do this, use the **`dsdd ReplaceID`** command, for example.

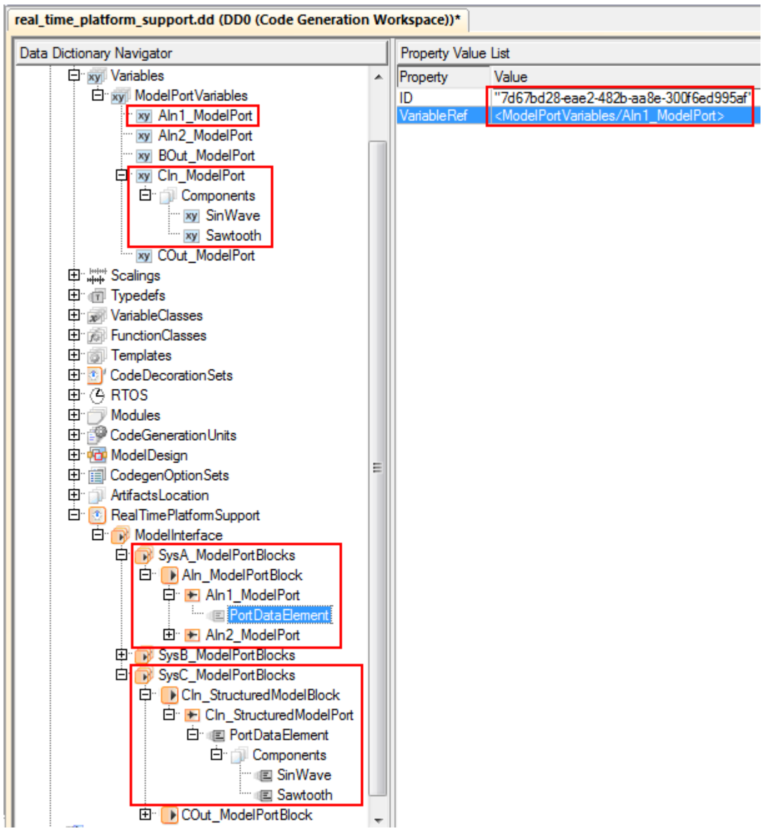
Tip

You can use the following API functions to manipulate model-port-variable-related DD objects:

- `[errorCode] = dsdd('`**`CreateSuitableModelPorts`**`', …`
  `<objectIdentifier>[,<ModelPortBlockGroup>]);`
- `errorCode = dsdd('`**`DeleteObsoleteModelPorts`**`', …`
  `<objectIdentifier>);`
- `[errorCode] = dsdd('`**`ReplaceID`**`', …`
  `<objectIdentifier>[,'ApplyToObjectKind',<objectkind>]`
  `[,'ApplyToID',<applytoid>]);`

Interim result

You created model port variables in the DD and grouped multiple model port variables of the same kind in one model port block. The following screenshot is taken from the REAL_TIME_PLATFORM_SUPPORT demo model:



Property	Value
ID	"7d67bd28-eae2-482b-aa8e-300f6ed995af"
VariableRef	<ModelPortVariables/AIn1_ModelPort>

In a next step, you model the interface of the TargetLink subsystem whose production code you want to export to the SIC file.

Part 2

To model the interface

Tip

These instructions assume that you do not want to change the existing interface of the production code. Other modeling styles are possible but not recommended.

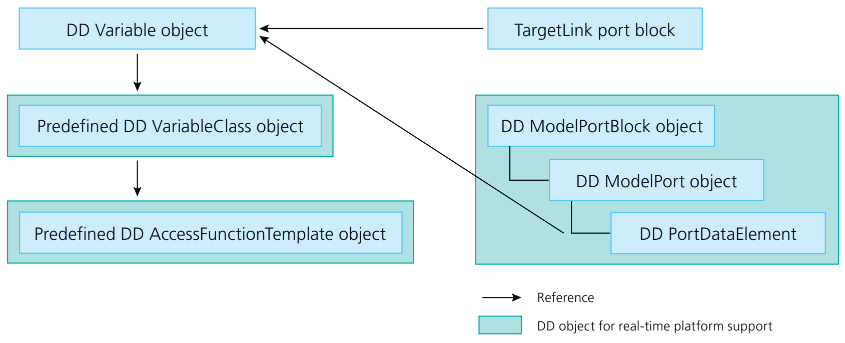
1 In your TargetLink model, open the TargetLink subsystem.

2 Select its entire content and press **Ctrl** + **G** to convert the selection into a subsystem.

3 At the TargetLink port blocks that are directly connected to the interfaces of the subsystem created in step 2 and the TargetLink subsystem, reference a model port variable.

Interim result

By referencing the model port variables at TargetLink port blocks, you established a mapping between the model elements and the DD objects representing the model port blocks in ConfigurationDesk.


DD Variable object
TargetLink port block
Predefined DD VariableClass object
Predefined DD AccessFunctionTemplate object
DD ModelPortBlock object
DD ModelPort object
DD PortDataElement
Reference
DD object for real-time platform support

In a next step, you generate the SIC file.

Part 3

To generate the SIC file

1. In the TargetLink Main Dialog, switch to the Tools page and click Export SIC.
2. In the TargetLink SIC Manager dialog, configure the export of the SIC file as required.
3. Click Export SIC, wait for the export to finish, and close the dialog.

Result

You specified the interface of a TargetLink subsystem to export its production code in an SIC file.

Related topics

Basics

Basics on Interoperating with ConfigurationDesk via SIC Files.. 94
REAL_TIME_PLATFORM_SUPPORT (📖 TargetLink Demo Models)
SIC Generation Limitations (📖 TargetLink Limitation Reference)

References

TargetLink SIC Manager (📖 TargetLink Tool and Utility Reference)
tlGenerateSic (📖 TargetLink API Reference)

Exchanging Data

Where to go from here

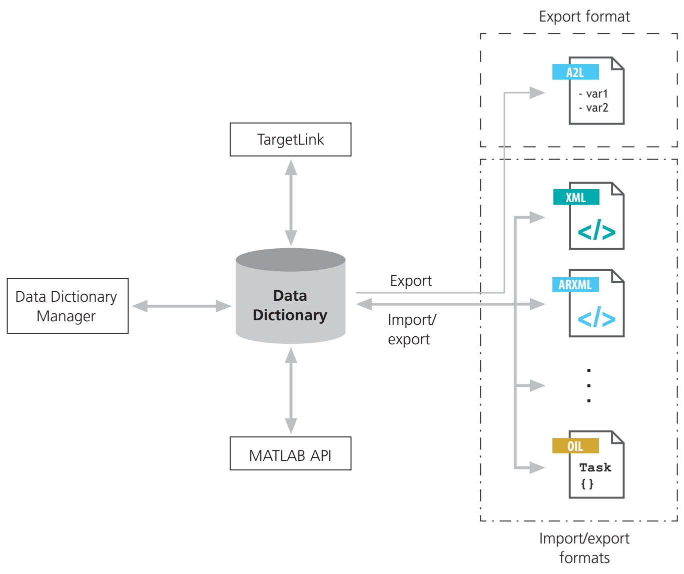
Information in this section

Introduction

Basics on Importing and Exporting Files

Import/export feature of the TargetLink Data Dictionary

The TargetLink Data Dictionary can import and export data files in various file formats, as shown in the following illustration.



Import and export modules

The TargetLink Data Dictionary has an interface for importing and exporting data. This interface is used by the Data Dictionary import and export modules. You do not have to update the TargetLink Data Dictionary to support further file formats, just add the necessary import and export modules.

Mapping data to DD objects

The contents of the imported data files are stored in a subtree of the TargetLink Data Dictionary. The data is mapped to DD objects and their properties according to predefined rules. The rules are implemented in import and export modules that support a specified file format. They ensure that the data is written to the right position during file import, and that DD objects of the subtree are written to the appropriate parts of the data files during file export.

For details, refer to Import Export Mapping Tables.

Related topics

Basics

Basics on Importing and Exporting Data (📖 TargetLink Data Dictionary Basic Concepts Guide)
Basics on Importing and Exporting Data Between MATLAB and DD Variable Objects............... 203

Exchanging A2L Files

Where to go from here

Information in this section

Introducing A2L Files

Where to go from here

Information in this section

Basics on the A2L File Format

A2L specification

ASAM e.V. (Association for Standardisation of Automation and Measuring Systems e.V.) has specified the A2L file format. You can download this specification and more information on the ASAM MCD standard from http://www.asam.net.

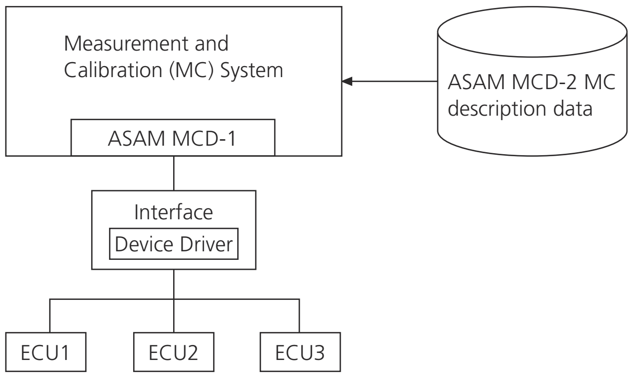
A2L files are ASCII files that list the parameters and measurement variables of the application running on an electronic control unit (ECU).

Purpose and content of A2L files

A2L files provide a measurement and calibration (MC) system with all the information required to access electronic control units (ECUs). An A2L file contains information on the following:

- The ECU, e.g., its memory structure
- The ECU application, including its variables

- The methods for converting the ECU's integer representation of variables into physical units
- The communication interface between the ECU and the MC system. This can be an ASAP1b interface, such as CCP (CAN Calibration Protocol) or XCP (Universal Measurement and Calibration Protocol), or another interface conforming with the ASAM MCD-1 standard.


Measurement and Calibration (MC) System
ASAM MCD-1
ASAM MCD-2 MC description data
Interface
Device Driver
ECU1
ECU2
ECU3

Supported versions

Version 1.6.1 of the ASAM MCD-2 MC standard is supported for A2L file export. Elements introduced in later versions of the standard are not supported.

Encoding

A2L files are encoded in UTF-8 without BOM. Refer to Overview of the Supported Character Sets on page 223.

Important elements of A2L files

A2L files contain various elements, which are defined by specific keywords. The following table briefly describes the most important keywords:

Keyword	Description
`ASAP2_VERSION`	Information on the version of the ASAM MCD-2 MC standard with which the A2L file complies.
`AXIS_PTS`	Information on the axis properties of look-up tables.
`CHARACTERISTIC`	Information on a calibratable parameter, such as its name, data type, memory address, scaling method, upper and lower limits, table data, etc.
`COMPU_METHOD`	Conversion method to convert the ECU's integer representation of a variable into its physical value or into a verbal description.[1)]
`FUNCTION`	Information on the functional hierarchy of the parameters and measurement variables in the ECU application.
`IF_DATA`	Configuration data for the ASAP1b interface that accesses the ECU hardware, e.g., via the CAN Calibration Protocol (CCP).
`MEASUREMENT`	Information on a measurable variable such as its name, data type, memory address, scaling methods, upper and lower limits, etc.
`MOD_COMMON`	Description data for the `MODULE`, e.g., byte order and variable alignments of the ECU.
`MOD_PAR`	Management data, e.g., version number of the ECU and memory layouts.

Keyword	Description
`MODULE`	Description of a software module of the ECU application, defining its calibratable and measurable variables, conversion methods and functions.
`PROJECT`	Definition of basic information on the ECU application, such as its name and a brief description.
`RECORD_LAYOUT`	Information on the data storage in the ECU memory, such as the data structures of look-up tables.[1]
`SYMBOL_LINK`	Information on the offset of the element defined in the A2L file relative to the address of the corresponding symbol of the linker MAP or ELF file. For structure components, TargetLink can generate two different formats for the value of the `SYMBOL_LINK` keyword: ▪ `<RootStructName>` **`<Offset>`** - For addresses obtained from a MAP file ▪ `<RootStructName>.`**`<ComponentName>`** **`0`** - For addresses obtained from an ELF file, where the address is known, the offset accordingly is `0` You can control the format to use via a XSL stylesheet.

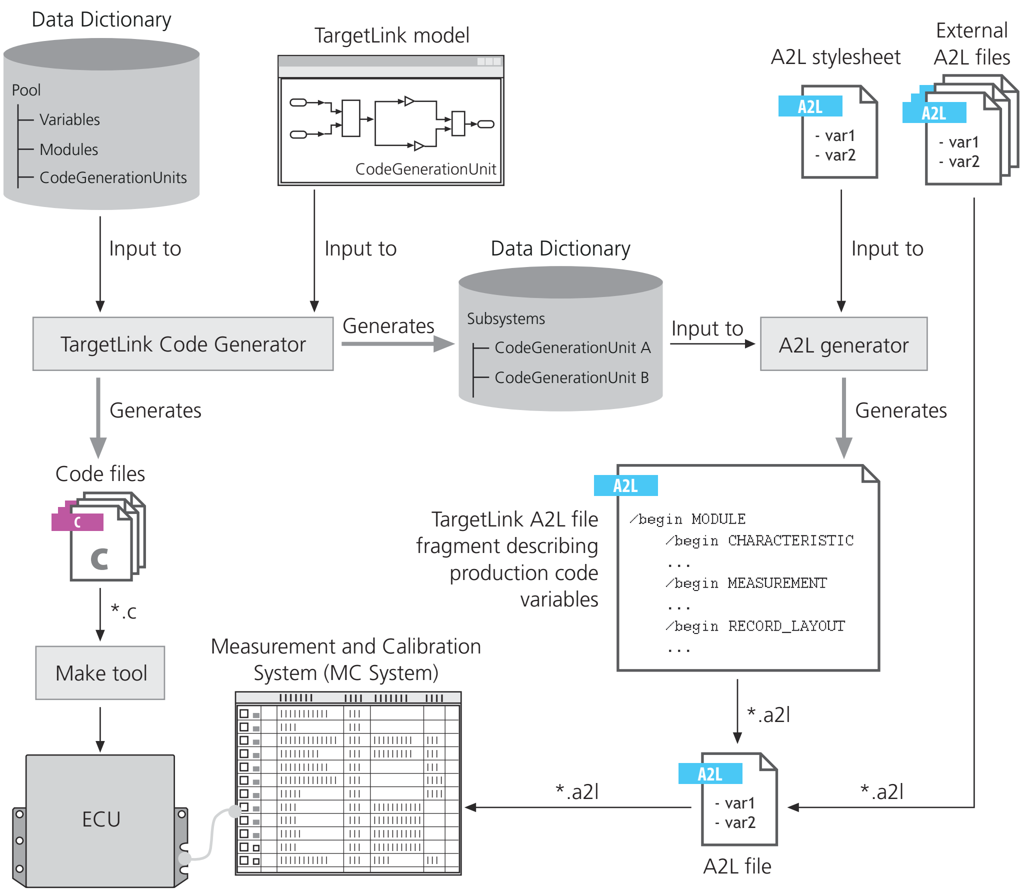
[1] You can add a prefix to this element's name via the **Name prefix** edit field on the **Export as A2L File** dialog or the **NamePrefix** property of the `dsdd_export_a2l_file` API function. Providing a prefix prevents naming conflicts when you merge incrementally generated A2L files.

Element definitions can be nested, some can contain parameters.

A2L files for ECU code generated with TargetLink

You can export an A2L file for production code generated with TargetLink from the DD **Subsystems** area. The code generated for a code generation unit (CGU) is represented by a DD **Subsystem** object.

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



By default, an A2L file exported by TargetLink from a single DD **Subsystem** object contains only the measurable and calibratable variables that are defined in this CGU's production code. This lets you incrementally generate A2L files that you can use in distributed development.

Note

If you want to incrementally generate A2L files, specify a unique name for each file and provide a name prefix for `COMPU_METHOD` and `RECORD_LAYOUT` elements. Providing a prefix prevents naming conflicts when you merge incrementally generated A2L files.

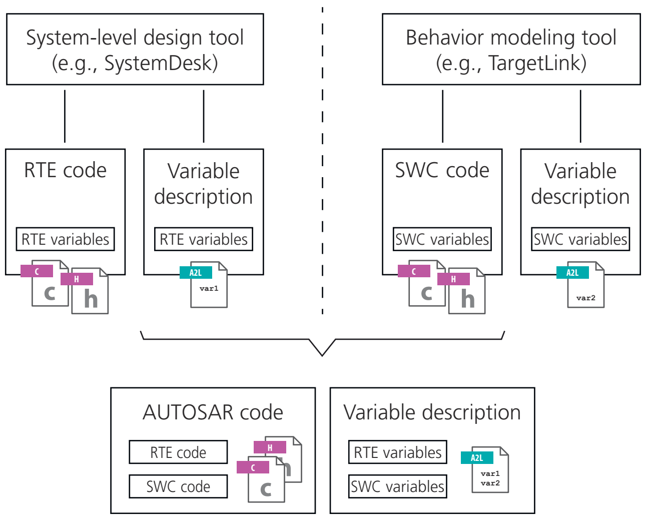
If you generate an A2L file from multiple DD **Subsystem** objects at the same time, the resulting A2L file contains the descriptions of the measurable and calibratable variables defined in the production code represented by all of the selected DD **Subsystem** objects.

Depending on your settings, this file can be an A2L file fragment without the `PROJECT` frame or a complete A2L file that contains the `PROJECT` frame. You can use the file with systems that provide measurement and calibration functionality and have the necessary A2L import utilities, such as ControlDesk.

AUTOSAR As a behavior modeling tool, TargetLink generates production code for software components (SWC code). Accordingly, the A2L files generated by TargetLink contain only descriptions of measurable or calibratable variables that were defined by TargetLink. By design, variables belonging to the Classic AUTOSAR RTE are *not* included, because they are defined by a system-level design tool, such as dSPACE SystemDesk.

If you need an A2L file that contains descriptions of Classic AUTOSAR RTE variables (RTE code), have it generated by a system-level design tool, such as dSPACE SystemDesk.

For more information, refer to Basics on Tools for Developing ECU Software as Described by Classic AUTOSAR (TargetLink Classic AUTOSAR Modeling Guide), Basics on Simulating Classic-AUTOSAR-Compliant SWCs (TargetLink Classic AUTOSAR Modeling Guide), and Basics on Generated Code (TargetLink Classic AUTOSAR Modeling Guide).



Example This is the typical structure of an A2L file:

```
ASAP2_VERSION 1 61
/begin PROJECT prj_name
   "project description"
   ...
   /begin MODULE name1
      ... (information about module name1) ...
   /end MODULE
   ...
   /begin MODULE name<n>
      ... (information about module name<n>) ...
   /end MODULE
/end PROJECT
```

These are subelements within a `MODULE` element:

```
/begin MODULE name
   "module description"
   /begin MOD_COMMON ...
      ... (general description data for the module) ...
   /end MOD_COMMON
   /begin IF_DATA ...
      ... (ASAP1b interface description) ...
   /end IF_DATA
   /begin CHARACTERISTIC ...
      ... (information about calibratable variable) ...
   /end CHARACTERISTIC
   ...
   /begin MEASUREMENT ...
      ... (information about measurable variable) ...
   /end MEASUREMENT
   ...
   /begin COMPU_METHOD ...
      ... (information about conversion method) ...
   /end COMPU_METHOD
   ...
   /begin AXIS_PTS ...
      ... (information about axis points) ...
   /end AXIS_PTS
   ...
   /begin  RECORD_LAYOUT
    ... (information on record layout of an adjustable object
         in memory)
   /end RECORD_LAYOUT
   ...
/end MODULE
```

Related topics

Basics

Basics on Customizing the A2L Export... 169

References

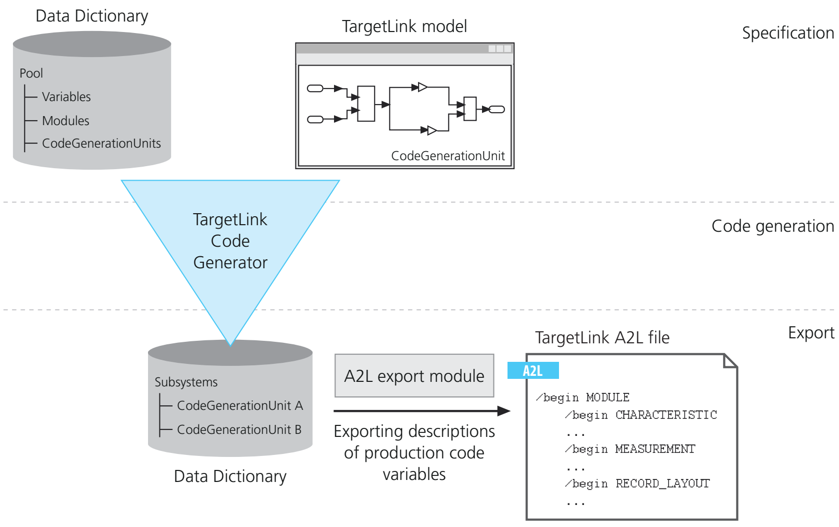
dsdd_export_a2l_file (📖 TargetLink API Reference)
Export as A2L File (📖 TargetLink Data Dictionary Manager Reference)

Basics on the Workflow for Exporting A2L Files

Workflow steps

The prerequisite for exporting A2L files is that you generated production code for your application.

The following illustration shows the process of A2L file generation and its three phases: specification, code generation and export.



The following table shows the workflow steps for exporting an A2L file:

Step	Description	Instruction
Preparing A2L Export		
1	Provide the data you want to export to the A2L file.	Specifying the Data to Export and Generating Code on page 120
2	Generate code for one or more code generation units (CGU) associated with the data that you want to export. The data you want to export must reside in DD **Subsystem** objects in the DD **Subsystems** area. DD **Subsystem** objects are automatically created by the Code Generator. Each represents the code generated for a CGU. **Note** The A2L export module exports data only from DD **Subsystem** objects that were generated by the Code Generator.	
3	Specify a **Build** object that provides information on the target for which you want to export the A2L file. A2L files provide a description of the calibratable and measurable variables of the ECU application. This description includes information such as byte order, alignment, and pointer length. This information is platform-(target/compiler)-dependent. Because it is required by the A2L export module, you must provide it in the DD **Build** object. **Note** Because many calibration systems support updating the variable addresses, it is possible to generate an A2L file without address information. However, if the generated A2L file has to contain the address information, it must also be provided in the DD **Build** object.	Specifying the Build Object on page 131

Step	Description	Instruction
4	Specify the ASAP1b interface. The ASAP1b interface defines the communication layer between an ECU and a measurement and calibration (MC) system connected to the ECU. Information on the interface will be included in the A2L file.	Specifying the ASAP1b Interface on page 143
5	Specify conversion methods for your variables.	Specifying the Conversion of Variable Values on page 150
Exporting A2L Files		
6	Export the A2L file.	Exporting the A2L File on page 161

Limitations

There are some limitations for generating an A2L file from an application built with TargetLink. Refer to ASAM MCD-2 MC File Generation Limitations (📖 TargetLink Limitation Reference).

Related topics

References

ASAM MCD-2 MC File Generation Limitations (📖 TargetLink Limitation Reference)

Preparing the A2L Export

Where to go from here

Information in this section

Specifying the Data to Export and Generating Code

Where to go from here

Information in this section

Basics on Specifying the Variables to Export to A2L Files

Exportable variables

TargetLink creates descriptions of variables in A2L files only if these variables are used in the production code of the code generation unit (CGU) it was generated from. Consider the following table:

Variable Used In	External Variable	Description in A2L File
Production code	Yes	Yes[1)]
	No	Yes
Stub code	Yes	No
	No	No

1) Select the External variables checkbox in the Export as A2L File dialog or set the ExternalVariables property of the `dsdd_export_a2l_file` API function to on to have TargetLink include external production code variables into the A2L file.

Additionally, you can instruct TargetLink to include the descriptions of variables that are defined in external code but not used in TargetLink's production code into its A2L files. Refer to Exporting A2L Files Containing Variables Defined in External Code on page 164.

Specifying the variables to export

Specify the production code variables you want to export to an A2L file by setting an appropriate variable class as shown in the following table:

A2L Element to Export	Predefined Variable Classes	User-Specified Variable Classes
`CHARACTERISTIC`	▪ `CAL` ▪ `STATIC_CAL` ▪ `EXTERN_CAL`[1)]	▪ Set the Info property to `readwrite`. ▪ Clear the `ERASABLE` checkbox in the DD Optimization property's Edit bitfield dialog.
`MEASUREMENT`	▪ `DISP` ▪ `STATIC_DISP` ▪ `EXTERN_DISP`[1)]	▪ Set the Info property to `readonly`. ▪ Clear the `ERASABLE` checkbox in the DD Optimization property's Edit bitfield dialog.

1) Select the External variables checkbox in the Export as A2L File dialog or set the ExternalVariables property of the `dsdd_export_a2l_file` API function to on to have TargetLink include variables associated with this class into the A2L file.

You can specify the variable class as shown in the following table:

Model Element or DD Object	Method
TargetLink block	Select the correct variable class directly in the block dialog
DD Variable objects	Reference the variable class at each DD Variable object You can reference these Variable objects at the following locations: ▪ At TargetLink blocks ▪ At DD look-up table (Block) objects

Note

Via DD look-up table objects you can centrally specify look-up tables in the Data Dictionary. You can use these objects for DD-based code generation and to reference them at TargetLink look-up table blocks.
Using them in DD-based code generation lets you generate look-up table structs and look-up table variables in code modules that are independent of model elements. You can then generate A2L files that describe these look-up tables.
For an example refer to the `CG_FROM_DD` (TargetLink Demo Models) demo model.

Mapping look-up-table-related variables

Variables generated for the table data or axes of look-up tables are mapped to the A2L elements by block- or object-specific rules shown in the following table:

Block or DD Object	Rule
TargetLink Look-Up Table block	▪ If the Table is calibratable or measurable, a `CHARACTERISTIC` entry of `CURVE` type is created with a standard axis, a common axis or a fixed axis. For details, refer to Basics on Controlling the Representation of Look-Up-Table-Related Variables in the A2L File on page 123. ▪ If the Table is neither calibratable nor measurable, and if the Axis is calibratable or measurable, only an `AXIS_PTS` entry is created for the Axis.
DD Look-up Table (1-D) object	
TargetLink Look-Up Table (2-D) block	▪ If the Table is calibratable or measurable, a `CHARACTERISTIC` entry of `MAP` type is created with standard axes, common axes or fixed axes. For details, refer to Basics on Controlling the Representation of Look-Up-Table-Related Variables in the A2L File on page 123. ▪ If the Table is neither calibratable nor measurable, ▪ And if Axis#1 (row) is calibratable or measurable, an `AXIS_PTS` entry is created for Axis#1 (row). ▪ And if Axis#2 (column) is calibratable or measurable, an `AXIS_PTS` entry is created for Axis#2 (column).
DD Look-up Table (2-D) object	
TargetLink Prelookup block	▪ If the Axis (breakpoint data) is calibratable or measurable, an `AXIS_PTS` entry is created.
DD PreLook-Up Index Search Block object	
TargetLink Interpolation Using Prelookup block	▪ If the Table is calibratable or measurable, a `CHARACTERISTIC` entry of `CURVE` type (for tables of one dimension) or `MAP` type (for tables of two dimensions) is created. ▪ The `CURVE`'s/`MAP`'s axis corresponding to the Interpolation Using Prelookup block's input is generated as a `COM_AXIS` if the input is connected to the Prelookup block, otherwise as a `FIX_AXIS`.
DD Interpolation Block object	
TargetLink Direct Look-Up Table (n-D) block	

Providing constrained limits

If you specify constrained limits via the DD Variable object's Min and Max properties, TargetLink will use this information to generate the `LowerLimit` and `UpperLimit` strings of `CHARACTERISTIC`, `MEASUREMENT`, and `AXIS_PTS` A2L elements.

If you do not explicitly specify constrained limits, TargetLink will calculate them from the variable's LSB, offset, and data type to generate the `LowerLimit` and `UpperLimit` strings.

Generating code

Generate code for one or more code generation units (CGU) associated with the data that you want to export.

The data you want to export must reside in DD Subsystem objects in the DD Subsystems area. DD Subsystem objects are automatically created by the Code Generator. Each represents the code generated for a CGU.

Note

The A2L export module exports data only from DD Subsystem objects that were generated by the Code Generator.

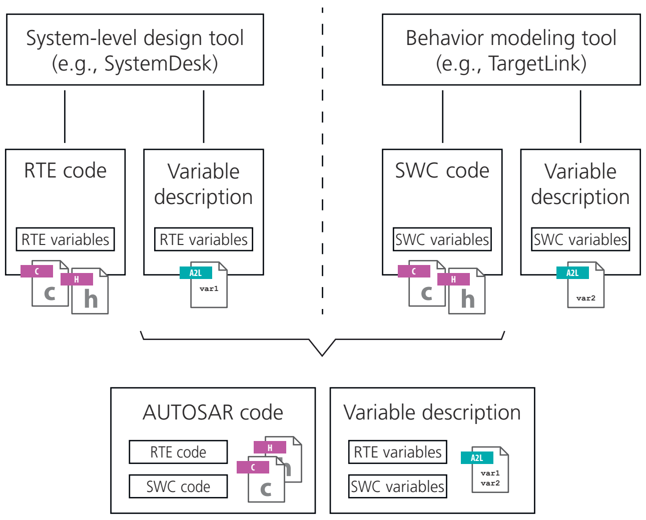
Mapping code generation units In the A2L file, TargetLink can generate the description of measurable and calibratable variables to a single `MODULE` element named `<CGU>`. To do this, clear the Merge A2L modules checkbox in the Export as A2L File dialog or set the MergeA2LModules property of the `dsdd_export_a2l_file` API function to `off`.

Mapping the function hierarchy In model-based code generation, TargetLink maps variables of different functions to A2L `FUNCTION` file elements to reflect the production code's function hierarchy in the A2L file.

AUTOSAR As a behavior modeling tool, TargetLink generates production code for software components (SWC code). Accordingly, the A2L files generated by TargetLink contain only descriptions of measurable or calibratable variables that were defined by TargetLink. By design, variables belonging to the Classic AUTOSAR RTE are *not* included, because they are defined by a system-level design tool, such as dSPACE SystemDesk.

If you need an A2L file that contains descriptions of Classic AUTOSAR RTE variables (RTE code), have it generated by a system-level design tool, such as dSPACE SystemDesk.

For more information, refer to Basics on Tools for Developing ECU Software as Described by Classic AUTOSAR (TargetLink Classic AUTOSAR Modeling Guide), Basics on Simulating Classic-AUTOSAR-Compliant SWCs (TargetLink Classic AUTOSAR Modeling Guide), and Basics on Generated Code (TargetLink Classic AUTOSAR Modeling Guide).



Related topics

Basics

References

Basics on Controlling the Representation of Look-Up-Table-Related Variables in the A2L File

Introduction

You can control the representation of look-up-table-related variables in the A2L file.

Look-up table in production code

Look-up table can be represented in production code via plain variables or via a structured data type, the so called *table map*.

By default, TargetLink generates plain variables for look-up table axes and tables values and a table map that contains pointer to them. You can change this by specifying suitable DD LookupFunctionTemplate objects.

The following table shows declarations of table maps generated from the TABLE2D (TargetLink Demo Models) demo model:

Declaration With Pointer	Declaration Without Pointer

`typedef struct MAP_Tab2DS17I1T4169_a_tag {`
`   UInt8 Nx;`
`   UInt8 Ny;`
`   const UInt16 * x_table;`
`   const UInt16 * y_table;`
`   const UInt16 * z_table;`
`} MAP_Tab2DS17I1T4169_a;`
	`typedef struct MyMAP_tag {`
	`   UInt8 Nx;`
	`   UInt8 Ny;`
	`   const volatile UInt16 x_table[14];`
	`   const volatile UInt16 y_table[11];`
	`   const volatile UInt16 z_table[14][11];`
	`} MyMAP;`

Look-up table structures in A2L files

Via the Use look-up structs checkbox on the Export as A2L File dialog, you can specify to represent look-up tables in the A2L file as a single structure variable, that is similar to the code (table map) generated by TargetLink.

Whether you have to select this checkbox depends on how the look-up tables are represented in production code.

Pointers in table maps

By default, TargetLink generates table maps that contain pointers to the axis and table variables. If the table maps in production code contain pointers, clear the Use look-up structs checkbox.

Note

Some measurement and calibration (MC) systems do not support access to table values and axis points via pointers. In this case, do not use look-up table structs for A2L file export.

The look-up tables are represented in the A2L file by separate table values and axis points.

The A2L file contains the following elements:

- One `CHARACTERISTIC` element for each look-up table. The `CHARACTERISTIC` element contains a reference to a shared axis (`COM_AXIS`) for each axis of the look-up table.
- `AXIS_PTS` elements for each axis of the look-up table. These axes can be shared by other look-up tables.
- One `RECORD_LAYOUT` element for each axis and one for the table values. If both axes have the same data type, only one `RECORD_LAYOUT` element is created for both of them.

Tip

You can create DD LookupFunctionTemplate objects to customize record layouts, refer to Basics on Customizing Table Maps (TargetLink Preparation and Simulation Guide).

Example

```
/begin CHARACTERISTIC
    Sa1_Look_Up_Table__2D__table    /* Name */
    "torque values"    /* LongIdentifier */
    MAP    /* Type */
    0x0000    /* Address */
    UWORD_COL_DIRECT    /* Deposit */
    0    /* MaxDiff */
    EQ_LSB_0_015625_OFF_m256    /* Conversion */
    -256    /* LowerLimit */
    767.984375    /* UpperLimit */
    BYTE_ORDER MSB_LAST
    /begin AXIS_DESCR
        COM_AXIS    /* Attribute */
        NO_INPUT_QUANTITY    /* InputQuantity */
        EQ_LSB_0_125    /* Conversion */
        14    /* MaxAxisPoints */
        0    /* LowerLimit */
        8191.875    /* UpperLimit */
        BYTE_ORDER MSB_LAST
        AXIS_PTS_REF Sa1_Look_Up_Table__2D__axis_1
    /end AXIS_DESCR
    /begin AXIS_DESCR
        COM_AXIS    /* Attribute */
        NO_INPUT_QUANTITY    /* InputQuantity */
        EQ_LSB_0_001953125    /* Conversion */
        11    /* MaxAxisPoints */
        0    /* LowerLimit */
        127.998046875    /* UpperLimit */
        BYTE_ORDER MSB_LAST
        AXIS_PTS_REF Sa1_Look_Up_Table__2D__axis_2
    /end AXIS_DESCR
/end CHARACTERISTIC
```

```
/begin AXIS_PTS
    Sa1_Look_Up_Table__2D__axis_1    /* Name */
    "speed axis"    /* LongIdentifier */
    0x0000    /* Address */
    NO_INPUT_QUANTITY    /* InputQuantity */
    UWORD_X_INCR_DIRECT    /* Deposit */
    0    /* MaxDiff */
    EQ_LSB_0_125    /* Conversion */
    14    /* MaxAxisPoints */
    0    /* LowerLimit */
    8191.875    /* UpperLimit */
    BYTE_ORDER MSB_LAST
/end AXIS_PTS
```

```
/begin AXIS_PTS
    Sa1_Look_Up_Table__2D__axis_2    /* Name */
    "throttle angle axis"    /* LongIdentifier */
    0x0000    /* Address */
    NO_INPUT_QUANTITY    /* InputQuantity */
    UWORD_X_INCR_DIRECT    /* Deposit */
    0    /* MaxDiff */
    EQ_LSB_0_001953125    /* Conversion */
    11    /* MaxAxisPoints */
    0    /* LowerLimit */
    127.998046875    /* UpperLimit */
    BYTE_ORDER MSB_LAST
/end AXIS_PTS
```

```
/begin RECORD_LAYOUT
    UWORD_COL_DIRECT    /* Name */
    FNC_VALUES 1 UWORD COLUMN_DIR DIRECT
/end RECORD_LAYOUT
```

```
/begin RECORD_LAYOUT
    UWORD_X_INCR_DIRECT    /* Name */
    AXIS_PTS_X 1 UWORD INDEX_INCR DIRECT
/end RECORD_LAYOUT
```

No pointers in table maps

If you customized the table maps not to contain pointers by specifying a suitable DD LookupFunctionTemplate object, select the Use look-up structs checkbox.

Note

This is best because only the table map is in memory, not each single variable.
Additionally, sharing the axes does not make sense because each table map belongs to exactly one look-up table.

The A2L file then contains the following elements:

- One `CHARACTERISTIC` element for each look-up table. The `CHARACTERISTIC` element contains a description (`AXIS_DESCR`) of a standard axis (`STD_AXIS`) for each axis of the look-up table.
- One `RECORD_LAYOUT` element for the look-up table structure variable. It contains information on the position in the structure variable at which table values and axis points data are stored.

Example

```
/begin CHARACTERISTIC
    Sa1_Look_Up_Table__2D__map    /* Name */
    "2-D Look-Up table struct for \"torquemap\""    /* LongIdentifier */
    MAP    /* Type */
    0x0000    /* Address */
    MyMAP    /* Deposit */
    0    /* MaxDiff */
    EQ_LSB_0_015625_OFF_m256    /* Conversion */
    -256    /* LowerLimit */
    767.984375    /* UpperLimit */
    BYTE_ORDER MSB_LAST
    /begin AXIS_DESCR
        STD_AXIS    /* Attribute */
        NO_INPUT_QUANTITY    /* InputQuantity */
        EQ_LSB_0_125    /* Conversion */
        14    /* MaxAxisPoints */
        0    /* LowerLimit */
        8191.875    /* UpperLimit */
        BYTE_ORDER MSB_LAST
    /end AXIS_DESCR
    /begin AXIS_DESCR
        STD_AXIS    /* Attribute */
        NO_INPUT_QUANTITY    /* InputQuantity */
        EQ_LSB_0_001953125    /* Conversion */
        11    /* MaxAxisPoints */
        0    /* LowerLimit */
        127.998046875    /* UpperLimit */
        BYTE_ORDER MSB_LAST
    /end AXIS_DESCR
/end CHARACTERISTIC
```

```
/begin RECORD_LAYOUT
    MyMAP    /* Name */
    NO_AXIS_PTS_X 1 UBYTE
    NO_AXIS_PTS_Y 2 UBYTE
    AXIS_PTS_X 3 UWORD INDEX_INCR DIRECT
    AXIS_PTS_Y 4 UWORD INDEX_INCR DIRECT
    FNC_VALUES 5 UWORD COLUMN_DIR DIRECT
/end RECORD_LAYOUT
```

No table map

If your production code does not contain table maps, clear the Use look-up structs checkbox.

The A2L file contains the following elements:

- One `CHARACTERISTIC` element for each look-up table. The `CHARACTERISTIC` element contains a reference to a shared axis (`COM_AXIS`) for each axis of the look-up table.
- `AXIS_PTS` elements for each axis of the look-up table. These axes can be shared by other look-up tables.
- One `RECORD_LAYOUT` element for each axis and one for the table values. If both axes have the same data type, only one `RECORD_LAYOUT` element is created for both of them.

Tip

You can create DD LookupFunctionTemplate objects to customize record layouts, refer to Basics on Customizing Table Maps (TargetLink Preparation and Simulation Guide).

In TargetLink, you can use a shared axis as the x-axis or y-axis for two or more look-up tables.

Example

```
/begin CHARACTERISTIC
    Sa1_Look_Up_Table__2D__table    /* Name */
    "torque values"    /* LongIdentifier */
    MAP    /* Type */
    0x0000    /* Address */
    UWORD_COL_DIRECT    /* Deposit */
    0    /* MaxDiff */
    EQ_LSB_0_015625_OFF_m256    /* Conversion */
    -256    /* LowerLimit */
    767.984375    /* UpperLimit */
    BYTE_ORDER MSB_LAST
    /begin AXIS_DESCR
        COM_AXIS    /* Attribute */
        NO_INPUT_QUANTITY    /* InputQuantity */
        EQ_LSB_0_125    /* Conversion */
        14    /* MaxAxisPoints */
        0    /* LowerLimit */
        8191.875    /* UpperLimit */
        BYTE_ORDER MSB_LAST
        AXIS_PTS_REF Sa1_Look_Up_Table__2D__axis_1
    /end AXIS_DESCR
    /begin AXIS_DESCR
        COM_AXIS    /* Attribute */
        NO_INPUT_QUANTITY    /* InputQuantity */
        EQ_LSB_0_001953125    /* Conversion */
        11    /* MaxAxisPoints */
        0    /* LowerLimit */
        127.998046875    /* UpperLimit */
        BYTE_ORDER MSB_LAST
        AXIS_PTS_REF Sa1_Look_Up_Table__2D__axis_2
    /end AXIS_DESCR
/end CHARACTERISTIC
```

```
/begin AXIS_PTS
    Sa1_Look_Up_Table__2D__axis_1    /* Name */
    "speed axis"    /* LongIdentifier */
    0x0000    /* Address */
    NO_INPUT_QUANTITY    /* InputQuantity */
    UWORD_X_INCR_DIRECT    /* Deposit */
    0    /* MaxDiff */
    EQ_LSB_0_125    /* Conversion */
    14    /* MaxAxisPoints */
    0    /* LowerLimit */
    8191.875    /* UpperLimit */
    BYTE_ORDER MSB_LAST
/end AXIS_PTS
```

```
/begin AXIS_PTS
    Sa1_Look_Up_Table__2D__axis_2    /* Name */
    "throttle angle axis"    /* LongIdentifier */
    0x0000    /* Address */
    NO_INPUT_QUANTITY    /* InputQuantity */
    UWORD_X_INCR_DIRECT    /* Deposit */
    0    /* MaxDiff */
    EQ_LSB_0_001953125    /* Conversion */
    11    /* MaxAxisPoints */
    0    /* LowerLimit */
    127.998046875    /* UpperLimit */
    BYTE_ORDER MSB_LAST
/end AXIS_PTS
```

```
/begin RECORD_LAYOUT
    UWORD_COL_DIRECT    /* Name */
    FNC_VALUES 1 UWORD COLUMN_DIR DIRECT
/end RECORD_LAYOUT
```

```
/begin RECORD_LAYOUT
    UWORD_X_INCR_DIRECT    /* Name */
    AXIS_PTS_X 1 UWORD INDEX_INCR DIRECT
/end RECORD_LAYOUT
```

Shared axes and working point

Suppose you referenced the same DD Variable object at the axes of two different look-up tables. Suppose further, that the DD VariableClass object that is referenced by this Variable object, has its Optimization property set to `MERGEABLE`. This means that both look-up tables have the same axis.

If you do not use look-up table structs for A2L file export, there will be a unique `AXIS_PTS` element representing the axis. Because this `AXIS_PTS` element is referenced by the `CHARACTERISTIC` elements created for the two look-up tables, it is a shared axis.

Note

Look-up tables can have different input signals. Therefore, `AXIS_PTS` elements can only be specified unambiguously if they are used by only one look-up table. In this case, the input quantity property is therefore set to `NO_INPUT_QUANTITY` in the A2L file. However, if the input quantity is known and measurable, it is set for the `AXIS_DESCR` element of the `MAP` and `CURVE` entry.
If you use this A2L file in an MC system that reads the input quantity only from the *AXIS_PTS* elements, no working point is displayed for these look-up tables.

Example The following excerpt shows an example representation of two 1-D look-up tables using a shared axis with different input signals in an A2L file:

```
/begin CHARACTERISTIC
   Sa1_Look_Up_Table_1_z_table    /* variable name */
   "speed"                        /* description */
   CURVE                          /* type */
   .........
   /begin AXIS_DESCR
      COM_AXIS                    /* axis type */
      Input_1                     /* input quantity */
      ...
      AXIS_PTS_REF LUT_X_table
   /end AXIS_DESCR
/end CHARACTERISTIC
```

```
/begin CHARACTERISTIC
   Sa1_Look_Up_Table_2_z_table    /* variable name */
   "speed"                        /* description */
   CURVE                          /* type */
   .........
   /begin AXIS_DESCR
      COM_AXIS                    /* axis type */
      Input_2                     /* input quantity */
      ...
      AXIS_PTS_REF LUT_X_table
   /end AXIS_DESCR
/end CHARACTERISTIC
```

```
/begin AXIS_PTS
   LUT_X_table                    /* name */
   "time"                         /* description */
   0x0000                         /* address */
   NO_INPUT_QUANTITY              /* input quantity */
   ...
/end AXIS_PTS
```

Related topics

Basics

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

References

Export as A2L File (TargetLink Data Dictionary Manager Reference)
Look-up Table (1-D) Block Object Dialog (TargetLink Data Dictionary Manager Reference)
Look-Up Table (2D) Block (TargetLink Model Element Reference)
Look-up Table (2-D) Block Object Dialog (TargetLink Data Dictionary Manager Reference)
Look-Up Table Block (TargetLink Model Element Reference)

Specifying the Build Object

Where to go from here

Information in this section

Basics on Specifying the Build Object

Build objects

Before you can export data as an A2L file, you have to provide information on the target for which the A2L file is to be generated. You have to specify information on the target in a **Build** object by providing the following files:

- Target Info file (`TargetInfo.xml`) that contains basic information on the target/compiler combination.
- Target config file (`TargetConfig.xml`) that contains information on basic data types.
- Optionally, a linker MAP file and assembler list files that contain ECU address information, if the A2L file is to contain address information.

Tip

TargetLink lets you export A2L files without providing a linker MAP file and assembler list files. In this case, however, the A2L file does not contain ECU address information. Many MC systems allow you to add ECU address information later.

Related topics

Basics

Examples

Details on Providing a Target Info file

Introduction

The target/compiler combination for which you want to generate an A2L file must be specified in a target Info file. This file is needed to create a DD **Build** object.

Purpose of the file

A target Info file is a XML file named `TargetInfo.xml` containing information for a specific target/compiler combination, such as the target processor name. It also contains information required by the MAP and list file parsers (file extensions).

Providing a target Info file

You have to provide a target Info file that contains the following elements and attributes, specifying your target/compiler combination:

Element	Attribute	Description
Target Processor Information		
`GeneralInfo`	`MyC`	CPU[1]
	`Compiler`	Compiler[2]
Target Compiler Information		
`BuildInfo`	`MapFileExtension`	Extension of linker MAP file
	`LstFileExtension`	File name extension of assembler list files

[1] The CPU name you specify must match the name in the **Target** column of the table in Details on Compilers Supported by the Symbol Table Import Module on page 135.

[2] The compiler name you specify must match the name in the **Abbreviation** column of the table in Details on Compilers Supported by the Symbol Table Import Module on page 135.

If you have the TargetLink Base Suite, you have a set of target Info files for TargetLink evaluation boards. There is one target Info file for each target/compiler combination in `<TL_InstRoot>\Matlab\Tl\ApplicationBuilder\BoardPackages\<Board>\<CompilerVersion>`.

The name of the `<CompilerVersion>` and `<Board>` subfolders must match the names in the Abbreviation and Board columns of the table in Details on Compilers Supported by the Symbol Table Import Module on page 135.

You can use the target Info files shipped with the TargetLink Base Suite.

Tip

If you do not have the TargetLink Base Suite or if you want to generate an A2L file for a target that is not supported by TargetLink, create a target Info file according to the table above:

```
<?xml version="1.0" ?>
<TargetInfo>
   <GeneralInfo Compiler="<TargetCompiler>" MyC="<Target>"/>
   <BuildInfo LstFileExtensions=".lst" MapFileExtension=".map"/>
</TargetInfo>
```

Related application note The *Generating TargetLink Code and ASAP2 Files for an Arbitrary Platform* application note demonstrates how to set up TargetLink if you want to generate code and ASAP2 files for a platform that is not supported by TargetLink's Target Simulation Module. Refer to www.dspace.com/go/TL_ArbitraryPlatforms.

Related topics

Basics

Details on Providing a Target Config File

Introduction

The target's basic data types must be specified in a target config file.

Purpose of the file

A target config file is an XML file named `TargetConfig.xml` containing information on the basic data types of the target/compiler combination such as the byte order, the alignment, etc.

Providing a target config file

You have to provide a target config file for the target for which you want to generate an A2L file.

If you have the TargetLink Base Suite, you have a set of target config files. There is one target config file for each target/compiler combination in `<TL_InstRoot>\Matlab\Tl\TargetConfiguration\<MicrocontrollerFamily>\<CompilerFamily>`.

Tip

If you want to export an A2L file for a target/compiler combination that is not supported by TargetLink, you can use the `TargetConfigTemplate.xml` file as template. This file is placed in `<TL_InstRoot>\Matlab\Tl\TargetConfiguration`.

Related application note The *Generating TargetLink Code and ASAP2 Files for an Arbitrary Platform* application note demonstrates how to setup TargetLink if you want to generate code and ASAP2 files for a platform that is not supported by TargetLink's Target Simulation Module. Refer to www.dspace.com/go/TL_ArbitraryPlatforms.

Related topics

Basics

Basics on TargetLink Base Types, Typedefs, and Header Files (TargetLink Customization and Optimization Guide)
Details on Providing a Target Info file.. 132

Examples

Example of Controlling the Generation of the File Containing TargetLink Base Data Types (TargetLink Customization and Optimization Guide)

Details on Providing a Linker MAP File and Assembler List Files

Introduction

To include ECU address information in the A2L file to be generated, a linker MAP file is required by the A2L export module. For some target/compiler combinations, assembler list files are also required.

Purpose of the file

To access calibratable and measurable variables of the ECU application from within an MC system, the addresses of the variables need to be included in the application's A2L file. The Symbol Table import module of the TargetLink Data Dictionary extracts ECU address information from the linker MAP file and also, for some target/compiler combinations, from assembler list files.

Note

The Symbol Table import module can extract ECU address information from the linker MAP file and the assembler list files only if they were created by one of the supported compilers. For a list of these compilers, refer to Details on Compilers Supported by the Symbol Table Import Module on page 135.

The module saves the information in the **Build** object and makes it accessible for the A2L export module.

Providing a linker MAP file

A linker MAP file and assembler list files are generated when you build the ECU application.

Tip

TargetLink lets you export A2L files without providing a linker MAP file and assembler list files. In this case, however, the A2L file does not contain ECU address information. Many MC systems allow you to add ECU address information later.

Related topics

Basics

Details on Compilers Supported by the Symbol Table Import Module

Extracting ECU address information

The following table lists compilers supported for specific targets. It shows:

- Whether ECU address information can be extracted from a linker MAP file.
- Whether an assembler list file is required to allow the export module to extract ECU address information.
- Whether additional files are required.
- Which configuration file to use.

Tip

For further support combinations that are part of a valid PIL Software Maintenance Service (SMS) contract, refer to www.dspace.com/go/tlpil.

Target	Compiler	Abbreviation	Board	Required Files		
				Linker Map File	Assembler List File	Configuration File
V850E2	Green Hills V8xx C Compiler	GHS51 GHS53 GHS2012 GHS2013 GHS2014 GHS2015 GHS2016 GHS2017 GHS2018 GHS2019	AB_050_Fx4_70F4012	True	True	v850e2_ghs_Config.xml
SH2	Renesas SH Series C Compiler	HIT60 HIT70 HIT80 HIT90 HIT91 HIT93 HIT94	EVB7058	True	False	sh2_hit60_Config.xml
S12X	Cosmic C Cross-Compiler	COSMIC46 COSMIC47 COSMIC48	EVB9S12XEP100	True	True	s12x_cosmic_Config.xml
	Metrowerks C Compiler CodeWarrior for MC9S12X	Metrowerks41 Metrowerks45 Metrowerks46 Metrowerks47 Metrowerks50 Metrowerks51	EVB9S12XEP100	True	False	hcs12_metrowerks_Config.xml
I86	GNU GCC C Compiler	GCC	HostPC32	True	False	i86_GCC_Config.xml
	Microsoft Visual C/C++ Compiler	MSVC	HostPC32	True	False	i86_MSVC_Config.xml
x86_64	GNU GCC C Compiler	GCC	HostPC64	True	False	i86_GCC_Config.xml
	Microsoft Visual C/C++ Compiler	MSVC	HostPC64	True	False	x64_MSVC_Config.xml
CORTEXR5F	Texas Instruments CCS C Compiler	CCS7	LAUNCHXL2570LC43	True	False	cortexr5f_ccs_Config.xml
CortexM3	Keil C Compiler	Keil41 Keil47 Keil51 Keil52	LBSIM_ARM	True	False	cortexm3_keil_Config.xml

Target	Compiler	Abbreviation	Board	Required Files		
				Linker Map File	Assembler List File	Configuration File
CortexM3	Keil C Compiler	Keil41 Keil47 Keil51 Keil52	MEDKit_ARM	True	False	cortexm3_keil_Config.xml
TriCore275	AltiumTasking TriCore C Cross-Compiler	Task41 Task42 Task43 Task50 Task60 Task61 Task62 Task63	LBSIM_TC275	True	True	tricore_task2_Config.xml
	AltiumTasking TriCore C Cross-Compiler	Task41 Task42 Task43 Task50 Task60 Task61 Task62 Task63	TBTC275	True	True	tricore_task2_Config.xml
	GNU GCC CCompiler	GNU46 GNU49	TBTC275	False	False	SymTableElfParser_Config2.xml
MPC5xxxVLE	Wind River Diab PowerPC Family C Compiler	Diab59	MPC5748GEVB	False	False	SymTableElfParser_Config2.xml
	Green Hills MPC55xx C Compiler	GHS50 GHS51 GHS52 GHS2012 GHS2013 GHS2014 GHS2015 GHS2016 GHS2017 GHS2018 GHS2019	MPC5748GEVB	True	True	mpc55xx_ghs_Config.xml
C16x	Altium Tasking 80166 C Cross-Compiler	TASK51 TASK60 TASK70 TASK75 TASK80 TASK85 TASK86 TASK87	Promo167	True	True	c16x_task_Config.xml

Target	Compiler	Abbreviation	Board	Required Files		
				Linker Map File	Assembler List File	Configuration File
SH2AFPU	Renesas C Compiler	Renesas90 Renesas91 Renesas93 Renesas94	SDK72513	True	False	sh2a_renesas90_Config.xml
XC2287	Altium Tasking VX 3.0	TASKINGVX30	SK_EBXC2287	False	False	XC2287_TaskingVX21_Config.xml
TriCore1766	AltiumTasking TriCore C Cross-Compiler	TASK25 TASK32 TASK35 TASK40 TASK42 TASK43	TBTC1766	True	True	tricore_task2_Config.xml
TriCore1766	AltiumTasking TriCore C Cross-Compiler	TASK25 TASK32 TASK35 TASK40 TASK42 TASK43	TBTC1766_20MHz	True	True	tricore_task2_Config.xml
TriCore1767	AltiumTasking TriCore C Cross-Compiler	TASK25 TASK32 TASK35 TASK40 TASK42 TASK43 Task50 Task60 Task61 Task62	TBTC1767	True	True	tricore_task2_Config.xml
TriCore1796	AltiumTasking TriCore C Cross-Compiler	TASK25 TASK32 TASK35 TASK40 TASK42 TASK43	TBTC1796	True	True	tricore_task2_Config.xml
RH850F1L	Green Hills RH850 C Compiler	GHS2013 GHS2014 GHS2015 GHS2016 GHS2017 GHS2018 GHS2019	YRH850F1L_R7F7010354	False	False	rh850f1l_ghs_Config.xml

Related topics

Basics

Basics on Obtaining Addresses of Variables and Functions (Symbol Table) (📖 TargetLink Preparation and Simulation Guide)
Details on Providing a Linker MAP File and Assembler List Files.. 134
Specifying the Build Object.. 131

How to Create and Specify the Build Object

Objective

You have to create and specify a build object containing information on the target for which the A2L file is to be generated.

Restrictions

The **Manage Build** dialog used to create and specify the build object is available only if MATLAB is installed.

Preconditions

The following preconditions apply:

- The target Info file specifying the target/compiler combination for which you want to generate an A2L file is available. Refer to Details on Providing a Target Info file on page 132.
- The target config file specifying the target's basic data types is available. Refer to Details on Providing a Target Config File on page 133.
- To include ECU address information in the A2L file, a linker MAP file must be available. Refer to Details on Providing a Linker MAP File and Assembler List Files on page 134.

Tip

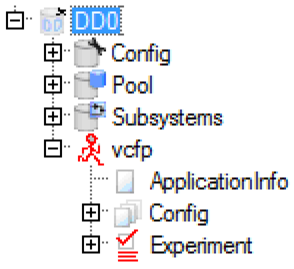
TargetLink lets you export A2L files without providing a linker MAP file and assembler list files. In this case, however, the A2L file does not contain ECU address information. Many MC systems allow you to add ECU address information later.

Method

To create and specify the Build object

1 Open your DD project file, e.g., `vcfp.dd`.

If your DD project file does not contain an application object, create and rename one as required.



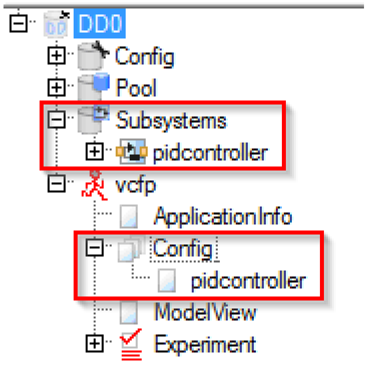
2 Right-click the Config object contained in the application object and select Create SubsystemConfig from the context menu.

A new DD SubsystemConfig object is created.

Note

DD SubsystemConfig objects are necessary if the created Build object has to contain address information.
The Symbol Table import module writes only those symbols to the DD Build object that are used or defined in the modules listed under the subsystem objects corresponding to the specified DD SubsystemConfig objects.

3 Give the DD SubsystemConfig object the same name as the subsystem object, e.g., `pidcontroller`.



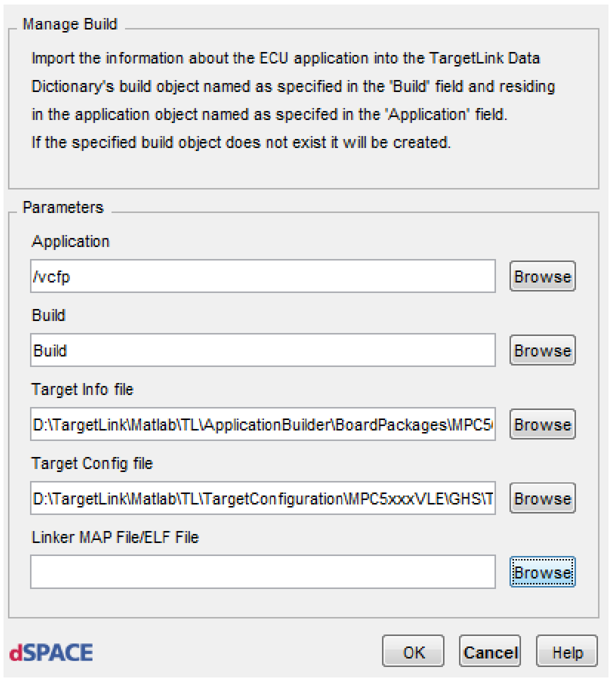
4 Repeat steps 2 and 3 for each subsystem object for which an A2L file is to be generated.

Tip

When you generate code with TargetLink, a DD Application object and a DD SubsystemConfig object are created automatically.

5 On the Data Dictionary Manager's **Tools** menu, click **Manage Build**.
The **Manage Build** dialog opens.

6 In the dialog, specify the following:
- The application object for which the build object is to be created
- The name of the build object to be created
- The target Info file
- The target config file
- The linker MAP file (if address information is to be imported into the created build object)



Manage Build

Import the information about the ECU application into the TargetLink Data Dictionary's build object named as specified in the 'Build' field and residing in the application object named as specifed in the 'Application' field.
If the specified build object does not exist it will be created.

Parameters

Application: /vcfp

Build: Build

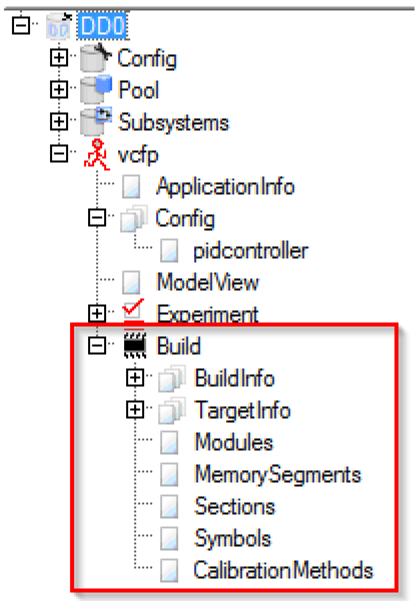
Target Info file: D:\TargetLink\Matlab\TL\ApplicationBuilder\BoardPackages\MPC5

Target Config file: D:\TargetLink\Matlab\TL\TargetConfiguration\MPC5xxxVLE\GHS\T

Linker MAP File/ELF File:

7 Click **OK**.

The new DD **Build** object is created.



Result

You created and specified a DD **Build** object.

Tip

To create and specify DD **Build** objects, you can also use the **`dsdd_manage_build`** API function.

Related topics

Basics

References

Create <DD Object> (📖 TargetLink Data Dictionary Manager Reference)
dsdd_manage_build (📖 TargetLink API Reference)
Manage Build (📖 TargetLink Data Dictionary Manager Reference)

Specifying the ASAP1b Interface

Where to go from here

Information in this section

Basics on ASAP1b Interfaces

Communication layer between ECU and MC system

The ASAP1b interface, also called the ASAP1b interface, defines the communication layer between an ECU and a measurement and calibration (MC) system connected to the ECU.

Specifying the interface for A2L file generation

The ASAP1b interface holds information on the memory structure of the ECU and the initialization of the communication protocol. You have to specify the ASAP1b interface to include this specification in the A2Lfile to be generated.

Supported ASAP1b interfaces

The A2L export module supports the following ASAP1b interfaces:

- ADDRESS
- CCP
- DIM
- DCI_GME1
- DCI_GSI1
- XCP
- CANAPE_EXT
- ETK

You can specify further ASAP1b interfaces not listed above. This allows you to include them in the generated A2L file. Refer to How to Support Another ASAP1b Interface on page 176.

Specifying the interface via IF_DATA blocks

In the A2L file, the ASAP1b interface is specified via `IF_DATA` blocks that contain interface-specific description data and parameters that are described via the ASAM MCD-2 MC metalanguage (AML).

The following table shows which A2L elements can have `IF_DATA` blocks and whether user interaction is required to generate them in the A2L file:

A2L Element Containing IF_DATA	Instruction
`AXIS_PTS`	Automatically generated during A2L generation.
`CHARACTERISTIC`	
`MEASUREMENT`	
`MODULE`	Specified via a DD IFData object. Refer to Specifying the MODULE IF_DATA block on page 144.

Specifying the MODULE IF_DATA block

Elements of the MODULE IF_DATA block TargetLink lets you generate `MODULE IF_DATA` block elements that contain keyword blocks and parameter entries. The following table shows an example:

Keyword Block	Parameter Entry

`/begin <KEYWORD>`
`   <Value>   /* <ParameterComment> */`
`   …`
`/end <KEYWORD>`
	`<PARAMETER>   [<VALUE>]`

Both can be nested:	

`/begin <KEYWORD1>`
`   <Value>   /* <ParameterComment> */`
`   /begin <KEYWORD2>`
`      …`
`   /end <KEYWORD2>`
`   …`
`/end <KEYWORD1>`
	`<PARAMETER1>`
	`   <PARAMETER2>`
	`   <PARAMETER3>`
	`   <PARAMETER4>  <Value1>  <Value2>`

Both can be mixed:	

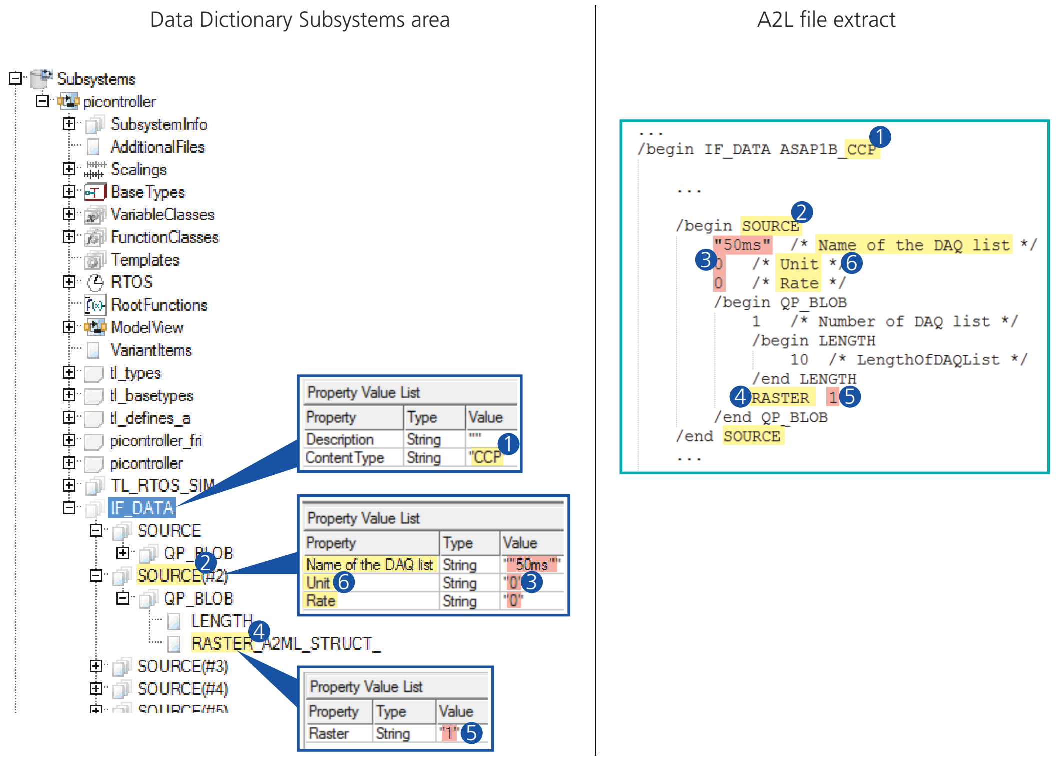
```
/begin <KEYWORD1>
   <Value>   /* <ParameterComment> */
   /begin <KEYWORD2>
      …
      <PARAMETER1>
         <PARAMETER2>
         <PARAMETER3>
         <PARAMETER4>  <Value1>  <Value2>
      …
   /end <KEYWORD2>
   …
/end <KEYWORD1>
```

DD objects representing A2L elements The following table shows the mapping of A2L elements to DD entities:

A2L Element	DD Entity
Keyword block	DD Generic object with `n` custom string property/value pairs.
Parameter entry	DD Generic object whose name contains `_A2ML_STRUCT_` and that has one custom string property for each value.

Example

The following illustration shows an extract from the `MODULE IF_DATA` block of the CCP ASAP1b interface:



Property Value List		
Property	Type	Value
Description	String	""
ContentType	String	"CCP"

Property Value List		
Property	Type	Value
Name of the DAQ list	String	""50ms""
Unit	String	"0"
Rate	String	"0"

Property Value List		
Property	Type	Value
Raster	String	"1"

```
...
/begin IF_DATA ASAP1B_CCP

    ...

    /begin SOURCE
        "50ms"   /* Name of the DAQ list */
        0    /* Unit */
        0    /* Rate */
        /begin QP_BLOB
            1   /* Number of DAQ list */
            /begin LENGTH
                10  /* LengthOfDAQList */
            /end LENGTH
            RASTER  1
        /end QP_BLOB
    /end SOURCE
    ...
```

Number	DD Object/Property Kind	A2L Element	Description
❶	DD IFData object with predefined string properties	Keyword block	The value of the ContentType property is used as the suffix of the `IF_DATA` keyword. If its value matches the ASAP1b interface provided via the export dialog or the API, TargetLink uses this object to generate the contents of the `IF_DATA` block. Otherwise, it is ignored. If no DD IFData object with a matching content type exists, TargetLink generates an empty `IF_DATA` block.
❷	DD Generic object whose name does not contain the `_A2ML_STRUCT_` suffix and that has custom string properties	Keyword block	The object's name is used as the keyword of a keyword block in the A2L file.
❸	Values of different custom string properties	Value	Each value is used as the value of a parameter of the A2L keyword's block.
❹	DD Generic object whose name contains the `_A2ML_STRUCT_` string and that contains exactly one custom string property	Parameter	The name without the `_A2ML_STRUCT_` suffix is used as the name of the parameter in the A2L file. The parameter is not provided as in keyword blocks. Because the DD Generic object is a child of the QP_BLOB object, the parameter occurs within the `QP_BLOB` keyword's block.
❺	The value of a custom string property	Value	The value of the custom string property is taken as the parameter's value in the A2L file.

Number	DD Object/Property Kind	A2L Element	Description
6	Custom string property	Parameter comment	The name of the custom string property is used as the parameter comment.

Related topics

Basics

HowTos

How to Create and Specify an IFData Object for a MODULE IF_DATA Block

MODULE elements

`MODULE` elements in the A2L file result from code generation units (CGUs). After code generation, each CGU is represented as a DD **Subsystem** object in the Data Dictionary's **Subsystems** area.

Preconditions

The following preconditions must be fulfilled:

- You specified the data to be generated in your A2L file and generated code. Refer to Specifying the Data to Export and Generating Code on page 120.
- You read Specifying the MODULE IF_DATA block on page 144.

Workflow

The workflow for creating and specifying a DD **IFData** object for DD **Subsystem** objects consists of the following steps:

- Creating the **IFData** object.

  Refer to Part 1 on page 147.
- Creating the specification of A2L `IF_DATA` keyword blocks.

  Refer to Part 2 on page 148.
- Creating the specification of A2L parameter entries.

  Refer to Part 3 on page 148.

Tip

It is sufficient to specify the `IF_DATA` element at only one DD Subsystem object if the following conditions are fulfilled:

- You selected the Merge A2L modules checkbox in the Export as A2L File dialog or set the MergeA2LModules property of the `dsdd_export_a2l_file` API function to `on`.
- You selected multiple DD Subsystem objects for A2L file generation.

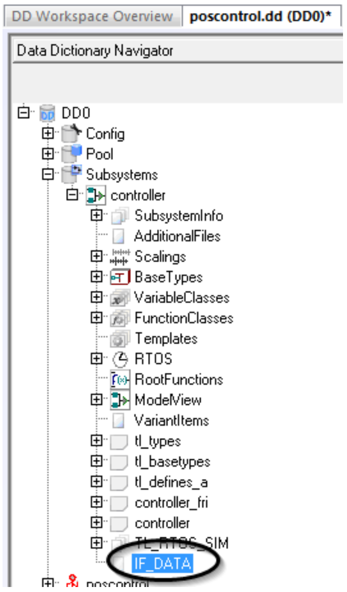
Part 1

To create the IFData object

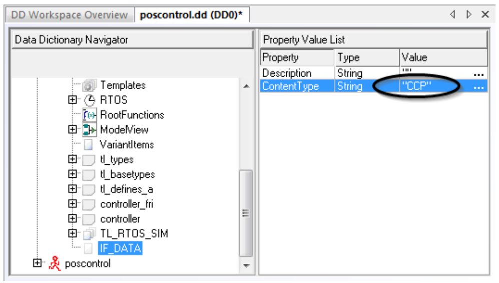
1 In the Data Dictionary Manager, expand the Subsystems object.

2 Locate the DD Subsystem object whose DD IFData object you want to create.

3 From its context menu, select Create IF_DATA. TargetLink creates a DD IFData object.



4 Set its ContentType property to `<ASAP1b interface>, e.g., CCP.`



Property	Type	Value
Description	String	""
ContentType	String	"CCP"

Interim result

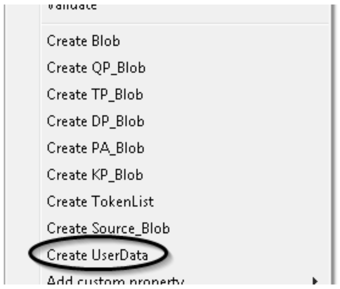
You created the DD Subsystem object's IFData object and specified the name of the ASAP1b interface.

You must now create one DD Generic object for each keyword block in the A2L file.

Part 2

To create the specification of A2L IF_DATA keyword blocks

1 From the context menu of the DD IFData object, select Create UserData.



TargetLink creates a DD Generic object as child object of the IFData object.

2 Rename the object to reflect the A2L `IF_DATA` keyword's name, e.g., `SOURCE`.

3 From the DD Generic object's context menu, select Add custom Property - String to specify the A2L `IF_DATA` keyword block's parameters. Add and adapt as many properties as required.

The name of each property results in a parameter comment in the corresponding `IF_DATA` keyword block of the A2L file. The value of each property results in the corresponding parameter value.

4 Repeat this part for each keyword block you want to specify. To nest keyword blocks, create DD Generic objects as child objects.

Interim result

You created the specification for one or more keyword blocks. You can now create more parameters that are not represented as keyword blocks but as parameter entries.

Part 3

To create the specification of A2L parameter entries

1 Select the DD Generic object that represents the A2L `IF_DATA` keyword block to that you want to add a parameter entry.

2 From its context menu, select Create UserData. TargetLink adds a DD Generic object to the object's subtree.

3 Rename the object to reflect the parameter's name and append the following suffix: `_A2ML_SRUCT_`, e.g., **`RASTER_A2ML_STRUCT_`**.

4 From the context menu, select Add custom Property - String to specify the A2L parameter entry.

The value of this property is used as the parameter entry's value. The property name is ignored.

5 Repeat this part for each parameter entry that you want to generate in the A2L file's `IF_DATA` block.

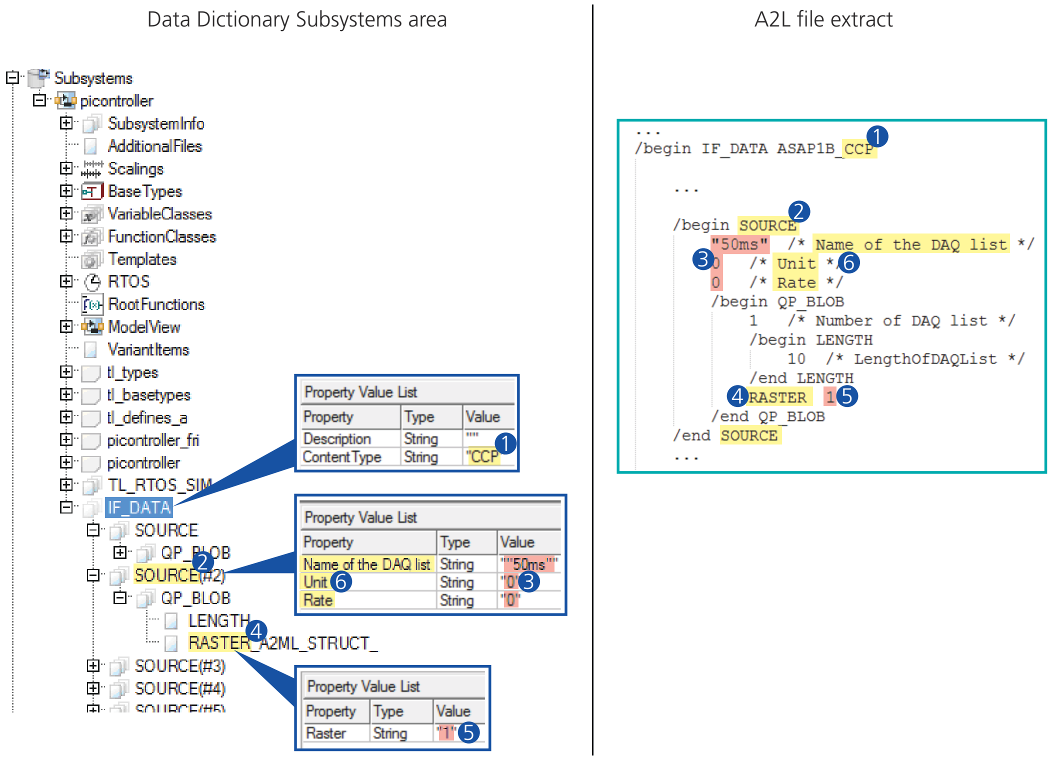
6 Save your project file.

Tip

You can export the **IFData** object to an XML file for later import. This facilitates tool chain automation. Refer to Exchanging XML Files on page 194.

Result

You created the specification of an `IF_DATA` block in the Data Dictionary. The following illustration shows an example:



Property Value List

Property	Type	Value
Description	String	""
ContentType	String	"CCP"

Property Value List

Property	Type	Value
Name of the DAQ list	String	"""50ms"""
Unit	String	"0"
Rate	String	"0"

Property Value List

Property	Type	Value
Raster	String	"1"

```
...
/begin IF_DATA ASAP1B_CCP

    ...

    /begin SOURCE
        "50ms"  /* Name of the DAQ list */
        0    /* Unit */
        0    /* Rate */
        /begin QP_BLOB
            1    /* Number of DAQ list */
            /begin LENGTH
                10  /* LengthOfDAQList */
            /end LENGTH
            RASTER  1
        /end QP_BLOB
    /end SOURCE
    ...
```

Related topics

Basics

References

Create <DD Object> (📖 TargetLink Data Dictionary Manager Reference)
dsdd_export_a2l_file (📖 TargetLink API Reference)
Export as A2L File (📖 TargetLink Data Dictionary Manager Reference)

Specifying the Conversion of Variable Values

Where to go from here

Information in this section

Basics on Conversion Methods for A2L Generation

Available conversion methods

In an A2L file, conversion methods are defined by the `COMPU_METHOD` elements that are referenced by `CHARACTERISTIC`, `MEASUREMENT`, and `AXIS_PTS` elements.

You specify `COMPU_METHOD` elements via DD Scaling objects. This lets you specify how the values of the calibratable and measurable variables are represent in the Measurement and Calibration (MC) system, e.g., as strings instead of numerical values.

Note

For code generation, TargetLink always uses linear conversion with LSB and offset.

You can use the following conversion methods as the value of the DD Scaling object's ConversionType property for A2L generation:

Conversion Method	Description
`FORM`	Represents the variable's internal value `x` in the ECU memory via an arbitrary formula.
`IDENTICAL`	Represents the variable's internal value x in the ECU memory via the linear conversion function $f(x) = x$.
`LINEAR`	Represents the variable's internal value `x` in the ECU memory via the linear conversion function $\mathrm{f(x)} = LSB{*}x + Offset$. For example, with $LSB = 2^{-11}$ and $Offset = 0$, the internal value 205 is represented by 0.1. (default)
`RAT_FUNC`	Represents the variable's physical value `x` via the fractional ration function $\mathrm{f}(x) = \left(a{*}\,x^2 + b{*}x + c\right)/\left(d{*}x^2 + e{*}x + f\right)$, which corresponds to the notation $(a,\ b,\ c,\ d, e\ , f)$.
`TAB_INTP`	Represents the variable's internal value `x` in the ECU memory as other numerical value (with interpolation). Used for conversions that cannot be represented as functions. For example, with `[1 10;2 20]` as the conversion table, `1` is represented as `10`, `2` as `20`, and `1.2` as `12` (interpolated value).
`TAB_NOINTP`	Represents the variable's internal value `x` in the ECU memory as other numerical value (without interpolation). Used for conversions that cannot be represented as functions. For example, with `[1 10;2 20]` as the conversion table, `1` is represented as `10`, `2` as `20`, and `1.2` as `10` (nearest value).
`TAB_VERB`	Represents the variable's internal value `x` in the ECU memory as a string. For example, with `(0 1)` as the conversion table and `["off","on"]` as conversion strings, `0` is represented as `off` and `1` as `on`.

Specifying the name string

You can specify the `Name` of the `COMPU_METHOD` elements by using DD Scaling objects and naming them as required.

If *both* of the following conditions hold, several different `COMPU_METHOD` elements are derived from the same DD Scaling object.

- It's ConversionType property is set to `LINEAR`.
- Its LSB and/or Offset properties have vectorial values whose elements are not identical.

TargetLink then creates a `COMPU_METHOD` element for each vector element and appends a suffix to the `COMPU_METHOD` element's `Name` string to avoid ambiguities.

Specifying the format string

The `COMPU_METHOD` elements of A2L files have a format string that determines how the value of a variable is displayed in the measurement and calibration (MC) system. This format string has the following schema:

Schema of A2l Format String
`%[<length>].<layout>`

`<length>` is an optional unsigned integer value which indicates the overall length. `<layout>` is a mandatory unsigned integer value which indicates the decimal places.

Customizing the format string By default, TargetLink determines the correct format string from the variable's LSB, offset, and data type.

You can customize the format string via the Format property of DD Scaling and Variable objects. Enter a string value that complies with the schema.

When generating A2L files via the `dsdd_export_a2l_file` or `dsdd('Export', 'Format','A2L', ...)` API functions, you can also provide the format string in C `printf` notation, if you set the respective API function's AllowCFormatSpecifiers property to `on`.

Obtaining NO_COMPU_METHOD for unscaled variables

To generate A2L file entries for variables that do not have scaling information (as `COMPU_METHOD` in the A2L file), make sure that the variable references the DD VOID_SCALING object. TargetLink will then generate `NO_COMPU_METHOD` elements for these variables:

```
/begin CHARACTERISTIC
   Table_Values   /* Name */
   ""   /* LongIdentifier */
   CURVE   /* Type */
   0x0000   /* Address */
   SWORD_COL_DIRECT   /* Deposit */
   0   /* MaxDiff */
   NO_COMPU_METHOD   /* Conversion */
   -32768   /* LowerLimit */
   32767   /* UpperLimit */
   BYTE_ORDER MSB_FIRST
   SYMBOL_LINK "Table_Values" 0
      /begin AXIS_DESCR
         COM_AXIS   /* Attribute */
         NO_INPUT_QUANTITY   /* InputQuantity */
         NO_COMPU_METHOD   /* Conversion */
         3   /* MaxAxisPoints */
         -32768   /* LowerLimit */
         32767   /* UpperLimit */
         BYTE_ORDER MSB_FIRST
         AXIS_PTS_REF Axis_1
      /end AXIS_DESCR
/end CHARACTERISTIC
```

Related topics

Basics

Integer Representation of Physical Quantities (📖 TargetLink Preparation and Simulation Guide)

HowTos

References

dsdd_export_a2l_file (📖 TargetLink API Reference)

How to Specify a Verbal Conversion Table

Objective

You must specify a verbal conversion table to represent the values of a variable as a string in an MC system.

Representing values

You can specify a verbal conversion table: for example, to represent the values of a Boolean variable as `OFF` or `ON` in the MC system. You must specify a corresponding (converted) value for each value of the variable in the ECU memory.

DD Scaling object

To specify a conversion table, you have to create a scaling object in the TargetLink Data Dictionary and specify its properties. Properties that affect the conversion – `ConversionType`, `ConversionStrings`, `ConversionTable` – are evaluated during A2L file generation.

Note

For code generation with TargetLink, you also need to specify the LSB and offset.

Method

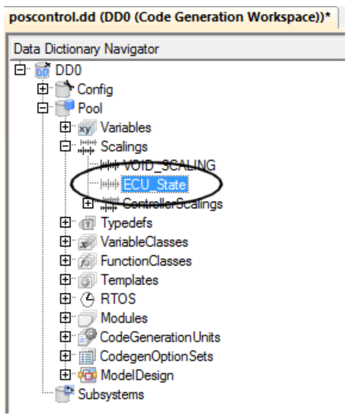
To specify a verbal conversion table

1 Open your Data Dictionary project file.

2 In the Pool area of the Data Dictionary Navigator, right-click Scalings, then click Create Scaling from the context menu.

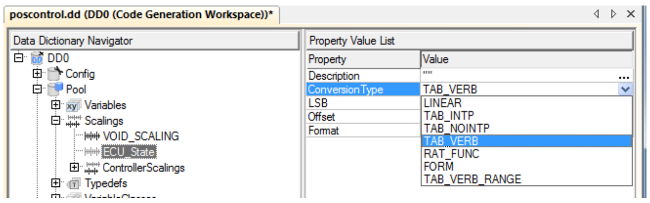
The Data Dictionary creates a new Scaling object.

3 Name the new Scaling object according to your requirements.



4 In the property value list of the new scaling object,

- Select TAB_VERB as the ConversionType.



poscontrol.dd (DD0 (Code Generation Workspace))*

Data Dictionary Navigator
- DD0
  - Config
  - Pool
    - Variables
    - Scalings
      - VOID_SCALING
      - ECU_State
      - ControllerScalings
    - Typedefs

Property Value List

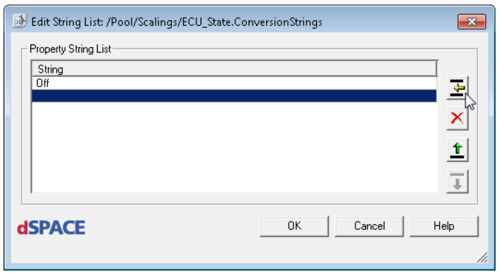
Property	Value
Description	""
ConversionType	TAB_VERB
LSB	
Offset	
Format	

LINEAR
TAB_INTP
TAB_NOINTP
TAB_VERB
RAT_FUNC
FORM
TAB_VERB_RANGE

- Specify LSB and Offset according to your requirements.

5 In the value field of the ConversionStrings property, click Browse.

The Edit StringList dialog opens.



Edit String List: /Pool/Scalings/ECU_State.ConversionStrings

Property String List

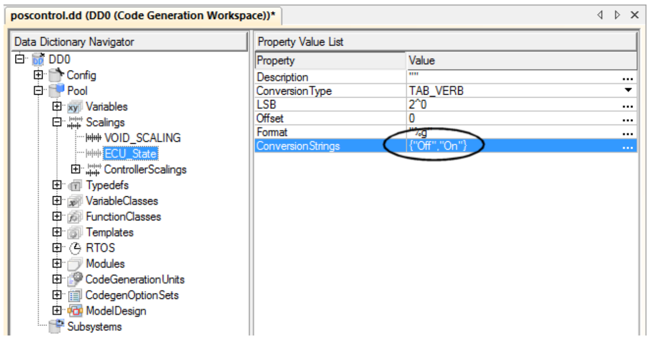
String
Off

6 In the dialog, click the topmost button on the right, then enter a string for your conversion table.

7 Repeat step 6 for each string of the conversion table.

8 In the Edit StringList dialog, click OK.

If you entered the strings `OFF` and `ON` in the Edit StringList dialog, the property value list of the `ConversionStrings` property looks like this:



poscontrol.dd (DD0 (Code Generation Workspace))*

Data Dictionary Navigator
- DD0
  - Config
  - Pool
    - Variables
    - Scalings
      - VOID_SCALING
      - ECU_State
      - ControllerScalings
    - Typedefs
    - VariableClasses
    - FunctionClasses
    - Templates
    - RTOS
    - Modules
    - CodeGenerationUnits
    - CodegenOptionSets
    - ModelDesign
  - Subsystems

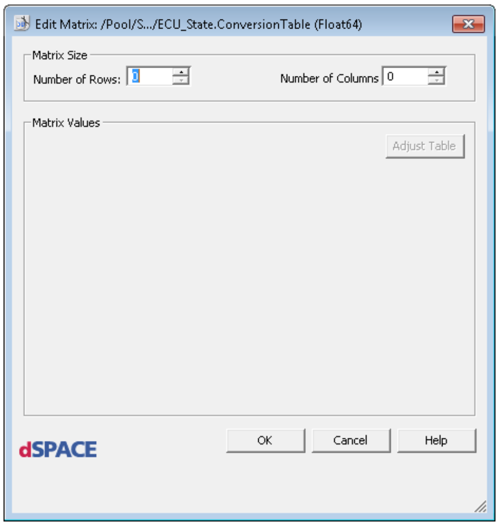
Property Value List

Property	Value
Description	""
ConversionType	TAB_VERB
LSB	2^0
Offset	0
Format	"%g"
ConversionStrings	{"Off","On"}

Each string you entered must have a corresponding value of the variable in the ECU memory. This value must be specified for the `ConversionTable` property.

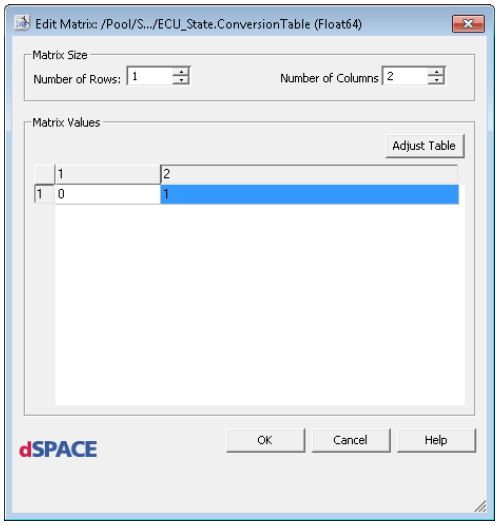
9 In the value field of the ConversionTable property, click Browse.

The Edit Matrix dialog opens.



10 In the dialog,

- Set the Number of Rows property to `1` and specify the Number of Columns property as required.
- Enter the integer values corresponding to the strings specified in the String Array dialog of the `ConversionStrings` property. For example, if you entered the string values `OFF` and `ON` for the `ConversionStrings` property, enter `0` and `1`.


Edit Matrix: /Pool/S.../ECU_State.ConversionTable (Float64)

Matrix Size

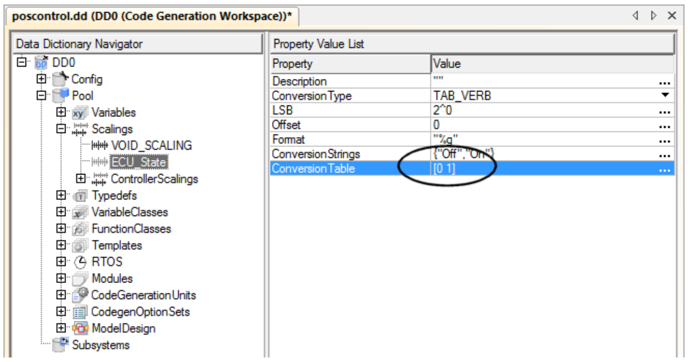
Number of Rows: 1   Number of Columns 2

Matrix Values

	1	2
1	0	1

11 In the Edit Matrix dialog, click OK.

The values `0` and `1` are entered in the property value list of the `ConversionTable` property:


poscontrol.dd (DD0 (Code Generation Workspace))*

Property Value List

Property	Value
Description	""
ConversionType	TAB_VERB
LSB	2^0
Offset	0
Format	"%g"
ConversionStrings	{"Off","On"}
ConversionTable	[0 1]

12 On the Data Dictionary Manager’s File menu, click Save, or press `Ctrl` + `S`.

The TargetLink Data Dictionary stores the new scaling object to the DD project.

Result

You specified a verbal conversion table in a scaling object. With this scaling object, the values `0` and `1` are represented by the verbal expressions `OFF` and `ON`, respectively. You can create further scaling objects with more complex verbal conversions.

Related topics

Basics

Basics on the TargetLink Data Dictionary (TargetLink Data Dictionary Basic Concepts Guide)
Specifying the Conversion of Variable Values.. 150

HowTos

How to Specify a Numeric Conversion Table.. 157

References

Create <DD Object> (TargetLink Data Dictionary Manager Reference)
Edit Matrix (TargetLink Data Dictionary Manager Reference)
Edit StringList (TargetLink Data Dictionary Manager Reference)
Save (TargetLink Data Dictionary Manager Reference)

How to Specify a Numeric Conversion Table

Objective

To represent the values of a variable as specific (numeric) values in an MC system, you must specify a numeric conversion table by specifying a corresponding (converted) value for each value of the variable in the ECU memory.

DD Scaling object

To specify a conversion table, you have to create a scaling object in the TargetLink Data Dictionary and specify it. Properties that affect the conversion – `ConversionType`, `ConversionTable` – will be evaluated during A2L file generation.

Note

For code generation with TargetLink, you also need to specify the LSB and offset.

Method

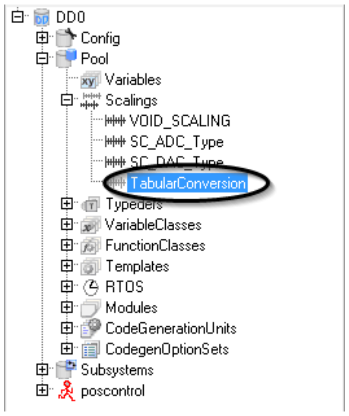
To specify a numeric conversion table

1 Open your Data Dictionary project file.

2 In the Pool area of the Data Dictionary Navigator, right-click Scalings, then click Create Scaling from the context menu.

The Data Dictionary creates a new scaling object.

3 Name the new Scaling object according to your requirements.

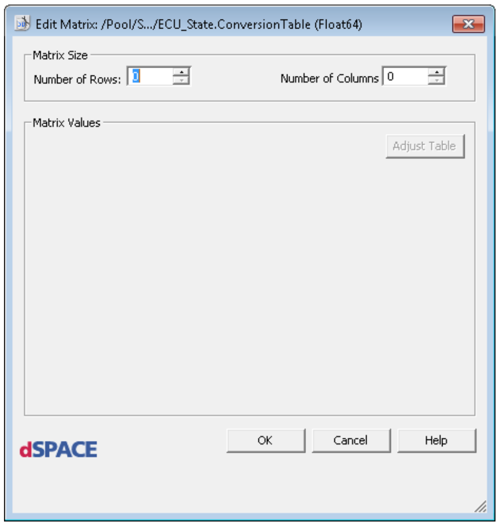


4 In the property value list of the new scaling object,

- Select `TAB_INTP` or `TAB_NOINTP` as the ConversionType.
- Specify `LSB` and `Offset` according to your requirements.

5 In the value field of the `ConversionTable` property, click Browse.

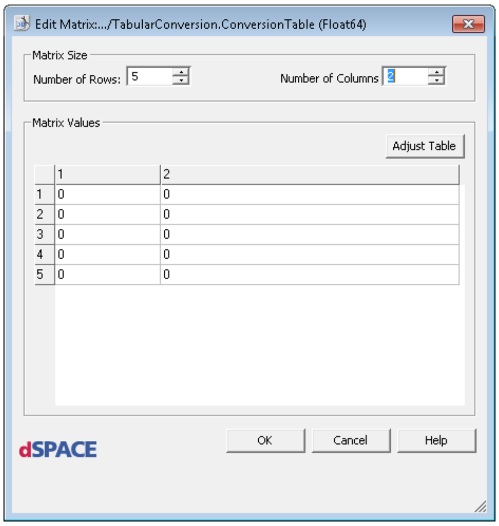
The Edit Matrix dialog opens.



6 In the dialog,

- Specify 2 in the Number of columns edit field, since you need 2 columns to enter the (integer) values of the variable and the corresponding (converted) values.

- Specify the number of value pairs of the conversion table in the **Number of rows** edit field.


Edit Matrix:.../TabularConversion.ConversionTable (Float64)

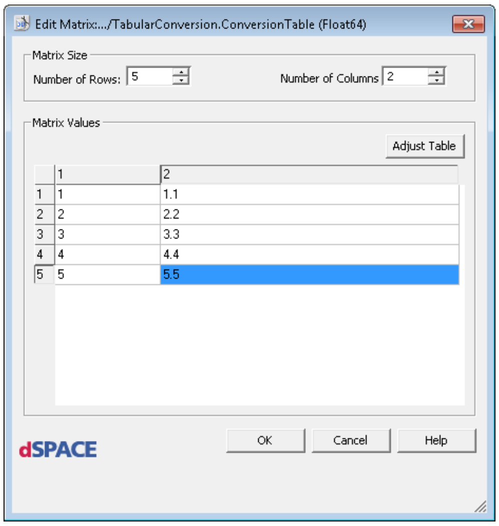
Matrix Size

Number of Rows: 5 Number of Columns 2

Matrix Values

	1	2
1	0	0
2	0	0
3	0	0
4	0	0
5	0	0

7 In the **Matrix Values** pane of the **Edit Matrix** dialog, enter the (integer) values of the variable in the left column, and the corresponding (converted) values in the right column.


Edit Matrix:.../TabularConversion.ConversionTable (Float64)

Matrix Size

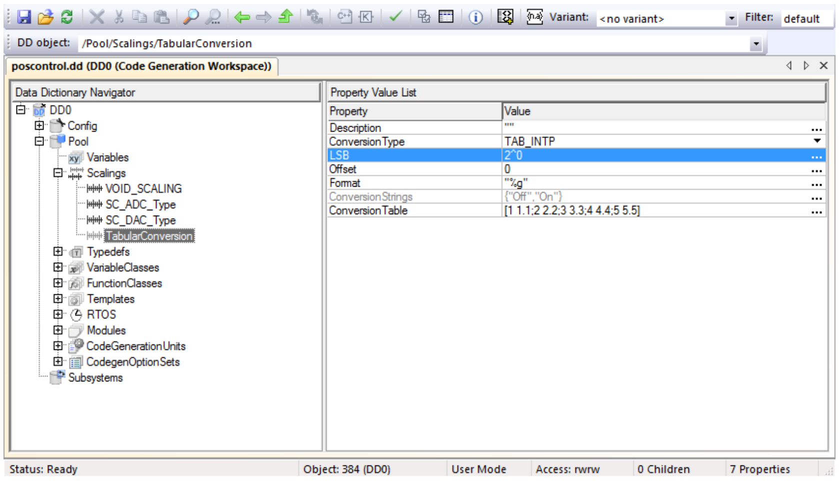
Number of Rows: 5 Number of Columns 2

Matrix Values

	1	2
1	1	1.1
2	2	2.2
3	3	3.3
4	4	4.4
5	5	5.5

8 In the **Edit Matrix** dialog, click **OK**.

The original values and the converted values are entered in the property value list of the `ConversionTable` property.


Variant: <no variant> Filter: default

DD object: /Pool/Scalings/TabularConversion

poscontrol.dd (DD0 (Code Generation Workspace))

Property Value List

Property	Value
Description	""
ConversionType	TAB_INTP
LSB	2^0
Offset	0
Format	"%g"
ConversionStrings	{"Off","On"}
ConversionTable	[1 1.1;2 2.2;3 3.3;4 4.4;5 5.5]

Status: Ready | Object: 384 (DD0) | User Mode | Access: rwrw | 0 Children | 7 Properties

9 On the Data Dictionary Manager's File menu, click Save, or press **`Ctrl`** + **`S`**.

The TargetLink Data Dictionary stores the new scaling object to the DD project.

Result

You specified a numeric conversion table in a scaling object. With this scaling object, the (original) values 1 … 5 will be represented by the (converted) values 1.1 … 5.5. You can create further scaling objects with more complex numeric conversions.

Related topics

Basics

Basics on the TargetLink Data Dictionary (📖 TargetLink Data Dictionary Basic Concepts Guide)
Specifying the Conversion of Variable Values.. 150

HowTos

How to Specify a Verbal Conversion Table.. 153

References

Create <DD Object> (📖 TargetLink Data Dictionary Manager Reference)
Edit Matrix (📖 TargetLink Data Dictionary Manager Reference)
Save (📖 TargetLink Data Dictionary Manager Reference)

Exporting the A2L File

Where to go from here

Information in this section

How to Export A2L files via the TargetLink Data Dictionary Manager

Objective

You can use the TargetLink Data Dictionary Manager to export data from a DD project as an A2L file.

Preconditions

Before you can export an A2L file, the following preconditions must be fulfilled:

- You prepared your data for export and generated code. Refer to Specifying the Data to Export and Generating Code on page 120.
- You provided information on the ECU application for which the A2L file is to be exported. Refer to Specifying the Build Object on page 131.
- You specified an ASAP1b interface. Refer to Specifying the ASAP1b Interface on page 143.

Method

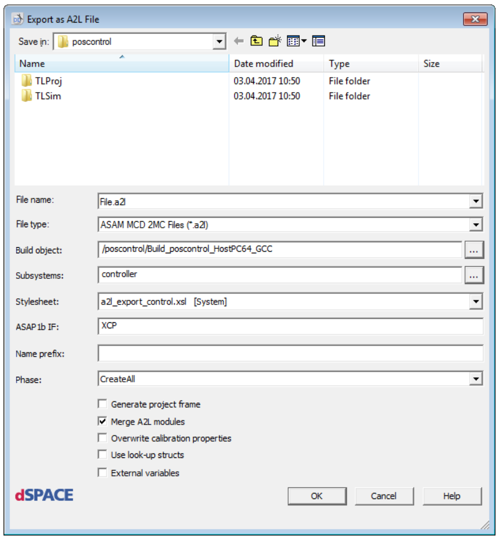
To export an A2L file via the Data Dictionary Manager

1 On the Data Dictionary Manager's File menu, click Export – as A2L File.

Note

A2L files can be exported only if DD0 is the active workspace.

The Data Dictionary Manager opens the **Export as A2L File** dialog.



Save in: poscontrol

Name	Date modified	Type	Size
TLProj	03.04.2017 10:50	File folder	
TLSim	03.04.2017 10:50	File folder	

File name:	File.a2l
File type:	ASAM MCD 2MC Files (*.a2l)
Build object:	/poscontrol/Build_poscontrol_HostPC64_GCC
Subsystems:	controller
Stylesheet:	a2l_export_control.xsl [System]
ASAP 1b IF:	XCP
Name prefix:	
Phase:	CreateAll

- ☐ Generate project frame
- ☑ Merge A2L modules
- ☐ Overwrite calibration properties
- ☐ Use look-up structs
- ☐ External variables

2 In the dialog, specify the A2L file to be exported.

3 Configure the export process according to your requirements.

4 Click **OK**.

Result

The selected data is exported as an A2L file.

Related topics

Basics

Preparing the A2L Export.. 119
Specifying the ASAP1b Interface... 143
Specifying the Build Object... 131

References

Export as A2L File (📖 TargetLink Data Dictionary Manager Reference)

How to Create a Build Object and Export Data as an A2L File via the MATLAB API

Export via DD MATLAB tool

TargetLink's MATLAB API provides the following function that you can use to create a DD **Build** object and export an A2L file via the API:

```
dsdd_export_a2l_file
```

Preconditions

Before you can export data as an A2L file:

- You have to generate code.
- You have to provide the following files:
  - Target Info file (`TargetInfo.xml`) that contains basic information on the target/compiler combination. Refer to Details on Providing a Target Info file on page 132.
  - Target config file (`TargetConfig.xml`) that contains information on basic data types. Refer to Details on Providing a Target Config File on page 133.
  - Optionally, a linker MAP file and assembler list files that contain ECU address information, if the A2L file is to contain address information. Refer to Details on Providing a Linker MAP File and Assembler List Files on page 134.
  - You must specify an ASAP1b interface for the data to be exported. Refer to Specifying the ASAP1b Interface on page 143.

Method

To create a Build object and export an A2L file via the MATLAB API

1 In the MATLAB Command Window, enter

```
bSuccess = dsdd_export_a2l_file(...
   <propertyname>, <propertyValue>, ...)
```

This function expects additional function parameters as propertyName/propertyValue pairs. For a list of valid properties see **`dsdd_export_a2l_file`**.

Result

The selected data is exported as an A2L file.

Related topics

Basics

Details on Providing a Linker MAP File and Assembler List Files	134
Details on Providing a Target Config File	133
Details on Providing a Target Info file	132
Preparing the A2L Export	119
Specifying the ASAP1b Interface	143

HowTos

How to Export Data as an A2L File via the API if a Build Object Exists	164

References

dsdd_export_a2l_file (📖 TargetLink API Reference)

How to Export Data as an A2L File via the API if a Build Object Exists

Generic export via the MATLAB API

TargetLink's MATLAB API provides the following function that you can use to export A2L files *if a DD* Build object *exists*:

```
[errorCode] = dsdd('Export','Format',A2L,…
<propertyName>,<propertyValue>,…);
```

Preconditions

- You generated code.
- You created a DD **Build** object, as described in Specifying the Build Object on page 131.

Method

To export data as an A2L file via the API if a Build object exists

1 In the MATLAB Command window, enter the following:

```
dsdd('Export',…
    'Format','A2l','Build', '<PathToBuildObject>', …
    <propertyName>, <propertyValue>,…)
```

This function expects additional function parameters as attributeName/attributeValue pairs. For valid attributes, refer to `dsdd('Export', 'Format','A2L', ...)`.

Result

The selected data is exported as an A2L file.

Related topics

Basics

Preparing the A2L Export.. 119

HowTos

How to Create a Build Object and Export Data as an A2L File via the MATLAB API.................. 162

Exporting A2L Files Containing Variables Defined in External Code

Introduction

You can instruct TargetLink to include the descriptions of variables that are defined in external code but not used in TargetLink's production code into its A2L files.

How to Export an A2L File Containing Variables Defined in External Code but Not Used in TargetLink's Production Code

Objective

To include descriptions of variables that fulfill the following conditions in A2L files generated by TargetLink:

- They are defined in external code.
- They are not used in TargetLink's production code.

This is done via DD-based code generation.

Preconditions

- The poscontrol demo model must be open.
- Production code has been generated.

Method

To export an A2L file containing variables defined in external code but not used in TargetLink's production code

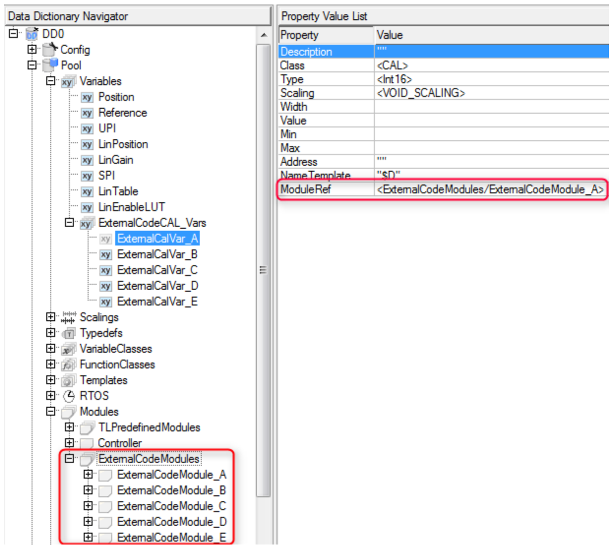
1 Create at least one DD Module object for the external code that contains the definitions of the variables that you want to include in your A2L file. Name them as required.

> **Note**
>
> When creating more than one module, group them in a DD ModuleGroup object. This makes it easier to reference them at a code generation unit (see step 6).
> Specifying different modules has no effect on the A2L file export.

2 Configure the modules by setting the following properties at their ModuleInfo child object:

Property	Value
NameTemplate	`$D`
CodeGenerationBasis	`DDBased`
Responsibility	`External`
ExcludeFromCodeGeneration	`off`

3 Reference the created modules at their DD **Variable** objects in the Pool area.



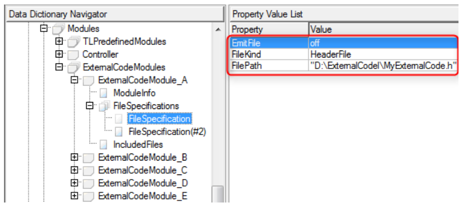
Data Dictionary Navigator

```
DD0
  Config
  Pool
    Variables
      Position
      Reference
      UPI
      LinPosition
      LinGain
      SPI
      LinTable
      LinEnableLUT
      ExternalCodeCAL_Vars
        ExternalCalVar_A
        ExternalCalVar_B
        ExternalCalVar_C
        ExternalCalVar_D
        ExternalCalVar_E
    Scalings
    Typedefs
    VariableClasses
    FunctionClasses
    Templates
    RTOS
    Modules
      TLPredefinedModules
      Controller
      ExternalCodeModules
        ExternalCodeModule_A
        ExternalCodeModule_B
        ExternalCodeModule_C
        ExternalCodeModule_D
        ExternalCodeModule_E
```

Property Value List

Property	Value
Description	""
Class	<CAL>
Type	<Int16>
Scaling	<VOID_SCALING>
Width	
Value	
Min	
Max	
Address	""
NameTemplate	"$D"
ModuleRef	<ExternalCodeModules/ExternalCodeModule_A>

4 At the modules, create DD **FileSpecification** objects for your header and source files. Set the **EmitFile** property to `off`.



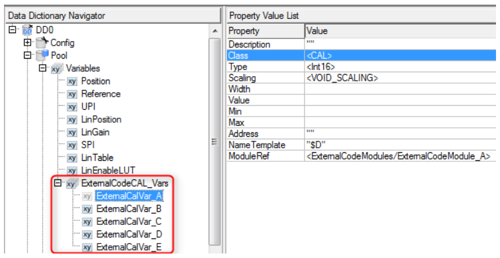
Data Dictionary Navigator

```
Modules
  TLPredefinedModules
  Controller
  ExternalCodeModules
    ExternalCodeModule_A
      ModuleInfo
      FileSpecifications
        FileSpecification
        FileSpecification(#2)
      IncludedFiles
    ExternalCodeModule_B
    ExternalCodeModule_C
    ExternalCodeModule_D
    ExternalCodeModule_E
```

Property Value List

Property	Value
EmitFile	off
FileKind	HeaderFile
FilePath	"D:\ExternalCodel\MyExternalCode.h"

5 Specify DD **Variable** objects describing your externally defined variables in the **Pool** area.



Data Dictionary Navigator

```
DD0
  Config
  Pool
    Variables
      Position
      Reference
      UPI
      LinPosition
      LinGain
      SPI
      LinTable
      LinEnableLUT
      ExternalCodeCAL_Vars
        ExternalCalVar_A
        ExternalCalVar_B
        ExternalCalVar_C
        ExternalCalVar_D
        ExternalCalVar_E
```

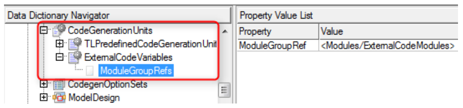
Property Value List

Property	Value
Description	""
Class	<CAL>
Type	<Int16>
Scaling	<VOID_SCALING>
Width	
Value	
Min	
Max	
Address	""
NameTemplate	"$D"
ModuleRef	<ExternalCodeModules/ExternalCodeModule_A>

6 Create a DD **CodeGenerationUnit** object (CGU) and name it as required.

At the CGU, create a DD **ModuleGroupRefs** child object.

Reference the module group you created in step 1 at the **ModuleGroupRefs** child object by specifying it in the **ModuleGroupRef** property.



Data Dictionary Navigator

```
CodeGenerationUnits
  TLPredefinedCodeGenerationUnit
  ExternalCodeVariables
    ModuleGroupRefs
CodegenOptionSets
ModelDesign
```

Property Value List

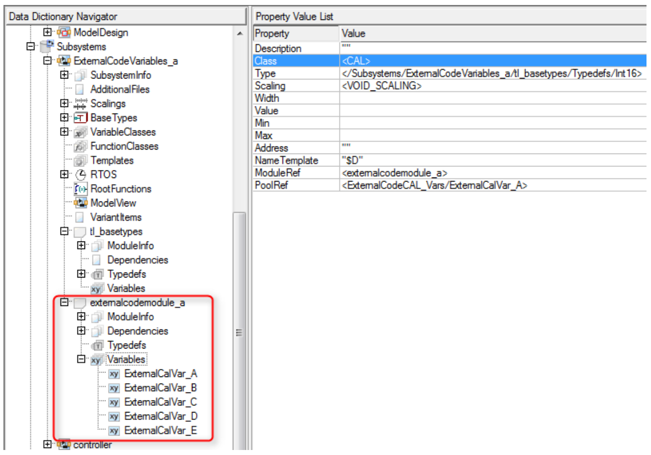
Property	Value
ModuleGroupRef	<Modules/ExternalCodeModules>

Note

If you did not group your modules, create DD ModuleRefs objects for each module instead and reference each module individually.

7 Start code generation from the CGU or the module.

A DD Subsystem object containing a DD Module object is created. No C modules are generated. You can locate the variables of your external code in the DD VariableGroup object belonging to the module:



Property Value List

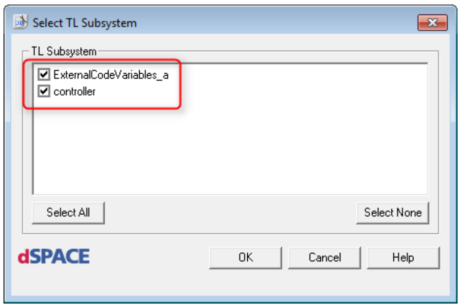
Property	Value
Description	""
Class	<CAL>
Type	</Subsystems/ExternalCodeVariables_a/tl_basetypes/Typedefs/Int16>
Scaling	<VOID_SCALING>
Width	
Value	
Min	
Max	
Address	""
NameTemplate	"$D"
ModuleRef	<externalcodemodule_a>
PoolRef	<ExternalCodeCAL_Vars/ExternalCalVar_A>

8 From the File menu, select Export - As A2L file ... to start the A2L export module.

9 In the Export as A2L File dialog, locate the Subsystems edit field and click Browse.

The Select TL Subsystem dialog opens.

10 Select the subsystems whose data you want to merge and click OK.



☑ ExternalCodeVariables_a
☑ controller

11 Specify all the settings in the Export as A2L File dialog as required and click OK. Refer to Preparing the A2L Export on page 119.

Result

You exported an A2L file that contains variables that were defined in external code. Your A2L file contains `CHARACTERISTICS` elements like the following:

```
/begin CHARACTERISTIC
    ExternalCalVar_A    /* Name */
    ""    /* LongIdentifier */
    VALUE    /* Type */
    0x0000    /* Address */
    SWORD_COL_DIRECT    /* Deposit */
    0    /* MaxDiff */
    VOID_SCALING    /* Conversion */
    -32768    /* LowerLimit */
    32767    /* UpperLimit */
    BYTE_ORDER MSB_LAST
/end CHARACTERISTIC
```

Related topics

Basics

Basics on Generating Code from the Data Dictionary (📖 TargetLink Customization and Optimization Guide)
Basics on Modules and Module Ownership (📖 TargetLink Customization and Optimization Guide)
Introducing A2L Files.. 112
Preparing the A2L Export.. 119

HowTos

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

Customizing the A2L File Export

Where to go from here

Information in this section

Basics on Customizing the A2L Export

A2L file export in 2 phases

The A2L export module performs A2L file export in two phases.

Phase 1: ConvertToA2L

This phase consists of two steps:

Creating calibration properties The A2L export module adds calibration properties required by calibration tools such as ControlDesk to the DD Variable objects representing calibratable or displayable variables.

The following table lists the DD properties and their corresponding A2L attributes:

Variable Type	A2L Element	DD Property
CAL	`CHARACTERISTIC`	ByteOrder MaxDiff
DISP	`MEASUREMENT`	ByteOrder Accuracy Resolution

Note

By default the export module overwrites the values of the above properties with values derived from existing data.

You can specify calibration tool-specific properties (see the table above) manually before you start code generation and the export process and specify not to overwrite calibration tool-specific properties you set manually. This is possible for DD Variable objects in the Pool or Subsystems subtrees.

Properties other than the properties listed in the table above are not overwritten by the export module. If you specify these properties manually, your specification will be included in the generated A2L file. Of course, this is only possible for properties that can be mapped to A2L attributes.

Transforming data and storage in the reserved workspace DD3 The data to be exported from the Subsystem area is transformed to intermediate DD Generic objects, representing future A2L file elements. These intermediate objects are stored in DD3 within the DD A2L object tree, respecting the A2L file hierarchy. The mandatory parameters of A2L file elements are stored as DD properties.

Phase 2: WriteA2Lfile

In this phase, the data that was stored in the reserved workspace DD3 during phase 1 is exported to the final A2L file.

Writing the final A2L file The intermediate DD A2L object in DD3 is first transformed into an intermediate XML file by the TargetLink Data Dictionary's XML export module.

Then this XML file is transformed into the final A2L file whose contents and layout are specified by an XSL style sheet. You can also create your own style sheet, which allows you to manipulate the contents and layout of the generated A2L file.

Specifying the style sheet

TargetLink provides predefined style sheets (XSL files) it uses to transform the intermediary XML file into the final A2L file. You can copy these XSL files into a directory of your choice and adapt them to your needs. Refer to Defining Search Paths for Customization Files, Demo Models,TSM Extension Packages, and Instruction Set Simulators (📖 TargetLink Customization and Optimization Guide).

Example For structure components, TargetLink can generate two different formats for the value of the `SYMBOL_LINK` keyword:

- `<RootStructName>` **`<Offset>`** - For addresses obtained from a MAP file
- `<RootStructName>.`**`<ComponentName>`** **`0`** - For addresses obtained from an ELF file, where the address is known, the offset accordingly is `0`

You can control the format to use via a XSL stylesheet.

Copy all the A2L stylesheets and adapt the `a2l_export_control.xsl` as described in the file's code comments:

For Addresses from MAP Files	For Addresses from ELF Files
Set the takeRootStructForSymbolLinkEntry XSL variable to `"yes"`	Set the takeRootStructForSymbolLinkEntry XSL variable to `"no"`

Specifying the phase run

You can specify which export phase to run as follows:

- Via the Phase list on the Export as A2L File dialog
- Via the Phase property of the `dsdd_export_a2l_file` or `dsdd('Export', 'Format','A2L', ...)` API function

API example

```
% Generate A2L file intermediate hierarchy in DD3
dsdd_export_a2l_file(..., 'Phase','ConvertToA2L');
% Modify the DD3 content
…
% Write the A2L file intermediate hierarchy in DD3 to A2L file
dsdd_export_a2l_file(...,'Phase','WriteA2LFile');
```

Representation of data in the various phases

The representation of data depends on the phase of the A2L file export process. Refer to Details on the Representation of Data During A2L File Export on page 171.

Related topics

Basics

References

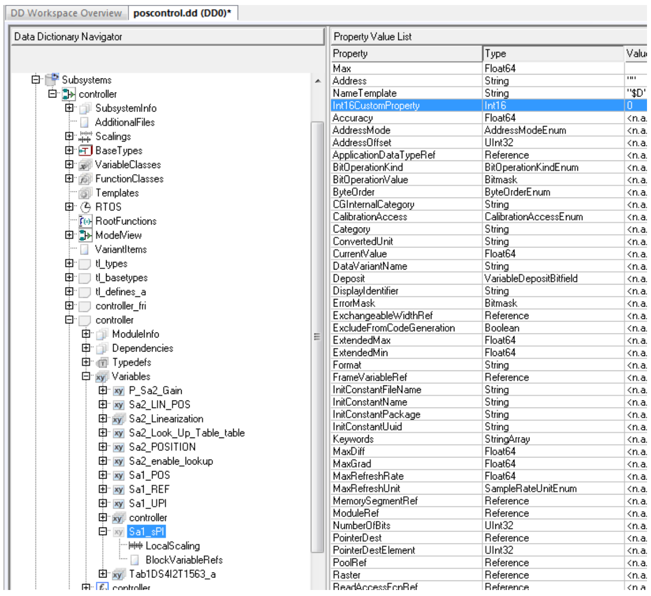
Details on the Representation of Data During A2L File Export

Introduction

The representation of data depends on the phase of the A2L file export process. You need to know the different representations to manipulate the A2L file export process. For example, if you want to create your own XSL style sheet for transforming the intermediate XML file into the final A2L file, you need to know how variables are represented in the intermediate XML file.

Representation in the DD object tree

Before you start A2L file export, the data to be exported is in the DD Subsystems object tree. The following screenshot shows an example representation of the measurable variable Sa1_sPI in the DD object tree. The variable belongs to the Subsystem object called `pidcontroller`.



DD Workspace Overview | poscontrol.dd (DD0)*

Data Dictionary Navigator

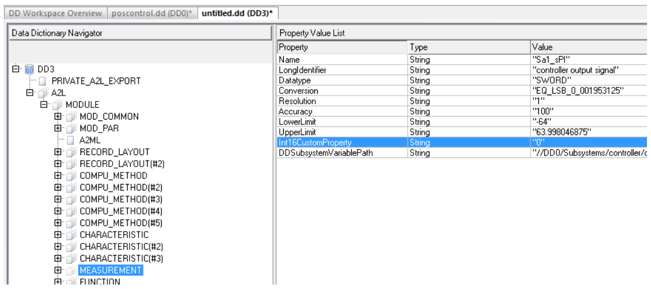
```
Subsystems
  controller
    SubsystemInfo
    AdditionalFiles
    Scalings
    BaseTypes
    VariableClasses
    FunctionClasses
    Templates
    RTOS
    RootFunctions
    ModelView
    VariantItems
    tl_types
    tl_basetypes
    tl_defines_a
    controller_fri
    controller
      ModuleInfo
      Dependencies
      Typedefs
      Variables
        P_Sa2_Gain
        Sa2_LIN_POS
        Sa2_Linearization
        Sa2_Look_Up_Table_table
        Sa2_POSITION
        Sa2_enable_lookup
        Sa1_POS
        Sa1_REF
        Sa1_UPI
        controller
        Sa1_sPI
          LocalScaling
          BlockVariableRefs
        Tab1DS4I2T1563_a
      controller
```

Property Value List

Property	Type	Valu
Max	Float64	
Address	String	""
NameTemplate	String	"$D'
Int16CustomProperty	Int16	0
Accuracy	Float64	<n.a.
AddressMode	AddressModeEnum	<n.a.
AddressOffset	UInt32	<n.a.
ApplicationDataTypeRef	Reference	<n.a.
BitOperationKind	BitOperationKindEnum	<n.a.
BitOperationValue	Bitmask	<n.a.
ByteOrder	ByteOrderEnum	<n.a.
CGInternalCategory	String	<n.a.
CalibrationAccess	CalibrationAccessEnum	<n.a.
Category	String	<n.a.
ConvertedUnit	String	<n.a.
CurrentValue	Float64	<n.a.
DataVariantName	String	<n.a.
Deposit	VariableDepositBitfield	<n.a.
DisplayIdentifier	String	<n.a.
ErrorMask	Bitmask	<n.a.
ExchangeableWidthRef	Reference	<n.a.
ExcludeFromCodeGeneration	Boolean	<n.a.
ExtendedMax	Float64	<n.a.
ExtendedMin	Float64	<n.a.
Format	String	<n.a.
FrameVariableRef	Reference	<n.a.
InitConstantFileName	String	<n.a.
InitConstantName	String	<n.a.
InitConstantPackage	String	<n.a.
InitConstantUuid	String	<n.a.
Keywords	StringArray	<n.a.
MaxDiff	Float64	<n.a.
MaxGrad	Float64	<n.a.
MaxRefreshRate	Float64	<n.a.
MaxRefreshUnit	SampleRateUnitEnum	<n.a.
MemorySegmentRef	Reference	<n.a.
ModuleRef	Reference	<n.a.
NumberOfBits	UInt32	<n.a.
PointerDest	Reference	<n.a.
PointerDestElement	UInt32	<n.a.
PoolRef	Reference	<n.a.
Raster	Reference	<n.a.
ReadAccessFcnRef	Reference	<n.a

Representation in the reserved workspace DD3

After phase 1 of the A2L file export process, the data to be exported is stored in the reserved workspace DD3, according to the A2L file hierarchy. As an example, the illustration below shows the representation of the measurable variable `Sa1_sPI` in the reserved workspace DD3. Note that all the properties are now of `String` type, regardless of the type they had originally.



DD Workspace Overview | poscontrol.dd (DD0)* | untitled.dd (DD3)*

Data Dictionary Navigator

```
DD3
  PRIVATE_A2L_EXPORT
  A2L
    MODULE
      MOD_COMMON
      MOD_PAR
      A2ML
      RECORD_LAYOUT
      RECORD_LAYOUT(#2)
      COMPU_METHOD
      COMPU_METHOD(#2)
      COMPU_METHOD(#3)
      COMPU_METHOD(#4)
      COMPU_METHOD(#5)
      CHARACTERISTIC
      CHARACTERISTIC(#2)
      CHARACTERISTIC(#3)
      MEASUREMENT
      FUNCTION
```

Property Value List

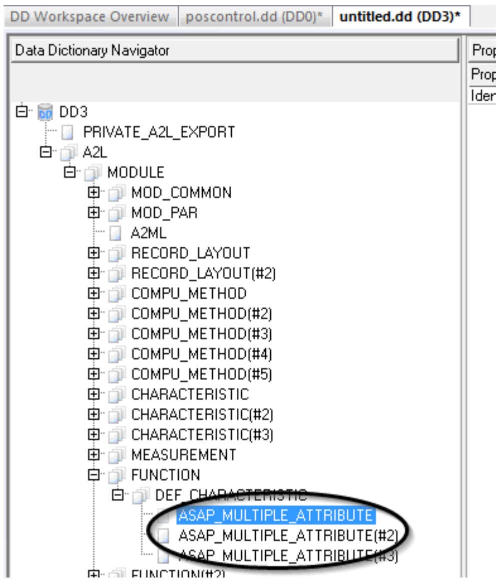
Property	Type	Value
Name	String	"Sa1_sPI"
LongIdentifier	String	"controller output signal"
Datatype	String	"SWORD"
Conversion	String	"EQ_LSB_0_001953125"
Resolution	String	"1"
Accuracy	String	"100"
LowerLimit	String	"-64"
UpperLimit	String	"63.998046875"
Int16CustomProperty	String	"0"
DDSubsystemVariablePath	String	"//DD0/Subsystems/controller/c

In the reserved workspace DD3, future A2L file elements such as `COMPU_METHOD`, `CHARACTERISTIC`, `MEASUREMENT`, etc. are represented as DD objects. They are written in upper case. For instructions to open workspaces, refer to Managing DD workspaces (📖 TargetLink Data Dictionary Basic Concepts Guide).

Mandatory parameters Mandatory parameters of A2L file elements are represented as DD properties. The DD property names and the parameter names specified in the A2L standard are identical.

Optional parameters Optional parameters of A2L file elements are represented as DD objects. They are written in upper case.

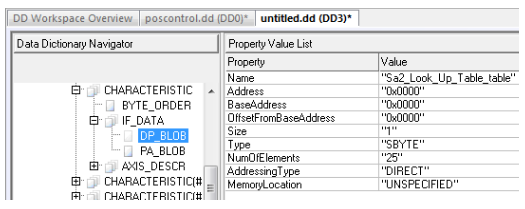
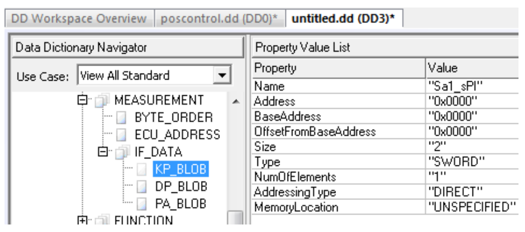
Mandatory parameters occurring several times If a mandatory parameter occurs several times ('multiple parameter'), the `ASAP_MULTIPLE_ATTRIBUTE` DD object is created for each occurrence of the parameter. It contains only one property, which indicates where the parameter occurs. In the reserved workspace DD3, a mandatory parameter occurring several times is represented like this:



```
DD Workspace Overview | poscontrol.dd (DD0)* | untitled.dd (DD3)*

Data Dictionary Navigator                                  Prop
                                                           Prop
                                                           Ider
DD3
  PRIVATE_A2L_EXPORT
  A2L
    MODULE
      MOD_COMMON
      MOD_PAR
      A2ML
      RECORD_LAYOUT
      RECORD_LAYOUT(#2)
      COMPU_METHOD
      COMPU_METHOD(#2)
      COMPU_METHOD(#3)
      COMPU_METHOD(#4)
      COMPU_METHOD(#5)
      CHARACTERISTIC
      CHARACTERISTIC(#2)
      CHARACTERISTIC(#3)
      MEASUREMENT
      FUNCTION
        DEF_CHARACTERISTIC
          ASAP_MULTIPLE_ATTRIBUTE
          ASAP_MULTIPLE_ATTRIBUTE(#2)
          ASAP_MULTIPLE_ATTRIBUTE(#3)
      FUNCTION(#2)
```

IF_DATA elements of CHARACTERISTIC, MEASUREMENT and AXIS_PTS elements in the reserved workspace DD3 An IF_DATA object is created in the reserved workspace DD3 for each CHARACTERISTIC, MEASUREMENT and AXIS_PTS element:

CHARACTERISTIC/AXIS_PTS Element in Reserved Workspace DD3	**MEASUREMENT Element in Reserved Workspace DD3**
	

```
DD Workspace Overview | poscontrol.dd (DD0)* | untitled.dd (DD3)*

Data Dictionary Navigator          Property Value List
                                   Property                 Value
CHARACTERISTIC                     Name                     "Sa2_Look_Up_Table_table"
  BYTE_ORDER                       Address                  "0x0000"
  IF_DATA                          BaseAddress              "0x0000"
    DP_BLOB                        OffsetFromBaseAddress    "0x0000"
    PA_BLOB                        Size                     "1"
  AXIS_DESCR                       Type                     "SBYTE"
CHARACTERISTIC(#                   NumOfElements            "25"
CHARACTERISTIC(#                   AddressingType           "DIRECT"
                                   MemoryLocation           "UNSPECIFIED"
```

```
DD Workspace Overview | poscontrol.dd (DD0)* | untitled.dd (DD3)*

Data Dictionary Navigator          Property Value List
Use Case: View All Standard        Property                 Value
MEASUREMENT                        Name                     "Sa1_sPI"
  BYTE_ORDER                       Address                  "0x0000"
  ECU_ADDRESS                      BaseAddress              "0x0000"
  IF_DATA                          OffsetFromBaseAddress    "0x0000"
    KP_BLOB                        Size                     "2"
    DP_BLOB                        Type                     "SWORD"
    PA_BLOB                        NumOfElements            "1"
FUNCTION                           AddressingType           "DIRECT"
                                   MemoryLocation           "UNSPECIFIED"
```

Each of the BLOB objects have the following properties:

Property	**Description**
Name	Name of the variable
Address	Address of the variable
Size	Size of a single element in bytes
Type	Data type of the variable in the A2L file (SBYTE, UBYTE, SWORD, UWORD, SLONG, ULONG,FLOAT32_IEEE, FLOAT64_IEEE)
NumOfElements	Number of elements of the variable
AddressingType	Addressing type of the variable (DIRECT, PBYTE,PWORD, PLONG)
MemoryLocation	Location of the memory (EXTERN or INTERN)

Custom properties Custom properties set at Variable or Scaling objects are represented as String properties of the CHARACTERISTIC, MEASUREMENT, AXIS_PTS, or COMPU_METHOD objects (see Int16CustomProperty in the figures above).

Note

You can set your own properties at Variable or Scaling objects created in the Pool area.
These properties are then propagated to variables in the Subsystems area and DD objects representing A2L entries in the reserved workspace DD3.
Custom properties are useful if you need to add additional information to the A2L file (e.g., a CAL property that does not have a suitable standard DD property). You can also use them in functions that you call after phase 1 of the A2L file export.
By default, user properties are ignored during style sheet transformation. You have to modify the existing style sheets or write new ones if you need to evaluate them.

Reference to DD Variable object The reference to the DD Variable object that the CHARACTERISTIC, MEASUREMENT, AXIS_PTS or AXIS_DESCR element has been generated for is saved as a DD path under the DDSubsystemVariablePath property of the intermediate DD object in the reserved workspace DD3 (see figures above).

Representation in the intermediate XML file

During phase 2, the TargetLink Data Dictionary's XML export module transforms the data in the reserved workspace DD3 into an intermediate XML file. Each DD object becomes an XML element, and each DD property becomes an XML attribute. The following example shows the representation of a measurable variable in the intermediate XML file.

```xml
<MEASUREMENT
   Name="Sa1_sPI" LongIdentifier="" Datatype="SWORD" Conversion="EQ_LSB_7_62939453125em06_OFF_0" Resolution="1"
   Accuracy="100"    LowerLimit="-0.25" UpperLimit="0.249992" Address="0x2004" Int16CustomProperty="0"
   DDSubsystemVariablePath="//DD0/Subsystems/picontroller/picontroller/Variables/P_Sa1_Ki">
   <BYTE_ORDER ByteOrder="MSB_LAST"/>
   <ECU_ADDRESS Address="0x2004"/>
   <IF_DATA Name="CCP">
      <KP_BLOB
         Name="Sa1_sPI" Address="0x2004" Size="2" Type="SWORD" NumOfElements="1" AddressingType="DIRECT"
         MemoryLocation="EXTERN"/>
      <DP_BLOB
         Name="Sa1_sPI" Address="0x2004" Size="2" Type="SWORD" NumOfElements="1" AddressingType="DIRECT"
         MemoryLocation="EXTERN"/>
      <PA_BLOB
         Name="Sa1_sPI" Address="0x2004" Size="2" Type="SWORD" NumOfElements="1" AddressingType="DIRECT"
         MemoryLocation="EXTERN"/>
   </IF_DATA>
</MEASUREMENT>
```

Mandatory parameters occurring several times In the intermediate XML file, mandatory parameters occurring several times are represented like this:

```
<FUNCTION LongIdentifier="" Name="Sa2_Linearization">
   <DEF_CHARACTERISTIC>
      <ASAP_MULTIPLE_ATTRIBUTE Identifier="P_Sa2_Gain"/>
      <ASAP_MULTIPLE_ATTRIBUTE Identifier="Sa2_Look_Up_Table_z_table"/>
      <ASAP_MULTIPLE_ATTRIBUTE Identifier="Sa2_enable_lookup"/>
   </DEF_CHARACTERISTIC>
</FUNCTION>
```

Representation in the final A2L file

Then the intermediate XML file is transformed into the final A2L file whose contents and layout are specified by an XSL style sheet. The following excerpt shows the representation of a measurable variable in the final A2L file.

```
/begin MEASUREMENT
   Sa1_sPI                                  /* Name */
   ""                                       /* LongIdentifier */
   SWORD                                    /* Datatype */
   EQ_LSB_7_62939453125em06_OFF_0           /* Conversion */
   1                                        /* Resolution */
   100                                      /* Accuracy */
   -0.25                                    /* LowerLimit */
   0.249992                                 /* UpperLimit */
   ECU_ADDRESS 0x0000                       /* address Sa1_sPI/*
   BYTE_ORDER MSB_LAST
   /begin IF_DATA ASAP1B_CCP
      KP_BLOB 0x0 0x0000 2
   /end IF_DATA
/end MEASUREMENT
```

Mandatory parameters occurring several times In the final A2L file, mandatory parameters occurring several times are represented like this:

```
/begin FUNCTION
   Sa2_Linearization                       /* Name */
   ""                                      /* LongIdentifier */
   /begin DEF_CHARACTERISTIC
      P_Sa2_Gain                           /* Identifier */
      Sa2_Look_Up_Table_z_table            /* Identifier */
      Sa2_enable_lookup                    /* Identifier */
   /end DEF_CHARACTERISTIC
/end FUNCTION
```

Related topics

Basics

Customizing the A2L File Export.. 168

References

Export Hook Scripts (📖 TargetLink File Reference)

How to Support Another ASAP1b Interface

Objective

To use an ASAP1b interface not listed in Basics on ASAP1b Interfaces on page 143.

AML and XSL file

Adding support of another ASAP1b interface involves an AML file and and XSL A2L style sheet file.

AML file An ASAM MCD-2MC metalanguage (AML) file specifies the format of the ASAP1b interface-specific parameters.

Style sheet An `IF_DATA` block is created for each `CHARACTERISTIC`, `MEASUREMENT` and `AXIS_PTS` element during A2L file generation. An ASAP1b interface-specific XSL style sheet controls how `IF_DATA` blocks for `CHARACTERISTIC`, `MEASUREMENT` and `AXIS_PTS` elements will appear in the generated A2L file.

Refer to Details on the Representation of Data During A2L File Export on page 171.

Preconditions

The following preconditions must be fulfilled:

- You created a folder for the A2L style sheets and the AML files and specified its location in the Data Dictionary - A2L Export topic of the Preferences Editor.

  Refer to Defining search paths (🕮 TargetLink Customization and Optimization Guide).
- You derived the A2L stylesheets from their templates.

  Refer to Creating customization files (🕮 TargetLink Customization and Optimization Guide).

Method

To support another ASAP1b interface

1 Provide an AML file `<ASAM_MCD_1_InterfaceName>.aml` that describes your ASAP1b interface and place it in the folder that you specified in A2L style sheets and AML files in the Data Dictionary - A2L Export topic of the Preferences Editor.

2 Create a new style sheet called `a2l_if_data_<ASAM_MCD_1_InterfaceName>.xsl`. Use an existing XSL file from the A2L style sheet and AML file folder as a template.

3 Specify the following templates within the new style sheet:
   - `IF_DATA_CHARACTERISTIC_<ASAM_MCD_1_InterfaceName>`
   - `IF_DATA_MEASUREMENT_<ASAM_MCD_1_InterfaceName>`
   - `IF_DATA_AXIS_PTS_<ASAM_MCD_1_InterfaceName>`

   To specify these templates, you need to know how `IF_DATA` blocks for `CHARACTERISTIC` and `MEASUREMENT` elements appear in the intermediate XML file that is created during A2L file export.

4 Copy the new style sheet file to the folder you specified in A2L style sheets and AML files in the Data Dictionary - A2L Export topic of the Preferences Editor.

5 In the A2L style sheet and AML file folder edit the `a2l_export_blobs.xsl` file and add the following lines:

Add ...	To ...
`<xsl:import href="a2l_if_data_<ASAM_MCD_1_InterfaceName>.xsl"/>`	`IF_DATA template Imports` XSL section

`<xsl:if test = "contains(@Name, '<ASAM_MCD_1_InterfaceName>')">`
`<xsl:call-template`
`name="IF_DATA_CHARACTERISTIC_<ASAM_MCD_1_InterfaceName>"></xsl:call-template>`
`</xsl:if>`
	`IF_DATA_CHARACTERISTIC` XSL template

`<xsl:if test = "contains(@Name, '<ASAM_MCD_1_InterfaceName>')">`
`<xsl:call-template name="IF_DATA_AXIS_PTS_<ASAM_MCD_1_InterfaceName>"></xsl:call-template>`
`</xsl:if>`
	`IF_DATA_AXIS_PTS` XSL template

`<xsl:if test = "contains(@Name, '<ASAM_MCD_1_InterfaceName>')">`
`<xsl:call-template`
`name="IF_DATA_MEASUREMENT_<ASAM_MCD_1_InterfaceName>"></xsl:call-template>`
`</xsl:if>`
	`IF_DATA_MEASUREMENT` XSL template

6 Save the `a2l_export_blobs.xsl` file.

Result

The `<ASAM_MCD_1_InterfaceName>` is now supported as an ASAP1b interface.

Related topics

Basics

Basics on ASAP1b Interfaces... 143
Defining Search Paths for Customization Files, Demo Models,TSM Extension Packages, and Instruction Set Simulators (TargetLink Customization and Optimization Guide)
Details on the Representation of Data During A2L File Export.................... 171
Specifying the ASAP1b Interface.. 143

References

Preferences Editor (TargetLink Tool and Utility Reference)

Exchanging AUTOSAR Files

Where to go from here

Information in this section

Introduction to Importing and Exporting AUTOSAR Files

Where to go from here

Information in this section

Basics on AUTOSAR Files

Content

AUTOSAR files contain a formal description of AUTOSAR elements of a software component in the XML format. For details, refer to the *Software Component Template* document at http://www.autosar.org.

Supported AUTOSAR Releases

For import and export, the Data Dictionary uses the AUTOSAR XML Schema, which is an XML language definition for exchanging AUTOSAR models and descriptions. The Data Dictionary supports the following AUTOSAR Releases:

Classic AUTOSAR Release	Revision
R19-11	19-11
4.4	4.4.0

Classic AUTOSAR Release	Revision
4.3	4.3.1 4.3.0
4.2	4.2.2 4.2.1
4.1	4.1.3 4.1.2 4.1.1
4.0	4.0.3 4.0.2

Basics on Packages

Packages

According to AUTOSAR, each AUTOSAR element such as a software component must be contained in a package. You can use these packages to structure elements contained in AUTOSAR files.

Package information in the Data Dictionary

It is best to provide package information as follows:

- Group DD objects that result in AUTOSAR elements in DD <Group> objects whenever possible.
- Add a GroupInfo object to each <Group> object to specify package information.

The following table shows where in the Data Dictionary you can specify the package information for a given AUTOSAR element kind:

Package Information For	Location in DD
Application data types	DD GroupInfo object of ApplicationDataTypeGroup objects
Artifacts belonging to autogenerated PIMs	`/Pool/Autosar/Config/ExportRules/<ExportRule>`
Atomic software component	DD GroupInfo object of SoftwareComponentGroup objects
ConstantSpecificationMappingSets for which you cannot specify package information in the Data Dictionary, because they are implicitly created during export	`/Pool/Autosar/Config/ExportRules/<ExportRule>`
Data constraints	`/Pool/Autosar/Config/ExportRules/<ExportRule>`
Data type mapping sets	DD GroupInfo object of DataTypeMappingSetGroup objects
Initialization constants for which you cannot specify package information in the Data Dictionary, because they are implicitly created during export	`/Pool/Autosar/Config/ExportRules/<ExportRule>`

Package Information For	Location in DD
Initial values of the following elements: ▪ PerInstanceCalPrm ▪ PerInstanceMemories[1)] ▪ SharedCalPrm	DD InitConstantPackage property of DD Variable objects
Interfaces	DD GroupInfo object of InterfaceGroup objects
Mode declarations	DD GroupInfo object of ModeDeclarationGroup objects
Scalings	DD GroupInfo object of ScalingGroup objects
Type definitions[2)]	DD GroupInfo object of TypedefGroup objects
Units	DD Package property of DD Unit objects.
Variables	DD GroupInfo object of VariableGroup objects

1) Applies only for per instance memories that are not auto-generated.
2) In TargetLink, implementation data types as defined by AUTOSAR are treated as type definitions.

Related topics

Basics

Basics on Exporting AUTOSAR Files.. 182
Basics on Exporting Splittable Elements.. 180
Basics on Importing AUTOSAR Files.. 181
Basics on Preparing SWCs for Multiple Instantiation (📖 TargetLink Classic AUTOSAR Modeling Guide)

Basics on Exporting Splittable Elements

Splittable elements according to AUTOSAR

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

Splittable elements in TargetLink

TargetLink lets you define a file name for splittable elements. During export, the splittable elements are then exported into these files:

Splittable Element	File Name Specification[1)]
`<AR-TYPED-PER-INSTANCE-MEMORYS>`	DD ArTypedPerInstanceMemoriesFileName property
`<CONSTANT-MEMORYS>`	DD ConstantMemoriesFileName property
`<EXPLICIT-INTER-RUNNABLE-VARIABLES>`	DD ExplicitInterRunnableVariablesFileName property
`<SWC-INTERNAL-BEHAVIOR>`	DD InternalBehaviorFileName property
`<IMPLICIT-INTER-RUNNABLE-VARIABLES>`	DD ImplicitInterRunnableVariablesFileName property
`<PER-INSTANCE-PARAMETERS>`	DD PerInstanceParametersFileName property

Splittable Element	File Name Specification[1]
`<SHARED-PARAMETERS>`	DD SharedParametersFileName property
`<STATIC-MEMORYS>`	DD StaticMemoriesFileName property

[1] Via properties of DD SoftwareComponent objects

Limitation

Note

Restricted to AUTOSAR 4.x export.
TargetLink does not import data prototypes if they are defined in a different AUTOSAR file than the internal behavior (splittable elements).

Related topics

Basics

AR_POSCONTROL (TargetLink Demo Models)

Importing AUTOSAR Files

Basics on Importing AUTOSAR Files

Importing AUTOSAR files

During file import, the TargetLink Data Dictionary maps the elements of an AUTOSAR file to corresponding DD objects, and stores them in the DD object tree.

Note

If a software component description is split into several AUTOSAR files and/or packages, the file/package that contains the ATOMIC-SOFTWARE-COMPONENT-TYPE element must be imported before the file/package with the related INTERNAL-BEHAVIOR element. Otherwise certain elements cannot be imported.

Package information

Package information contained in AUTOSAR files is preserved during import to the Data Dictionary.

Configuring import

You can configure the import at the DD `/Pool/Autosar/Config/ImportExport` object.

Import methods

You can use the following methods for importing AUTOSAR files:

Method	Description
Import from Container `dsdd('Import', 'Format','Container', …)`	Lets you import software component containers.
Import from AUTOSAR File `dsdd('Import', 'Format','AUTOSAR', …)`	Lets you import AUTOSAR files to the Data Dictionary.
`tl_generate_swc_model`	Lets you generate a frame model from AUTOSAR data in DD project files or AUTOSAR files. The command automatically performs an import for AUTOSAR files.

Related topics

Basics

Basics on Packages 179

References

Import from AUTOSAR File (📖 TargetLink Data Dictionary Manager Reference)
Import from Container (📖 TargetLink Data Dictionary Manager Reference)
tl_generate_swc_model (📖 TargetLink API Reference)

Exporting AUTOSAR Files

Basics on Exporting AUTOSAR Files

Exporting AUTOSAR files

TargetLink maps DD objects and object properties to AUTOSAR elements during export to AUTOSAR files. The TargetLink version that is used to generate the AUTOSAR file is written to this file as a comment.

Exporting from the Data Dictionary

You can export AUTOSAR data from the Data Dictionary as shown in the following table:

Pool Area	Subsystems Area
You can export interface descriptions, type definitions and scalings from the `DataDictionary/Pool` area. You can export objects below the following subtrees: ▪ Interfaces ▪ Scalings ▪ Typedefs	You can export a complete description of an SWC's internal behavior and implementation from the `DataDictionary/Subsystems` area. You can select from the subsystems of the current DD project file.

Note

When you export from the `/Subsystems` area of the Data Dictionary, the data from the latest code generation is exported. Changes made in the `/Pool` area after code generation are ignored.

Configuring export

You can configure the export at the DD `/Pool/Autosar/Config/ImportExport` object.

Package information You can specify how AUTOSAR packages are distributed to files during export by setting the ExportStrategy export option, which is located at `/Pool/Autosar/Config/ImportExport`.

For details, refer to Basics on Packages on page 179.

Data types

TargetLink's Data Dictionary provides a predefined system template for AUTOSAR release 4.x. This template contains different DD TypedefGroup objects. If you use DD Typedef objects belonging to these groups, TargetLink exports additional ARXML files. This is shown in the following table:

Typedef Group	Package	AutosarFileName
PlatformTypes	AUTOSAR_Platform/ImplementationDataTypes	AUTOSAR_Platform.arxml
TLDataTypes	TargetLink/TLDataTypes	TargetLink.TLDataTypes.arxml
DataTypes	-	-

UUIDs

TargetLink generates UUIDs for AUTOSAR elements during code generation in the DD Subsystems area if no UUID is specified for the corresponding DD objects in the Pool area.

Export methods

You can use the following methods for exporting AUTOSAR files:

Method	Description
Export as AUTOSAR File (📖 TargetLink Data Dictionary Manager Reference) `dsdd('Export', 'Format', 'AUTOSAR', …)`	Lets you export AUTOSAR files from the Data Dictionary.
Export as Container (📖 TargetLink Data Dictionary Manager Reference) `tl_export_container`	Lets you export software component containers.

Related topics

References

Import/Export Hook Scripts (AUTOSAR) (📖 TargetLink File Reference)
tl_export_container (📖 TargetLink API Reference)

Customizing AUTOSAR File Import and Export

Basics on Using Hook Scripts for Customization (AUTOSAR)

Introduction

To customize AUTOSAR file import and export via hook scripts.

Basics on hook scripts

TargetLink provides several hook scripts that you can derive from their templates. These hook scripts allow you to add your own commands at specific points of TargetLink actions such as AUTOSAR file import and export. This lets you adapt TargetLink actions to your company's development tool chain. The hook scripts contain predefined variables for you to work with.

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 (📖 TargetLink Customization and Optimization Guide).

Import

To customize the AUTOSAR file import, you can use the following hook scripts:

- **`tl_pre_arimport_hook`** (📖 TargetLink File Reference)
- **`tl_post_arimport_hook`** (📖 TargetLink File Reference)

These hook scripts are called immediately before and after the AUTOSAR file import via the Import dialog or API function.

Export

To customize the AUTOSAR file export, you can use the following hook scripts:

- `tl_pre_arexport_hook` (📖 TargetLink File Reference)
- `tl_post_arexport_hook` (📖 TargetLink File Reference)

These hook scripts are called immediately before and after the AUTOSAR file export via Export dialog or API function.

Exporting AUTOSAR files after code generation

TargetLink lets you export AUTOSAR files after code generation. You have to add an appropriate DD API function in the `tl_post_codegen_hook` (📖 TargetLink File Reference) hook script.

Related topics

Basics

Basics on Using Hook Scripts (📖 TargetLink Customization and Optimization Guide)
Deriving Customization Files From Their Templates (📖 TargetLink Customization and Optimization Guide)

HowTos

How to Create Customization Files via the Create Customization Files Dialog (📖 TargetLink Customization and Optimization Guide)
How to Define TargetLink's Search Path for DD-Related Customization Files (📖 TargetLink Customization and Optimization Guide)

References

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

Exchanging OIL Files

Where to go from here

Information in this section

Introduction to OIL Files

OIL File Format

OSEK Implementation Language (OIL)	OIL is a configuration language for describing complete OSEK systems (Open Systems and the Corresponding Interfaces for Automotive Electronics) for system configuration and generation. The OIL file format was defined by the OSEK/VDX committee, a joint project of the automotive industry that aims at defining an industry-standard for open-ended architecture for distributed control units in vehicles.
Purpose and content of OIL files	OIL files give information on the configuration and generation of OSEK applications. They thus contain all the parameters necessary for configuring and generating the operating-system-specific OSEK objects for the OSEK application.
Supported versions	The TargetLink Data Dictionary allows you to import and export OIL files. It supports file versions 2.2 and 2.3 for file import and for file export.
File format specification	The OSEK/VDX committee specified the OIL file format.

Importing OIL Files

Introduction

The TargetLink Data Dictionary serves as a common platform for data exchange, allowing you to import OIL files.

Where to go from here

Information in this section

Basics of Importing OIL Files

Various kinds of OIL files

The TargetLink Data Dictionary can handle the following kinds of OIL files:

- OIL files that contain the complete OIL description, including the implementation part
- OIL file fragments that contain the `CPU` container

  The file has to be syntactically correct and the `CPU` container has to be complete, starting with the `CPU` keyword and ending with its matching braces.
- OIL file fragments that contain one or several OIL elements, e.g., `TASK`s

  The file has to be syntactically correct and the OIL elements have to be complete, starting with the respective keyword and ending with its matching braces.

Imported OIL files are always stored in the Pool area of the DD object tree.

How to Import OIL Files via the Data Dictionary Manager

Objective

The TargetLink Data Dictionary allows you to import OIL files via the Data Dictionary Manager.

Note

OIL files can include further OIL files. To find an included OIL file which does not reside in the same directory as the main one, the import module needs to know the folder that the included OIL file resides in. At the `Pool/RTOS/RTOSInfo` object, enter the path of the included OIL file as the value of its OILIncludeFileSearchPath property. If several paths are specified, they must be separated by commas.

Method

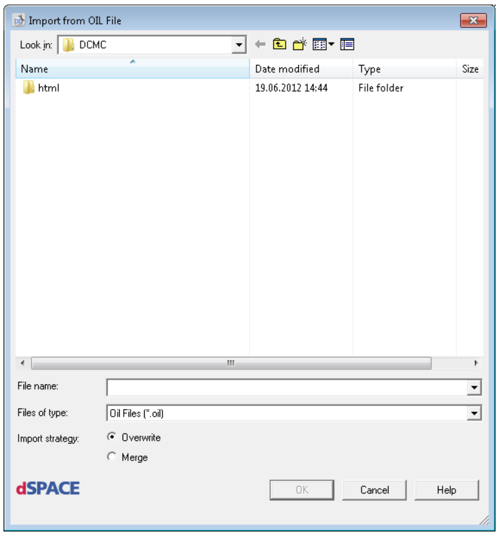
To import an OIL file via the Data Dictionary Manager

1 From the menu bar, select File – Import – from OIL File.

Note

OIL files can be imported only if DD0 is the active workspace.

The following Import from OIL file dialog opens.



2 In the dialog, select the OIL file to be imported.

3 Select the import mode. For more information on the import mode, refer to Import from OIL File (TargetLink Data Dictionary Manager Reference).

Result

The selected OIL file is imported into the TargetLink Data Dictionary. It is stored in the Pool area.

Related topics

HowTos

How to Export OIL Files via the TargetLink Data Dictionary Manager..................................... 192

References

Import from OIL File (📖 TargetLink Data Dictionary Manager Reference)

How to Import OIL Files via the MATLAB API

Objective

The generic `Import` command enables you to import an OIL file to the TargetLink Data Dictionary via the MATLAB API.

Note

OIL files can include further OIL files. To find an included OIL file which does not reside in the same directory as the main one, the import module needs to know the folder that the included OIL file resides in. Via the **`dsdd('SetRTOSInfoOILIncludeFileSearchPath',<objectIdentifier>,<propertyValue>);`** command, enter the path of the included OIL file. If several paths are specified, they must be separated by commas.

Preconditions

The file which is to be imported must comply with the OIL file format. For details, refer to OIL File Format on page 186.

Method

To import an OIL file via MATLAB

1 In the MATLAB Command Window, type

```
errCode = dsdd('Import','Format','OIL',…
   'File', '<fileName>'…
   [, '<propertyName1>', '<propertyValue1>',…]);
```

Result

If the command was successful (errCode = 0), the specified OIL file is imported to the TargetLink Data Dictionary. If the command was not successful (errCode ≠ 0), you can use `dsdd_check_msg(errCode)` to get an error message.

Example

The following example shows how you can import an OIL file into the Data Dictionary, overwriting existing data.

```
% import OIL file
```

```
errCode = dsdd('Import','Format','OIL', 'File', 'multirate_osek.oil', ...
   'ImportMode', 'Merge');
if dsdd_check_msg(errCode), return; end
```

Related topics

Basics

HowTos

Exporting OIL Files

Introduction

The TargetLink Data Dictionary serves as a common platform for data exchange, allowing you to export OIL files.

Where to go from here

Information in this section

Basics of Exporting OIL Files

RTOS objects in the TargetLink Data Dictionary

In the TargetLink Data Dictionary, there are two locations for the DD RTOS objects which can be exported into an OIL file:

- The `Pool` area (`/Pool/RTOS`)
- The `Subsystems` area (`/Subsystems/<Subsystem>/RTOS`)

OIL export from Pool area

During OIL file export from the Pool area, the DD objects stored in `/Pool/RTOS` are read and saved in the newly created OIL export file. These are objects which were previously imported or which you created.

OIL export from Subsystems area

The Subsystems area can contain several DD Subsystem objects that you can export an OIL file from. During export, the Data Dictionary reads the OIL elements from the DD RTOS objects contained in the selected DD Subsystem objects and stores them in the OIL file. Each DD RTOS object contained in a DD Subsystem object stores OSEK elements which are used by production code described by the DD Subsystem object. It is possible to export an OIL file from more than one DD Subsystem object.

OIL export from Pool and Subsystems areas

During OIL file export from the DD Pool and Subsystems areas, the DD objects stored in one or more subsystem-specific DD RTOS objects (`/Subsystems/<Subsystem>/RTOS`) are merged with the DD objects of the `Pool/RTOS` area and then saved in the newly created OIL export file. Merging is necessary because the Subsystems area contains only OIL objects used and/or created by the Code Generator. If settings in the DD Pool area and in the DD Subsystems area are inconsistent, the settings of the DD Subsystems area are exported.

How to Export OIL Files via the MATLAB API

Objective

The generic command Export enables you to export RTOS data to OIL files via the MATLAB API.

Precondition

The data to export must be in the workspace DD0.

Method

To export an OIL file via the MATLAB API

1 At the prompt of the MATLAB Command Window, type

```
errCode = dsdd('Export','Format','OIL',...
   [, '<propertyName1>', '<propertyValue1>', ...]);
```

Result

If the command was successful (errCode = 0), the specified RTOS data is exported from the TargetLink Data Dictionary to an OIL file. If the command was not successful (errCode ≠ 0), you can use `dsdd_check_msg(errCode)` to get an error message.

Example

The following example shows how you can export an OIL file from the Data Dictionary via the MATLAB API.

```
% export RTOS data to the OIL file
errCode = dsdd('Export','Format', 'Oil',…
   'File','mulirate_osek.oil',…
   'ExportFromPool', 'on',…
   'UseImportedFileStructure', 'off',…
   'Subsystems', 'controller');
if dsdd_check_msg(errCode), return;
```

Related topics

HowTos

How to Import OIL Files via the MATLAB API.. 189

How to Export OIL Files via the TargetLink Data Dictionary Manager

Objective

To export OIL files via the TargetLink Data Dictionary Manager.

Preconditions

- The TargetLink Data Dictionary Manager must be open.
- The data to export must be in the workspace DD0.

Method

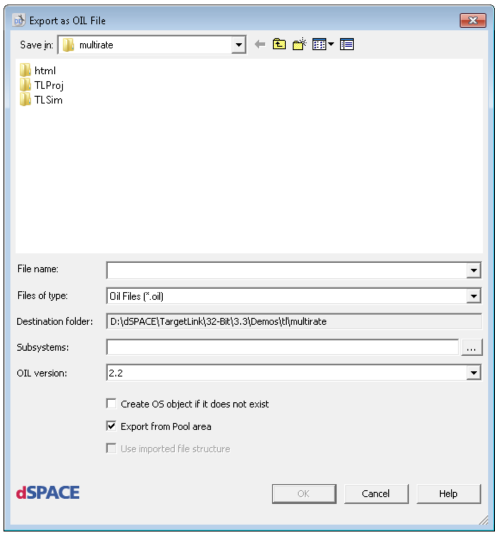
To export an OIL file via the TargetLink Data Dictionary Manager

1 From the menu bar, select File – Export – as OIL File.

Note

OIL files can be exported only if DD0 is the active workspace.

The Export as OIL File dialog opens.



Save in: multirate

File name:

Files of type: Oil Files (*.oil)

Destination folder: D:\dSPACE\TargetLink\32-Bit\3.3\Demos\tl\multirate

Subsystems:

OIL version: 2.2

☐ Create OS object if it does not exist

☑ Export from Pool area

☐ Use imported file structure

2 In the dialog, select the folder and the name for the OIL file.

3 Locate the **Subsystems** edit field and click **Browse** to select the subsystems which you want to export.

4 Set the optional parameters for the OIL file export and click **OK**. For more information on these parameters, refer to Export as OIL File (📖 TargetLink Data Dictionary Manager Reference).

Result

The selected DD objects are exported to the specified OIL file.

Related topics

HowTos

How to Import OIL Files via the Data Dictionary Manager.. 188

References

Export as OIL File (📖 TargetLink Data Dictionary Manager Reference)

Exchanging XML Files

Introduction

As a common platform for data exchange, the TargetLink Data Dictionary allows you to export XML files.

Where to go from here

Information in this section

Introduction to XML Export

Basics on the XML File Format

eXtensible Markup Language (XML)

XML is a markup language to identify structured information in documents. It is a subset of SGML (Standard Generalized Markup Language), from which it is derived. XML omits the more complex parts of SGML and makes it much easier to:

- Write applications to handle document types
- Author and manage structured information
- Share structured information across various computing systems

The XML file format was developed under W3C (World Wide Web Consortium), the main standardization body for the World Wide Web.

Purpose and content of XML files

The XML standard is becoming increasingly important for exchanging a wide variety of data on the Web and elsewhere. XML files hold text and data and provide information on the structure of documents.

Supported version

The TargetLink Data Dictionary allows you to import and export XML files. It supports file versions 1.0 and 1.1 for file import and file export.

File format specification

The W3C has specified the XML file format. You can download this specification and further information on the XML format from http://www.w3.org/XML.

Exporting XML Files

Where to go from here

Information in this section

Basics of Exporting XML Files

Export modes

There are two modes for the XML file export from the TargetLink Data Dictionary:

- Simple export mode
- Extended export mode

Simple export mode

The simple export mode allows for exporting any DD object, which results in the loss of all information related to the Data Dictionary data model, e.g., object kind. If the XML file created by this export is imported into the **TargetLink Data Dictionary**, only generic DD objects are created. You cannot use an XML file created with the simple export mode to recover your DD later.

Extended export mode

The extended export mode allows the export of any DD object, and all data-model-specific DD information is kept. If the XML file created in this way is imported into the **TargetLink Data Dictionary**, all DD objects comply with the TargetLink Data Dictionary data model.

To ensure that XML files exported in extended export mode can be reimported without reformatting or adapting them, the TargetLink Data Dictionary XML File Import and Export module issues a warning if the data could not be exported completely. To check if your DD Project file is valid, you can use the `Validate` command of the TargetLink Data Dictionary.

Validation

During the export process, you can validate the resulting XML file against the selected schema (XSD or DTD File). To do this, select a schema in the XML export pane or use the API command. For more information, refer to Export as XML File (TargetLink Data Dictionary Manager Reference).

Data model revision

The current data model revision number of the DD is saved in the exported XML file.

Related topics

HowTos

References

Export as XML File (TargetLink Data Dictionary Manager Reference)

How to Export XML Files via the TargetLink Data Dictionary Manager

Objective

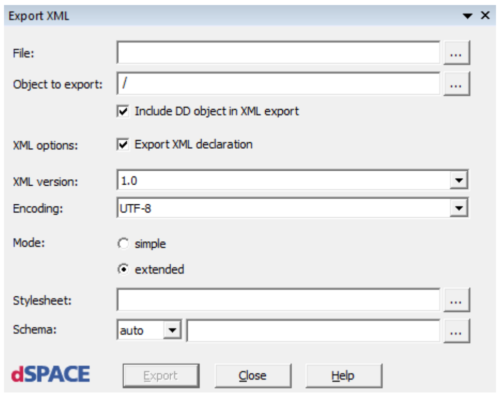
The simple XML export mode of the TargetLink Data Dictionary enables you to map DD objects to basic elements of the XML file. The extended XML export mode allows you to store data dictionaries in text form. The same export dialog is used for both the simple and the extended export mode.

Method

To export an XML file via the TargetLink Data Dictionary Manager

1 From the menu bar, choose File – Export – as XML file.

The following Export as XML File dialog opens.


Export XML

File:

Object to export: /

[✓] Include DD object in XML export

XML options: [✓] Export XML declaration

XML version: 1.0

Encoding: UTF-8

Mode: ( ) simple [✓ extended]

Stylesheet:

Schema: auto

2 In the dialog, select an existing XML file to overwrite or type the name of an XML file to which the selected DD object and its child objects are to be exported.

3 Select the DD object to be exported.

4 Select the export mode.

5 If you want to define your export in more detail, you can specify further parameters. For more information on these parameters, refer to Export as XML File (📖 TargetLink Data Dictionary Manager Reference).

Result

The selected DD object and its descendants are exported from the **TargetLink Data Dictionary** and the selected DD elements are mapped to the specified XML file.

Related topics

HowTos

How to Import XML Files via the TargetLink Data Dictionary Manager.................................... 200

References

Export as XML File (📖 TargetLink Data Dictionary Manager Reference)

How to Export XML Files via the MATLAB API

Objective

The generic command `Export` enables you to export DD objects to XML files via the MATLAB API.

Method

To export an XML file via the MATLAB API

1 In the MATLAB Command Window, type one of the following syntaxes:

Syntax 1	Syntax 2

`%export DD object's children`
`errCode = dsdd('Export','Format','XML',...`
`   'File','<fileName>',...`
`   'Parent','<DDObject>'...`
`   [, propertyName, propertyValue,…]);`
	`%export whole DD object`
	`errCode = dsdd('Import','Format','XML',...`
	`   'File','<fileName>',...`
	`   'Root','<DDObject>'...`
	`   [, propertyName, propertyValue,…]);`

Result

If the command was successful (errCode = 0), the selected DD object and all its descendants are exported to the XML file. If the command was not successful (errCode ≠ 0), you can use `dsdd_check_msg(errCode)` to get an error message or `dsdd('GetMessageList')` to retrieve all errors and warnings that occurred during export.

Example

The following example shows how you can export an XML file from the Data Dictionary.

```
% export Config/Units object to XML file
errCode = dsdd('Export','Format','XML',...
   'File','Units.xml',...
   'Root', '/Config/Units');
if dsdd_check_msg(errCode), return;
end
```

Related topics

HowTos

How to Import XML Files via the MATLAB API.. 201

References

dsdd_check_msg (TargetLink API Reference)

Importing XML Files

Introduction

As a common platform for data exchange, the TargetLink Data Dictionary allows you to import XML files.

Where to go from here

Information in this section

Basics on Importing XML Files

Import modes

There are two modes for the XML file import to the TargetLink Data Dictionary:
- Simple import mode
- Extended import mode

Simple import mode

In the simple mode, any XML file can be imported into the TargetLink Data Dictionary. The DD objects and DD properties which are created during the import correspond to specific XML nodes and attributes. The created DD items are generic, meaning that they do not match the TargetLink Data Dictionary data model.

Note

In the simple mode, a selected XML file can be imported into any generic DD object. However, the TargetLink Data Dictionary becomes invalid if you try to import an XML file into a DD area where the TargetLink Data Dictionary Model does not allow generic objects.

Extended import mode

The extended import mode can be used to import XML files that contain DD items (DD objects and DD properties) and its hierarchy. The DD items created in this way comply with the TargetLink Data Dictionary data model.

Note

In the extended mode, the selected XML file can be imported only to DD objects which conform with the contents of the imported XML file and the TargetLink Data Dictionary data model.

Validation

During the import process, you can validate the XML file against a selected schema (XSD or DTD File). To do this, select a schema in the XML import pane or use the API command. For more information, refer to Import from XML File (📖 TargetLink Data Dictionary Manager Reference).

Robust XML import

The XML import of the Data Dictionary is tolerant to invalid XML files. This helps to ensure that XML files exported from previous versions of the Data Dictionary can be imported into the Data Dictionary of TargetLink 5.x.

Data model revision

When importing an XML file, TargetLink checks whether the revision number of the XML file and the data model match, which ensures compatibility. If the revision numbers do not match, an upgrade dialog opens for upgrading the DD. If you use the `Import` API command, you can skip the upgrade dialog.

Related topics

HowTos

References

dsdd('Import','Format','XML'...) (📖 TargetLink Data Dictionary Reference)
Import (📖 TargetLink Data Dictionary Reference)
Import from XML File (📖 TargetLink Data Dictionary Manager Reference)
Mapping DD Objects to XML File Elements (📖 TargetLink Data Dictionary Reference)
Mapping in Extended Import and Export Mode (📖 TargetLink Data Dictionary Reference)
Mapping in Simple Import and Export Mode (📖 TargetLink Data Dictionary Reference)

How to Import XML Files via the TargetLink Data Dictionary Manager

Objective

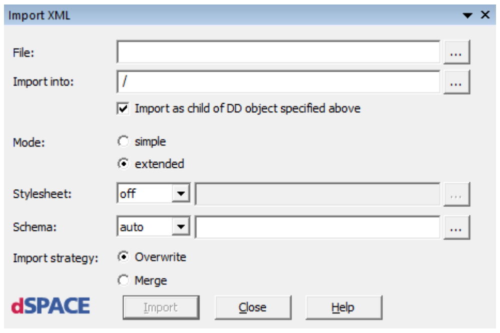
The simple XML import mode of the TargetLink Data Dictionary enables you to map any type of XML file to DD objects. The extended XML import mode allows you to store valid DD project files in text form. The same import dialog is used for both the simple and the extended import modes.

Method

To import an XML file via the TargetLink Data Dictionary Manager

1 From the menu bar, choose File – Import – from XML file.

The following Import from XML File dialog opens.



Import XML

File:
Import into: /
[✓] Import as child of DD object specified above
Mode: ○ simple [● extended]
Stylesheet: off
Schema: auto
Import strategy: [● Overwrite] ○ Merge

2 In the dialog, select the XML file to be imported.

3 Select the DD object to import the XML file to.

4 Select the import mode.

5 If you want to define your import in more detail, you can specify additional parameters. For more information refer to Import from XML File (📖 TargetLink Data Dictionary Manager Reference).

Result

The selected XML file is imported to the TargetLink Data Dictionary and mapped to the specified DD objects.

Related topics

HowTos

How to Export XML Files via the TargetLink Data Dictionary Manager.................................... 196

References

Import from XML File (📖 TargetLink Data Dictionary Manager Reference)

How to Import XML Files via the MATLAB API

Objective

The generic command `Import` enables you to import an XML file to the TargetLink Data Dictionary via the MATLAB API.

Preconditions

The file to be imported must comply with the XML file format. For details, refer to Basics on the XML File Format on page 194.

Method

To import an XML file via the MATLAB API

1 In the MATLAB Command Window, type one of the following syntaxes:

Syntax 1	Syntax 2

`%import as children to a parent object`
`errCode = dsdd('Import','Format','XML',...`
`   'File','<fileName>',...`
`   'Parent','<pathToParentDDObject>'...`
`   [, propertyName, propertyValue,…]);`
	`%import a whole object`
	`errCode = dsdd('Import','Format','XML',...`
	`   'File','<fileName>',...`
	`  'Root','<pathToDDObject>' ...`
	`   [, propertyName, propertyValue,…]);`

Result

If the command was successful (errCode = 0), the specified XML file is imported to the TargetLink Data Dictionary. If the command was not successful (errCode ≠ 0), you can use `dsdd_check_msg(errCode)` to get an error message.

Example

The following example shows how you can import an XML file into the Data Dictionary.

```
% import XML file into /Config/Units node
errCode = dsdd('Import','Format','XML','File','Units.xml',...
         'Parent', '/Config/Units',...
         'Mode', 'extended');
if dsdd_check_msg(errCode), return;end
```

Related topics

Basics

HowTos

References

dsdd_check_msg (📖 TargetLink API Reference)
Import from XML File (📖 TargetLink Data Dictionary Manager Reference)

Exchanging Variable Objects With MATLAB

Introduction

Variables of Simulink or Stateflow models are often kept in separate MATLAB files. To make these variables available in the DD, you can import them. You can work with these variables in the DD and then export them back to MATLAB later.

Where to go from here

Information in this section

Introduction to Exchanging Variable Objects

Basics on Importing and Exporting Data Between MATLAB and DD Variable Objects

Exchanging Variable objects

If you work with both MATLAB and the Data Dictionary, it is often useful to exchange data between the two. This is done by the DD tool named MATLAB Import Export (MLIE). Because it cannot be used for all use cases and environments, it is provided as a kind of demo that you can use separately or as the basis for your customized version.

MATLAB import export (MLIE)

To fulfill its tasks, MLIE works together with the DD and MATLAB. Its tasks are:

- To import variable objects from MATLAB workspaces (base or model workspace) or from MATLAB files (`.m` and `.mat`) into the DD.
- To export **Variable** objects from the DD into the MATLAB structure or into MATLAB files.

MLIE handles simple variables and also Simulink data objects (SDOs).

The word *variable object* denotes simple variables and SDOs, where it is not necessary to distinguish between the two.

Accessing MLIE

There are two ways to access MLIE:

- Via the MATLAB Command Window (MLIE API)
- Via the Data Dictionary Manager dialogs for import and export (MLIE GUI)

The first method requires a thorough knowledge of MATLAB and of M script coding. The second method gives you intuitive access to import and export functionalities.

Functional range of MLIE

No matter whether you start MLIE via MATLAB or via the Data Dictionary Manager, its functional range is nearly the same.

These functionalities are available only if you use the MLIE API to perform import and export tasks:

- Using filters during import
- Deleting value properties during import
- Adapting MLIE to your own needs

Limitations

There are a few limitations on working with MLIE. For more information, refer to MATLAB Import/Export Limitations (MLIE) on page 213.

Related topics

Basics

Basics on Importing and Exporting Data (📖 TargetLink Data Dictionary Basic Concepts Guide)
Customization Files For The MATLAB Import Export API (MLIE) (📖 TargetLink File Reference)
Exporting Variable Objects.. 208
Importing Variable Objects.. 204

Importing Variable Objects

Where to go from here

Information in this section

How to Import Variable Objects via the Data Dictionary Manager

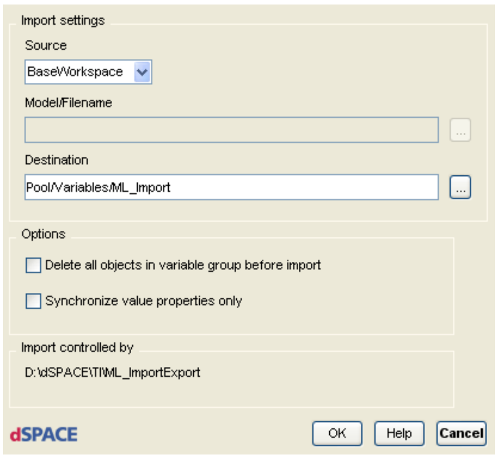
Objective

To store and manage variable objects from MATLAB workspaces or MATLAB files in the DD workspace (DD0), you can import them into the DD via an import dialog.

Method

To import variable objects via the Data Dictionary Manager

1 From the menu bar, select Tools – Import MATLAB Variable Objects.
The following Import MATLAB Variable Objects dialog opens.


Import settings
Source
BaseWorkspace
Model/Filename
Destination
Pool/Variables/ML_Import
Options
☐ Delete all objects in variable group before import
☐ Synchronize value properties only
Import controlled by
D:\dSPACE\TI\ML_ImportExport

2 In the dialog, set the import parameters. For more information on these parameters, refer to Import MATLAB Variable Objects (📖 TargetLink Data Dictionary Manager Reference).

Result

The selected MATLAB variable objects are imported to the specified variable group in the DD according to the defined import parameters.

Related topics

HowTos

How to Export Variable Objects via the Data Dictionary Manager... 208

How to Import Variable Objects via the MATLAB Import Export API

Objective

You can import variable objects from MATLAB into the primary workspace DD0 and manage them with the Data Dictionary Manager.

Preconditions

- MATLAB's current directory must be set to `<TL_InstRoot>\Demos\TL\ML_ImportExport`.
- Variable objects must exist in the MATLAB Base Workspace.

> **Tip**
>
> In the MATLAB Base Workspace, you often find automatically created answer variables like `ans = 1`. If these are not needed for further processing, it is useful to delete them before the import or to use a filter. For more information on filters, refer to How to Use Filters During Import on page 207.

Method

To import variable objects via the MATLAB Import Export API

1 In the MATLAB Command Window, type `dsdd_mlie_import`.

Result

All the variable objects available in the MATLAB Base Workspace are imported into the primary workspace (DD0) with their original names. A new DD variable group named ML_Import is created below `/Pool/Variables` to store all the imported variable objects.

> **Tip**
>
> - The method presented here describes how to import variable objects from the MATLAB Base Workspace into the DD without properties. You can also import variable objects from MATLAB files (`M.` and `MAT.`) by entering **`dsdd_mlie_import ('file,'<filename>')`** in the MATLAB Command Window.
> - The Data Dictionary Manager has an import dialog for easier input handling, which you can also open via the MATLAB Command Window by typing `dsdd_mlie_import_dlg.`

Related topics

HowTos

How to Export Variable Objects via the MATLAB Import Export API	209
How to Use Filters During Import	207

How to Use Filters During Import

Objective

To simplify the import of very many MATLAB variable objects, filters can be used. They allow you to import variable objects from MATLAB workspaces or from MATLAB files into the primary workspace (DD0).

Filter mechanism

You must implement the filter mechanism as a callback function. The filter can be set to include or exclude variable objects during the import.

Filter criteria

A filter can be applied to object names and object classes.

Use of several filters

The filter callback can recursively call other filter callbacks.

Method

To use a filter during import

1. Open a MATLAB file and create a callback function. The function signature must be as follows:
   `function is Imported = <functionname> (VariableName, VariableClass)`
2. Save the MATLAB file under the same name as the name of the callback function contained in it.
3. In the MATLAB Command Window, type `dsdd_mlie_import ('filter', '<functionname>')`

Result

All the variable objects in the MATLAB Base Workspace which match the defined filter are imported into the primary workspace (DD0) with their original names. They are stored in a new DD variable group named ML_Import below `/Pool/Variables`.

Example

The following example shows a filter callback function:

```
function isImported = mlie_test_filter(VariableName, VariableClass)
   isImported = 1; % default: import
   % do not import "ans"
   if (strcmp(VariableName, 'ans'))
      isImported = 0;
      return;
   end; % if
   % do not import Simulink.Parameter objects
   if (strcmp(VariableClass, 'Simulink.Parameter'))
      isImported = 0;
      return;
   end; % if
return;
```

If this callback function is used as a filter mechanism during import, all the variable objects which match the `If` statements are filtered out by setting `isImported` to `0`. All the other objects are imported.

Exporting Variable Objects

Where to go from here

Information in this section

How to Export Variable Objects via the Data Dictionary Manager

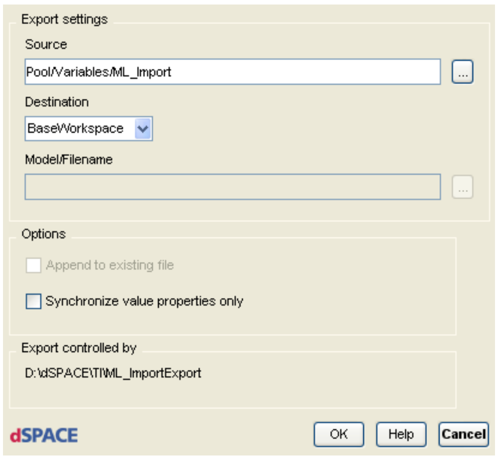
Objective

Variable objects from the DD workspace (DD0) can be exported to MATLAB workspaces or separate MATLAB files to provide them to other users: for example, to users who are not allowed to see all the coding information contained in a TargetLink model or who do not use TargetLink, but need the information to initialize Simulink models.

Method

To export Variable objects via the Data Dictionary Manager

1 From the menu bar, select Tools – Export DD Variable Objects.
The following Export DD Variable Objects dialog opens.


Export settings
Source
Pool/Variables/ML_Import
Destination
BaseWorkspace
Model/Filename

Options
Append to existing file
Synchronize value properties only
Export controlled by
D:\dSPACE\TI\ML_ImportExport

2 In the dialog, set the parameters for the export. You can only select DD objects in the primary workspace (DD0) as the source. For more information on these parameters, refer to Export DD Variable Objects (📖 TargetLink Data Dictionary Manager Reference).

Result

The selected DD **Variable** objects are exported to the MATLAB workspace according to the defined export parameters.

Related topics

HowTos

How to Import Variable Objects via the Data Dictionary Manager.. 205

How to Export Variable Objects via the MATLAB Import Export API

Objective

You can export **Variable** objects from the DD into MATLAB, for example, if you need to access the variable objects outside the DD.

Preconditions

Below `/Pool/Variables` there must be a DD variable group named `ML_Import` from which variable objects can be exported. In addition, MATLAB's current directory must be set to `<TL_InstRoot>\Demos\TL\ML_ImportExport`.

Method

To export Variable objects via the MATLAB Import Export API

1 In the MATLAB Command Window, type `dsdd_mlie_export`.

Result

All the **Variable** objects of the variable group ML_Import and of its subgroups are exported from the DD into the MATLAB Base Workspace. If variable objects with the same name as the exported ones already exist in the base workspace, they are updated if their object class matches. If this is not the case, they are replaced.

Tip

- The method presented here describes how to export Variable objects from the DD into the MATLAB Base Workspace without properties. You can also export Variable objects from the DD into separate MATLAB files (`M` and `MAT`) by entering **`dsdd_mlie_export ('file', '<filename>')`** in the MATLAB Command Window.
- The Data Dictionary Manager has an export dialog for easier input handling, which you can also open via the MATLAB Command Window by typing `dsdd_mlie_export_dlg.`

Related topics

HowTos

Customizing the MATLAB Import Export API (MLIE)

Where to go from here

Information in this section

Basics on Customizing the MATLAB Import Export API

Introduction

You can adapt the MATLAB Import Export (MLIE) API to your needs: for example, if you need to change the default mapping of object properties or if you work with your own Simulink objects.

Provided files

`<TL_InstRoot>\Demos\TL\ML_ImportExport\Private` contains the customization files which control the behavior of MLIE.

These files are all either directly or indirectly called by the files `dsdd_mlie_import.m` and `dsdd_mlie_export.m`, which are located in `<TL_InstRoot>\Demos\TL\ML_ImportExport`. These two files call the callback functions for each object to be processed and rejected by a filter. One object is processed at a time.

Source and target objects

Variable objects are either source or target objects, depending on whether they are imported or exported:

	Source Object Localization	*Target Object Localization*
Import	MATLAB workspaces or in MATLAB files	DD below the given variable group
Export	DD below the given variable group	MATLAB workspaces or in MATLAB files

Tasks of callback functions

Callback functions are responsible for:

- Retrieving properties from the currently processed source object
- Creating the corresponding target object
- Converting source properties to a suitable form for the target object
- Adding or saving source properties to the target object

Callback function signature

The signature of the callback function is as follows: `function wasNotHandled = mlie_default_import_cb(VarName, VarValue, st_Options)`

The following table describes the signature:

Signature Component	**Description**
wasNotHandled	The return value has to be set to 0 or 1: ▪ The value 0 indicates that the callback handled the current source object. ▪ The value 1 indicates that the callback did not handle the current source object. The default callback therefore tries to import the object. Thus, you only need to take care of objects which the default cannot handle or for which you want to change the behavior of the default callback.
VarName	Character array that holds the name of the current object.
VarValue	All the values (contents) of the current object.
st_Options	A structure containing the options that were given to the dsdd_mlie_import/export scripts. The callback acts according to the specified options, for example, the File or VarGroup options.

Related topics

Basics

Customization Files For The MATLAB Import Export API (MLIE) (📖 TargetLink File Reference)

HowTos

How to Customize the MATLAB Import Export API

Objective

You can adapt the MATLAB Import Export (MLIE) API to your requirements: for example, if you need to change the default mapping of object properties or if you work with your own Simulink objects.

Possible methods

There are two possible methods to customize the MLIE API:

- You can edit one or more of the provided files that contain the specific functions you want to change.

  For more information, refer to Method 1.
- You can write your own callback functions for import/export that handle the full task of importing/exporting and property mapping between MATLAB/Simulink and the DD.

  For more information, refer to Method 2.

Method 1

How to customize the MATLAB Import Export API by editing files

Tip

It is recommended to make a copy of the whole demo folder and to work only with the copied files. Then you can always return to the original files again. Moreover, your changes will not be overwritten by future patch versions of TargetLink.

1 View and edit the files in the `private` subfolder.

Result

The MLIE API is customized according to your changes.

Method 2

How to customize the MATLAB Import Export API by writing your own callback function

1 In a MATLAB file create your own callback function called `<function>`.

2 Save the MATLAB file as `<function>.mat` in one of these locations:
   - On the MATLAB path
   - In the `private` subfolder, if you want to use any of the existing helper functions from inside your callback function.

Result

The MLIE API is customized according to your changes. It uses your callback function if the property value pair 'Mapping', '<yourfunction name>' is specified as an option for `dsdd_mlie_import/export().`

Related topics

Basics

MATLAB Import/Export Limitations (MLIE)

Introduction

When working with the MATLAB Import Export tool, you should be aware of some limitations and problems you might experience.

MATLAB Import/Export Limitations (MLIE)

Introduction

The following limitations apply to the current release of TargetLink. Keep them in mind when working with MLIE.

Supported workspace

Only the DD0 workspace is supported for import and export.

DD Variants

DD variants cannot be imported or exported in one data structure. If variables with variants exist, MLIE always uses the default variant.

Variable object names	For the export to work, Variable object names in the DD must be restricted to MATLAB identifiers.
Numerics	MLIE import is restricted to numerical variables and maximum two-dimensional real matrices. Other variable kinds, for example chars or enumerations, are not supported.
Complex data structures	Complex data structures in MATLAB initialization files are not supported. You have to extend the functionality of the import script accordingly. For more information, refer to How to Customize the MATLAB Import Export API on page 212.
MATLAB files	If a MATLAB file consists of data which MLIE does not support or which is not TargetLink-related, the data is only partially imported. This partial import is particularly likely if all model initialization data (TargetLink-related and others) is kept in one file only. You should avoid partial import by removing unsupported and/or non-TargetLink-related data structures from your file.
MATLAB structures	MLIE does not support MATLAB structures.
DD variables of simple data type	DD variables of simple data type with no value specified are not exported. They are of no use in MATLAB anyway.
Synchronization of Simulink Data Objects	The synchronization of Simulink data objects (SDOs) works only if the SDOs were imported into the DD via MLIE at least once previously.
Simulink.Bus	Simulink.Bus objects are not supported.
Related topics	HowTos How to Customize the MATLAB Import Export API........ 212

Integrating TargetLink Code in External Applications

Introduction

TargetLink was developed with a focus on clean and readable code. You can therefore easily extract generated source files from the TargetLink environment and integrate them in your own ECU C modules.

Where to go from here

Information in this section

How to Export Generated Files from the TargetLink Environment

Objective

TargetLink offers a **File Export Utility**, which lets you export generated files from the TargetLink environment to a selected destination folder.

Precondition

You generated code for your model.

If you want to export the code for production use, the **Clean code** checkbox on the **Code Generation** page of the **TargetLink Main Dialog** should be selected when generating code. This ensures that the production code is generated without logging macros and with full optimization. For details on the effects of logging macros, refer to Logging by Instrumenting the Code (Log Macros) (📖 TargetLink Preparation and Simulation Guide).

Restrictions

This is a step-by-step instruction to export files via the TargetLink File Export Utility. Alternatively, you can use the `tl_export_files` API command .

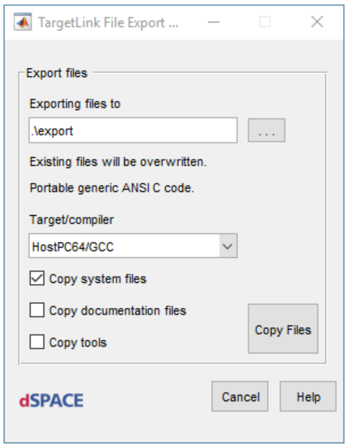
Method

To export generated files from the TargetLink environment

1 Go to the Tools page of the TargetLink Main Dialog.

2 If you want to export documentation files, click the Generate documentation button to generate documentation for your model. For details, refer to Generating Documentation on Model Characteristics on page 23.

3 Click Export Files to open the TargetLink File Export Utility dialog.



4 In the TargetLink File Export Utility dialog, select the destination folder either via the Browse button or by entering a folder name in the Destination edit field. If the folder does not exist, a dialog opens asking you if you want to create it.

5 From the Target/compiler list, select a target processor/compiler you want to export the files for.

Note

The list contains only directories that are compatible with the target processor/compiler used for the last code generation run.

6 Optionally, you can make the following settings if required:

Purpose	Setting
To export TargetLink system header files and standard libraries such as `dsfxp.lib`	Copy system files checkbox
To export documentation files	Copy documentation files checkbox

Purpose	Setting
To export the Data Dictionary project file, a list of exported files and sample M scripts.	Copy tools checkbox

7 Click Copy files to close the TargetLink File Export Utility dialog.

Result

You exported files from the TargetLink environment. Depending on your settings for the export, the following files were exported:

File/Folder	Description
source code files	-
`ReadMe.txt`	Contains a list and concise descriptions of the exported files as well as guidance about the integration of the generated files into a project.
system header files and standard libraries[1)]	-
`_doc`[1)]	Contains the generated documentation.
`_tools`[1)]	Contains the following files: ▪ the Data Dictionary project file ▪ A file named `exported_files.m`, that contains a list of all exported files ▪ Sample M-scripts for postprocessing, for example, to generate an ASAM MCD-2 MC file for the final ECU application

[1)] Generated only if the relevant checkbox was selected for the export.

Tip

You can customize the file export process via the following hook scripts:
`tl_pre_export_files_hook` (📖 TargetLink File Reference)
`tl_post_export_files_hook` (📖 TargetLink File Reference)
For an introduction to hook scripts, refer to Basics on Using Hook Scripts for Customization (AUTOSAR) on page 184.

Related topics

Basics

HowTos

References

File Export Utility (TargetLink Tool and Utility Reference)
tl_export_files (TargetLink API Reference)
tl_post_export_files_hook (TargetLink File Reference)

How to Use TargetLink Code in a Compiler IDE

Objective

If you want to use TargetLink code in a compiler IDE, you have to perform the following steps.

Method

To use TargetLink code in a Compiler IDE

1. Load the generated C files for the TargetLink subsystems into your IDE project, for example, the `picontroller.c` file for the subsystem picontroller.
2. Make sure that the `include` subfolder, located in the export destination folder, is contained in the search path for header files.

   Whether the `include` subfolder exists or not depends on certain circumstances, for example if 64-bit operations are needed or code for a FIR Filter block is generated. The non-existence of this folder means, that there are no additional header files needed for library functions or macros.

   Depending on the code generation options on the Code Generation page of the TargetLink Main Dialog, an object library was copied to the `lib` subfolder of the export destination folder:

   - `dsfxp.lib` for generic ANSI C code
   - `dsfxp_c.lib` for target- and compiler-specific optimized ANSI C code
   - `dsfxp_a.lib` for target- and compiler-specific optimized code that does not comply with ANSI C rules
3. Link the corresponding library to your application.

Result

You have now set up the IDE of your compiler.

Related topics

Basics

Basics on the Code Compilation Process (TargetLink Preparation and Simulation Guide)
Basics on the Code Generation Process (TargetLink Preparation and Simulation Guide)

HowTos

How to Use TargetLink Code in a Project Based on Makefiles

Objective

If you want to use TargetLink code in a project based on makefiles, you have to perform the following steps.

Method

To use TargetLink code in a project based on makefiles

1. Provide a makefile which contains your own compiler and linker options, and the rules to compile additional C modules not generated by TargetLink.
2. Include the makefile fragment `<model>.mk` in your makefile. This file contains a list of all source files generated by TargetLink and dependency rules.
3. Make sure that the `include` subfolder, located in the export destination folder, is contained in the compiler's search path for header files.

   Whether the `include` subfolder exists or not depends on certain circumstances, for example if 64-bit operations are needed or code for a FIR Filter block is generated. The non-existence of this folder means, that there are no additional header files needed for library functions or macros.

   Depending on the code generation options on the Code Generation page of the TargetLink Main Dialog, an object library was copied to the `lib` subfolder of the export destination folder:
   - `dsfxp.lib` for generic ANSI C code.
   - `dsfxp_c.lib` for target- and compiler-specific optimized ANSI C code.
   - `dsfxp_a.lib` for target- and compiler-specific optimized code that does not comply with ANSI C rules.
4. Link the library to your application.

Result

You have now embedded TargetLink code in a project based on makefiles.

How to Embed TargetLink Code into Your Own C Modules

Objective

After you performed the steps in How to Use TargetLink Code in a Compiler IDE on page 218 you are ready to use TargetLink code in your own C modules. Before you proceed, refer to the documentation generated for the TargetLink functions to get an overview of function prototypes and global variables used in the generated code.

Method

To embed TargetLink code into your own C modules

1. Include the header files for the TargetLink code to be called from your C module at the beginning of your file.
2. Make sure that there is a corresponding variable definition in your C module for each variable of your TargetLink model that is declared externally (for example, a variable with the class `EXTERN_GLOBAL`) . These variables must have the same data type, scaling parameters and initial values as specified in TargetLink.
3. If it exists, call the restart function (`RESTART_<name>`) for each TargetLink subsystem in the model within your own initialization function.

   You can specify that a variable has to be initialized in a restart function by specifying any value in the RestartFunctionName property of the variable's VariableClass (or also via Templates). If you do so, TargetLink uses the restart function to initialize variables before the simulation (re-)starts. For details, refer to Example of Initializing Variables via Restart Functions (TargetLink Customization and Optimization Guide).
4. Call the root step function(s) from your controller task(s). To pass inputs to and receive outputs from this function, use the interface specified in the TargetLink InPort and OutPort blocks (global variables, function arguments, etc.). The interface can also contain variables from other block types. For example, you can pass a Gain value as a call-by-value parameter to this function. Use the same data types and scaling parameters. Make sure that the sample time of your task(s) and the sample time specified for the TargetLink model are identical.
5. If it exists, call the termination function (`TERM_<name>`) of the TargetLink subsystem(s) within your own termination function.
6. If it exists (and similar to the restart function), call the initialization functions `INIT_<name>` for the TargetLink subsystem(s) to reset states from your own code if required.

   TargetLink uses the Init function to reset the states of enabled, triggered, and If-Action subsystems. If the TargetLink subsystem root-level is enabled or triggered from the surrounding Simulink model and configured for state reset (States when enabling property set to reset), you have to call the function, as it cannot be called from inside the TargetLink production code.

Example

Suppose you want to integrate the code generated for the demo model PIPT1 into your C module. Your source code could look like this:

```
…
/* include model header file */
#include "picontroller.h"
…
void task_100us(void)
{
…
   /* read input variables */
   u1 = read_io(...);
   u2 = read_io(...);
   ...
   /* call generated model step function */
   picontroller();
   ...
   /* write output signal */
   write_io(y);
   …
}
…
void restart_controller_variables(void)
{
   …
   /* call model RESTART function */
   RESTART_picontroller();
   …
}
void reset_states(void)
{
   …
   /* call model INIT function */
   INIT_pidcontroller();
   …
}
```

Result

You have now embedded TargetLink code into your own C module.

Note

You can specify the names of the root functions generated by TargetLink with a Function block on the top level of the TargetLink Subsystem or with a root function template in the TargetLink Data Dictionary.

Related topics

Basics

Basics on Using Subsystems (📖 TargetLink Customization and Optimization Guide)
Basics on Variable Classes (📖 TargetLink Customization and Optimization Guide)
Controlling Subsystem Execution (📖 TargetLink Customization and Optimization Guide)
Details on Controlling the Generation of Initialization, Termination and Restart Functions for External Step Functions by Using Function Classes (📖 TargetLink Customization and Optimization Guide)

HowTos

How to Select Function Classes (📖 TargetLink Customization and Optimization Guide)
How to Specify Subsystem Interfaces (📖 TargetLink Customization and Optimization Guide)
How to Use TargetLink Code in a Compiler IDE... 218

Examples

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

Localizing Production Code Using Different Character Sets

Extended character coding

While English is a commonly used language in the specification of models, it is often desirable to enter comments or descriptions in the language best known to the user. The ASCII standard encodes 127 characters and thus cannot display all characters of different languages. Therefore, an extended character encoding standard is necessary.

Audience profile

It is assumed that you are already familiar with MATLAB and Simulink. For information on MATLAB and Simulink, refer to *Using MATLAB* and *Using Simulink* provided by MathWorks®.

Where to go from here

Information in this section

Overview of the Supported Character Sets

Character sets in TargetLink

TargetLink is closely coupled to MATLAB. Therefore, it follows the character encoding of MATLAB. This allows character encoding according to the following schemes:

- ASCII strings
- Double-byte character sets (DBCS)
- Unicode characters

Areas with Unicode support

When you create models with TargetLink, there are several fields where you can enter descriptions or names. With TargetLink Unicode support, you can enter a description in a language containing characters that are not part of the ASCII standard, for example. The following areas in TargetLink are Unicode-compliant:

- Descriptions in Simulink blocks.
- Descriptions in TargetLink blocks.
- Descriptions in TargetLink Data Dictionary entries.
- Comments in A2L files.
- Descriptions in AUTOSAR XML files.

During production code generation, these strings are passed to the production code unchanged if they can be represented in the specified character encoding. The strings are placed in comments and are accepted by most compilerseven if they contain characters that are not compliant with the ASCII standard.

Note

Specify all names that result in C identifiers, such as variables, types, and functions in ASCII characters. This is required by the C language specification. It is recommended to specify names of DD objects and properties in ASCII characters as well.

Portability of models and data

Transferring models and their data to platforms with a different character encoding might cause problems when characters are displayed. It is recommended to use UTF-8 (default) to avoid these problems. User-provided custom code must be encoded in the same character encoding as specified in the Data Dictionary.

Specifying the character encoding

For each project, you can specify a character encoding via the CharacterSet property in the `Config/General` subtree of the Data Dictionary. You can also use the following API command:`dsdd('SetGeneralCharacterSet','/Config','<character set>')`.

TargetLink supports the following values:

- UTF-8 (default)
- UTF-16
- Windows-1252 (CP1252, Western European)
- ISO8859-1 (Latin-1)
- Shift_JIS (Shift Japanese Industrial Standards)

Note

UTF-8 encoding is always used for the following files:
- HTML files
- XML files
- A2L files
- Files generated by the document generation

The character encoding specified via the CharacterSet property is not taken into account for the listed files.

Related topics

Basics

Basics on the TargetLink Data Dictionary (📖 TargetLink Data Dictionary Basic Concepts Guide)

How to Specify the Used Character Set

Objective

To specify the character set encoding used in your current project.

Specifying a character set

In the Data Dictionary Manager specify the character set encoding as required. The following values are supported by TargetLink:
- UTF-8 (default)
- UTF-16
- Windows-1252 (CP1252, Western European)
- ISO8859-1 (Latin-1)
- Shift_JIS (Shift Japanese Industrial Standards)

Note

TargetLink supports Unicode in Stateflow charts only for descriptions and annotations.

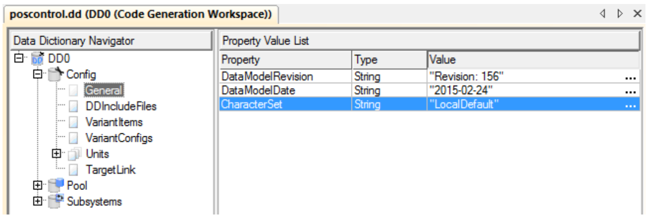
Preconditions

You have opened a TargetLink model and the related Data Dictionary. Refer to How to Start the Data Dictionary Manager (📖 TargetLink Data Dictionary Basic Concepts Guide).

Method

To specify the used character set

1 In the Data Dictionary Navigator, open the `/Config/General` subtree.



Property	Type	Value
DataModelRevision	String	"Revision: 156" ...
DataModelDate	String	"2015-02-24" ...
CharacterSet	String	"LocalDefault" ...

2 In the Property Value List specify the CharacterSet as required.

Result

You specified the character set encoding for the current project.

Note

To specify the character set, you can use the API command dsdd('SetGeneralCharacterSet', '/Config', '<character set>'), as well.

Related topics

Basics

Basics on the TargetLink Data Dictionary (TargetLink Data Dictionary Basic Concepts Guide)
Overview of the Supported Character Sets.. 223

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

1-D look-up table	A look-up table that maps one input value (x) to one output value (y).
2-D look-up table	A look-up table that maps two input values (x,y) to one output value (z).

A

Abstract interface An interface that allows you to map a project-specific, physical specification of an interface (made in the TargetLink Data Dictionary) to a logical interface of a modular unit. If the physical interface changes, you do not have to change the Simulink subsystem or the partial DD file and therefore neither the generated code of the modular unit.

Access function (AF) A C function or function-like preprocessor macro that encapsulates the access to an interface variable.

See also read/write access function and variable access function.

Acknowledgment Notification from the RTE that a data element or an event message have been transmitted.

Activating RTE event An RTE event that can trigger one or more runnables.

See also activation reason.

Activation reason The activating RTE event that actually triggered the runnable.

Activation reasons can group several RTE events.

Active page pointer A pointer to a data page. The page referenced by the pointer is the active page whose values can be changed with a calibration tool.

Adaptive AUTOSAR Short name for the AUTOSAR *Adaptive Platform* standard. It is based on a service-oriented architecture that aims at on-demand software updates and high-end functionalities. It complements Classic AUTOSAR.

Adaptive AUTOSAR behavior code Code that is generated for model elements in Adaptive AUTOSAR Function subsystems or Method Behavior subsystems. This code represents the behavior of the model and is part of an adaptive application. Must be integrated in conjunction with ARA adapter code.

Adaptive AUTOSAR Function A TargetLink term that describes a C++ function representing a partial functionality of an adaptive application. This function can be called in the C++ code of an adaptive application. From a higher-level perspective, Adaptive AUTOSAR functions are analogous to runnables in Classic AUTOSAR.

Adaptive AUTOSAR Function subsystem An atomic subsystem used to generate code for an Adaptive AUTOSAR Function. It contains a Function block whose AUTOSAR mode is set to `Adaptive` and whose Role is set to `Adaptive AUTOSAR Function`.

ANSI C Refers to C89, the C language standard ANSI X3.159-1989.

Application area An optional DD object that is a child object of the DD root object. Each Application object defines how an ECU program is built from the generated subsystems. It also contains some experiment data, for example, a list of variables to be logged during simulations and results of code coverage tests.

Build objects are children of Application objects. They contain all the information about the binary programs built for a certain target platform, for example, the symbol table for address determination.

Application data type Abstract type for defining types from the application point of view. It allows you to specify physical data such as measurement data. Application data types do not consider implementation details such as bit-size or endianness.

Application data type (ADT) According to AUTOSAR, application data types are used to define types at the application level of abstraction. From the application point of view, this affects physical data and its numerical representation. Accordingly, application data types have physical semantics but do not consider implementation details such as bit width or endianness.

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

See also implementation data type (IDT).

Application layer The topmost layer of the ECU software. The application layer holds the functionality of the ECU software and consists of atomic software components (atomic SWCs).

ARA adapter code Adapter code that connects Adaptive AUTOSAR behavior code with the Adaptive AUTOSAR API or other parts of an adaptive application.

Array-of-struct variable An array-of-struct variable is a structure that either is non-scalar itself or that contains at least one non-scalar substructure at any nesting depth. The use of array-of-struct variables is linked to arrays of buses in the model.

Artifact A file generated by TargetLink:

- Code coverage report files
- Code generation report files
- Metadata files
- Model-linked code view files
- Production code files
- Simulation application object files
- Simulation frame code files
- Stub code files

Artifact location A folder in the file system that contains an artifact. This location is specified relatively to a project folder.

ASAP2 File Generator A TargetLink tool that generates ASAP2 files for the parameters and signals of a Simulink model as specified by the corresponding TargetLink settings and generated in the production code.

ASCII In production code, strings are usually encoded according to the ASCII standard. The ASCII standard is limited to a set of 127 characters implemented by a single byte. This is not sufficient to display special characters of different languages. Therefore, use another character encoding, such as UTF-8, if required.

Asynchronous operation call subsystem A subsystem used when modeling *asynchronous* client-server communication. It is used to generate the call of the `Rte_Call` API function and for simulation purposes.

See also operation result provider subsystem.

Asynchronous server call returns event An RTE event that specifies whether to start or continue the execution of a runnable after the execution of a server runnable is finished.

Atomic software component (atomic SWC) The smallest element that can be defined in the application layer. An atomic SWC describes a single functionality and contains the corresponding algorithm. An atomic SWC communicates with the outside only via the interfaces at the SWC's ports. An atomic SWC is defined by an internal behavior and an implementation.

Atomic software component instance An atomic software component (atomic SWC) that is actually used in a controller model.

AUTOSAR Abbreviation of *AUT*omotive *O*pen *S*ystem *AR*chitecture. The AUTOSAR partnership is an alliance in which the majority of OEMs, suppliers, tool providers, and semiconductor companies work together to develop and establish a de-facto open industry-standard for automotive electric/electronics (E/E) architecture and to manage the growing E/E complexity.

AUTOSAR import/export Exchanging standardized software component descriptions between AUTOSAR tools.

AUTOSAR subsystem An atomic subsystem that contains a Function block whose AUTOSAR mode property is set to `Classic`. See also operation subsystem, operation call with runnable implementation subsystem, and runnable subsystem.

AUTOSAR tool Generic term for the following tools that are involved in the ECU network software development process according to AUTOSAR:

- Behavior modeling tool
- System-level tool
- ECU-centric tool

TargetLink acts as a behavior modeling tool in the ECU network software development process according to AUTOSAR.

Autoscaling Scaling is performed by the Autoscaling tool, which calculates worst-case ranges and scaling parameters for the output, state and parameter variables of TargetLink blocks. The Autoscaling tool uses either worst-case ranges or simulated ranges as the basis for scaling. The upper and lower worst-case range limits can be calculated by the tool itself. The Autoscaling tool always focuses on a subsystem, and optionally on its underlying subsystems.

B

Basic software The generic term for the following software modules:

- System services (including the operating system (OS) and the ECU State Manager)
- Memory services (including the NVRAM manager)
- Communication services
- I/O hardware abstraction
- Complex device drivers

Together with the RTE, the basic software is the platform for the application layer.

Batch mode The mode for batch processing. If this mode is activated, TargetLink does not open any dialogs. Refer to How to Set TargetLink to Batch Mode (TargetLink Orientation and Overview Guide).

Behavior model A model that contains the control algorithm for a controller (function prototyping system) or the algorithm of the controlled system (hardware-in-the-loop system). Can be connected in ConfigurationDesk via model ports to build a real-time application (RTA). The RTA can be executed on real-time hardware that is supported by ConfigurationDesk.

Block properties Properties belonging to a TargetLink block. Depending on the kind of the property, you can specify them at the block and/or in the Data Dictionary. Examples of block properties are:

- Simulink properties (at a masked Simulink block)
- Logging options or saturation flags (at a TargetLink block)
- Data types or variable classes (referenced from the DD)
- Variable values (specified at the block or referenced from the DD)

Bus A bus consists of subordinate bus elements. A bus element can be a bus itself.

Bus element A bus element is a part of a bus and can be a bus itself.

Bus port block Bus Inport, Bus Outport are bus port blocks. They are similar to the TargetLink Input and Output blocks. They are virtual, and they let you configure the input and output signals at the boundaries of a TargetLink subsystem and at the boundaries of subsystems that you want to generate a function for.

Bus signal Buses combine multiple signals, possibly of different types. Buses can also contain other buses. They are then called nested buses.

Bus-capable block A block that can process bus signals. Like bus port blocks, they allow you to assign a type definition and, therefore, a variable class to all the bus elements at once. The following blocks are bus-capable:

- Constant
- Custom Code (type II) block
- Data Store Memory, Data Store Read, and Data Store Write

- Delay
- Function Caller
- ArgIn, ArgOut
- Merge
- Multiport Switch (Data Input port)
- Probe
- Sink
- Signal Conversion
- Switch (Data Input port)
- Unit Delay
- Stateflow Data
- MATLAB Function Data

C

Calibratable variable Variable whose value can be changed with a calibration tool during run time.

Calibration Changing the calibration parameter values of ECUs.

Calibration parameter Any ECU variable type that can be calibrated. The term *calibration parameter* is independent of the variable type's dimension.

Calprm Defined in a calprm interface. Calprms represent calibration parameters that are accessible via a measurement and calibration system.

Calprm interface An interface that is provided or required by a software component (SWC) via a port (AUTOSAR).

Calprm software component A special software component (SWC) that provides calprms. Calprm software components have no internal behavior.

Canonical In the DD, array-of-struct variables are specified canonically. Canonical means that you specify one array element as a representative for all array elements.

Catalog file (CTLG) A description of the content of an SWC container. It contains file references and file category information, such as source code files (`C` and `H`), object code files (such as `O` or `OBJ`), variable description files (`A2L`), or AUTOSAR files (`ARXML`).

Characteristic table (Classic AUTOSAR) A look-up table as described by Classic AUTOSAR whose values are measurable or calibratable. See also compound primitive data type

Classic AUTOSAR Short name for the AUTOSAR *Classic Platform* standard that complements Adaptive AUTOSAR.

Classic initialization mode The initialization mode used when the Simulink diagnostics parameter Underspecified initialization detection is set to `Classic`.

See also simplified initialization mode.

Client port A require port in client-server communication as described by Classic AUTOSAR. In the Data Dictionary, client ports are represented as DD ClientPort objects.

Client-server interface An interface that describes the operations that are provided or required by a software component (SWC) via a port (AUTOSAR).

Code generation mode One of three mutually exclusive options for generating TargetLink standard production code, AUTOSAR-compliant production code or RTOS-compliant (multirate RTOS/OSEK) production code.

Code generation unit (CGU) The smallest unit for which you can generate code. These are:

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

Code output style definition file To customize code formatting, you can modify a code output style definition file (XML file). By modifying this file, you can change the representation of comments and statements in the code output.

Code output style sheets To customize code formatting, you can modify code output style sheets (XSL files).

Code section A section of generated code that defines and executes a specific task.

Code size Amount of memory that an application requires specified in RAM and ROM after compilation with the target cross-compiler. This value helps to determine whether the application generated from the code files fits in the ECU memory.

Code variant Code variants lead to source code that is generated differently depending on which variant is selected (i.e., varianted at code generation time). For example, if the Type property of a variable has the two variants Int16 and Float32, you can generate either source code for a fixed-point ECU with one variant, or floating-point code with the other.

Compatibility mode The default operation mode of RTE generators. The object code of an SWC that was compiled against an application header generated in compatibility mode can be linked against an RTE generated in compatibility mode (possibly by a different RTE generator). This is due to using standardized data structures in the generated RTE code.

See also vendor mode.

Compiler inlining The process of replacing a function call with the code of the function body during compilation by the C compiler via inline expansion.

This reduces the function call overhead and enables further optimizations at the potential cost of larger code size.

Composition A structuring element in the application layer. A composition consists of software components and their interconnections via ports.

Compound primitive data type A primitive application data type (ADT) as defined by Classic AUTOSAR whose category is one of the following:

- COM_AXIS
- CUBOID
- CUBE_4
- CUBE_5
- CURVE
- MAP
- RES_AXIS
- VAL_BLK
- STRING

Compute-through-overflow (CTO) Calculation method for additions and subtraction where overflows are allowed in intermediate results without falsifying the final result.

Concern A concept in component-based development. It describes the idea that components separate their concerns. Accordingly, they must be developed in such a way that they provide the required functionality, are flexible and easy to maintain, and can be assembled, reused, or replaced by newer, functionally equivalent components in a software project without problems.

Config area A DD object that is a child object of the DD root object. The `Config` object contains configuration data for the tools working with the TargetLink Data Dictionary and configuration data for the TargetLink Data Dictionary itself. There is only one `Config` object in each DD workspace. The configuration data for the TargetLink Data Dictionary is a list of included DD files, user-defined views, data for variant configurations, etc. The data in the Config area is typically maintained by a Data Dictionary administrator.

ConfigurationDesk A dSPACE software tool for implementing and building real-time applications (RTA).

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

Constrained range limits User-defined minimum (Min) or maximum (Max) values that the user ensures will never be exceeded. The Code Generator relies on these ranges to make the generated production code more efficient. If no

Min/Max values are entered, the implemented range limits are used during production code generation.

Constrained type A DD Typedef object whose Constraints subtree is specified.

Container A bundle of files. The files are described in a catalog file that is part of the container. The files of a container can be spread over your file system.

Container Manager A tool for handling containers.

Container set file (CTS) A file that lists a set of containers. If you export containers, one container set file is created for every TargetLink Data Dictionary.

Conversion method A method that describes the conversion of a variable's integer values in the ECU memory into their physical representations displayed in the Measurement and Calibration (MC) system.

Custom code Custom code consists of C code snippets that can be included in production code by using custom code files that are associated with custom code blocks. TargetLink treats this code as a black box. Accordingly, if this code contains custom code variables you must specify them via custom code symbols.. See also external code.

Custom code symbol A variable that is used in a custom code file. It must be specified on the Interface page of custom code blocks.

Customer-specific C function An external function that is called from a Stateflow diagram and whose interface is made known to TargetLink via a scripting mechanism.

D

Data element Defined in a sender-receiver interface. Data elements are information units that are exchanged between sender ports, receiver ports and sender-receiver ports. They represent the data flow.

Data page A structure containing all of the calibratable variables that are generated during code generation.

Data prototype The generic term for one of the following:

- Data element
- Operation argument
- Calprm
- Interrunnable variable (IRV)
- Shared or PerInstance Calprm
- Per instance memory

Data receive error event An RTE event that specifies to start or continue the execution of a runnable related to receiver errors.

Data received event An RTE event that specifies whether to start or continue the execution of a runnable after a data element is received by a receiver port or sender-receiver port.

Data semantics The communication of data elements with last-is-best semantics. Newly received data elements overwrite older ones regardless of whether they have been processed or not.

Data send completed event An RTE event that specifies whether to start or continue the execution of a runnable related to a sender acknowledgment.

Data transformation A transformation of the data of inter-ECU communication, such as end-to-end protection or serialization, that is managed by the RTE via transformers.

Data type map Defines a mapping between implementation data types (represented in TargetLink by DD **Typedef** objects) and application data types.

Data type mapping set Summarizes all the data type maps and mode request type maps of a software component (SWC).

Data variant One of two or more differing data values that are generated into the same C code and can be switched during ECU run time using a calibratable variant ID variable. For example, the **Value** property of a gain parameter can have the variants 2, 3, and 4.

DataItemMapping (DIM) A DataItemMapping object is a DD object that references a ReplaceableDataItem (RDI) and a DD variable. It is used to define the DD variable object to map an RDI object to, and therefore also the implementation variable in the generated code.

DD child object The DD object below another DD object in the DD object tree.

DD data model The DD data model describes the object kinds, their properties and constraints as well as the dependencies between them.

DD file A DD file (*.dd) can be a DD project file or a partial DD file.

DD object Data item in the **Data Dictionary** that can contain DD child objects and DD properties.

DD object tree The tree that arranges all DD objects according to the DD data model.

DD project file A file containing the DD objects of a DD workspace.

DD root object The topmost DD object of the DD workspace.

DD subtree A part of the DD object tree containing a DD object and all its descendants.

DD workspace An independent organizational unit (central data container) and the largest entity that can be saved to file or loaded from a DD project file. Any number of DD workspaces is supported, but only the first (DD0) can be used for code generation.

Default enumeration constant Represents the default constant, i.e., the name of an enumerated value that is used for initialization if an initial value is required, but not explicitly specified.

Direct reuse The Code Generator adds the instance-specific variables to the reuse structure as leaf struct components.

E

ECU Abbreviation of *electronic control unit*.

ECU software The ECU software consists of all the software that runs on an ECU. It can be divided into the basic software, run-time environment (RTE), and the application layer.

ECU State Manager A piece of software that manages modes. An ECU state manager is part of the basic software.

Enhanceable Simulink block A Simulink® block that corresponds to a TargetLink simulation block, for example, the **Gain** block.

Enumerated value An enumerated value consists of an enumeration constant and a corresponding underlying integer value (enumeration value).

Enumeration constant An enumeration constant defines the name for an enumerated value.

Enumeration data type A data type with a specific name, a set of named enumerated values and a default enumeration constant.

Enumeration value An enumeration value defines the integer value for an enumerated value.

Event message Event messages are information units that are defined in a sender-receiver interface and exchanged between sender ports or receiver ports. They represent the control flow. On the receiver side, each event message is related to a buffer that queues the received messages.

Event semantics Communication of data elements with first-in-first-out semantics. Data elements are received in the same order they were sent. In simulations, TargetLink behaves as if data semantics was specified, even if you specified event semantics. However, TargetLink generates calls to the correct RTE API functions for data and event semantics.

ExchangeableWidth A DD object that defines code variants or improves code readability by using macros for signal widths.

Exclusive area Allows for specifying critical sections in the code that cannot preempt/interrupt each other. An exclusive area can be used to specify the mutual exclusion of runnables.

Executable application The generic term for offline simulation applications and real-time applications.

Explicit communication A communication mode in Classic AUTOSAR. The data is exchanged whenever data is required or provided.

Explicit object An explicit object is an object in production code that the Code Generator created from a direct specification made at a DD object or at a model element. For comparison, see implicit object.

Extern C Stateflow symbol A C symbol (function or variable) that is used in a Stateflow chart but that is defined in an external code module.

External code Existing C code files/modules from external sources (e.g., legacy code) that can be included by preprocessor directives and called by the C code generated by TargetLink. Unlike Custom code, external code is used as it is.

External container A container that is owned by the tool with that you are exchanging a software component but that is not the tool that triggers the container exchange. This container is used when you import files of a software component which were created or changed by the other tool.

F

Filter An algorithm that is applied to received data elements.

Fixed-Point Library A library that contains functions and macros for use in the generated production code.

Function AF The short form for an access function (AF) that is implemented as a C function.

Function algorithm object Generic term for either a MATLAB local function, the interface of a MATLAB local function or a local MATLAB variable.

Function class A class that represents group properties of functions that determine the function definition, function prototypes and function calls of a function in the generated production code. There are two types of function classes: predefined function class objects defined in the `/Pool/FunctionClasses` group in the DD and implicit function classes (default function classes) that can be influenced by templates in the DD.

Function code Code that is generated for a modular unit that represents functionality and can have abstract interfaces to be reused without changes in different contexts, e.g. in different integration models.

Function inlining The process of replacing a function call with the code of the function body during code generation by TargetLink via inline expansion. This reduces the function call overhead and enables further optimizations at the potential cost of larger code size.

Function interface An interface that describes how to pass the inputs and outputs of a function to the generated production code. It is described by the function signature.

Function subsystem A subsystem that is atomic and contains a Function block. When generating code, TargetLink generates it as a C function.

Functional Mock-up Unit (FMU) An archive file that describes and implements the functionality of a model based on the Functional Mock-up Interface (FMI) standard.

G

Global data store The specification of a DD DataStoreMemoryBlock object that references a variable and is associated with either a Simulink.Signal object or Data Store Memory block. The referenced variable must have a module specification and a fixed name and must be global and non-static. Because of its central specification in the Data Dictionary, you can use it across the boundaries of CGUs.

I

Implementation Describes how a specific internal behavior is implemented for a given platform (microprocessor type and compiler). An implementation mainly consists of a list of source files, object files, compiler attributes, and dependencies between the make and build processes.

Implementation data type (IDT) According to AUTOSAR, implementation data types are used to define types on the implementation level of abstraction. From the implementation point of view, this regards the storage and manipulation of digitally represented data. Accordingly, implementation data types have data semantics and do consider implementation details, such as the data type.

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

See also application data type (ADT).

Implementation variable A variable in the generated production code to which a ReplaceableDataItem (RDI) object is mapped.

ImplementationPolicy A property of data element and Calprm elements that specifies the implementation strategy for the resulting variables with respect to consistency.

Implemented range The range of a variable defined by its scaling parameters. To avoid overflows, the implemented range must include the maximum and minimum values the variable can take in the simulation application and in the ECU.

Implicit communication A communication mode in Classic AUTOSAR. The data is exchanged at the start and end of the runnable that requires or provides the data.

Implicit object Any object created for the generated code by the TargetLink Code Generator (such as a variable, type, function, or file) that may not have been specified explicitly via a TargetLink block, a Stateflow object, or the TargetLink Data Dictionary. Implicit objects can be influenced via DD templates. For comparison, see explicit object.

Implicit property If the property of a DD object or of a model based object is not directly specified at the object, this property is created by the Code Generator and is based on internal templates or DD **Template** objects. These properties are called implicit properties. Also see implicit object and explicit object.

Included DD file A partial DD file that is inserted in the proper point of inclusion in the DD object tree.

Incremental code generation unit (CGU) Generic term for code generation units (CGUs) for which you can incrementally generate code. These are:

- Referenced models
- Subsystems configured for incremental code generation

Incremental CGUs can be nested in other model-based CGUs.

Indirect reuse The Code Generator adds pointers to the reuse structure which reference the indirectly reused instance-specific variables.

Indirect reuse has the following advantages to direct reuse:

- The combination of shared and instance-specific variable.
- The reuse of input/output variables of neighboring blocks.

Inline expansion The process of replacing a function call with the code of the function body. See also function inlining and compiler inlining.

Instance-specific variable A variable that is accessed by one reusable system instance. Typically, instance-specific variables are used for states and parameters whose value are different across instances.

Instruction set simulator (ISS) A simulation model of a microprocessor thatcan execute binary code compiled for the corresponding microprocessor. This allows the ISS to behave in the same way as the simulated microprocessor.

Integration model A model or TargetLink subsystem that contains modular units which it integrates to make a larger entity that provides its functionality.

Interface Describes the data elements, NvData, event messages, operations, or calibration parameters that are provided or required by a software component (SWC) via a port (AUTOSAR).

Internal behavior An element that represents the internal structure of an atomic software component (atomic SWC). It is characterized by the following entities and their interdependencies:

- Exclusive area
- Interrunnable variable (IRV)
- Per instance memory
- Per instance parameter
- Runnable
- RTE event
- Shared parameter

Interrunnable variable (IRV) Variable object for specifying communication between the runnables in one atomic software component (atomic SWC).

Interrupt service routine (ISR) function A function that implements an ISR and calls the step functions of the subsystems that are assigned by the user or by the TargetLink Code Generator during multirate code generation.

Intertask communication The flow of data between tasks and ISRs, tasks and tasks, and between ISRs and ISRs for multirate code generation.

Is service A property of an interface that indicates whether the interface is provided by a basic software service.

ISV Abbreviation for instance-specific variable.

L

Leaf bus element A leaf bus element is a subordinate bus element that is not a bus itself.

Leaf bus signal See also leaf bus element.

Leaf struct component A leaf struct component is a subordinate struct component that is not a struct itself.

Legacy function A function that contains a user-provided C function.

Library subsystem A subsystem that resides in a Simulink® library.

Local container A container that is owned by the tool that triggers the container exchange.

The tool that triggers the exchange transfers the files of a software component to this container when you export a software component. The external container is not involved.

Local MATLAB variable A variable that is generated when used on the left side of an assignment or in the interface of a MATLAB local function. TargetLink does not support different data types and sizes on local MATLAB variables.

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

M

Macro A literal representing a C preprocessor definition. Macros are used to provide a fixed sequence of computing instructions as a single program statement. Before code compilation, the preprocessor replaces every occurrence of the macro by its definition, i.e., by the code that it stands for.

Macro AF The short form for an access function (AF) that is implemented as a function-like preprocessor macro.

MATLAB code elements MATLAB code elements include MATLAB local functions and local MATLAB variables. MATLAB code elements are not available in the Simulink Model Explorer or the Property Manager.

MATLAB local function A function that is scoped to a MATLAB main function and located at the same hierarchy level. MATLAB local functions are treated like MATLAB main functions and have the same properties as the MATLAB main function by default.

MATLAB main function The first function in a MATLAB function file.

Matrix AF An access function resulting from a DD AccessFunction object whose VariableKindSpec property is set to `APPLY_TO_MATRIX`.

Matrix signal Collective term for 2-D signals implemented as matrix variable in production code.

Matrix variable Collective term for 2-D arrays in production code that implement 2-D signals.

Measurement Viewing and analyzing the time traces of calibration parameters and measurement variables, for example, to observe the effects of ECU parameter changes.

Measurement and calibration system A tool that provides access to an ECU for measurement and calibration. It requires information on the calibration parameters and measurement variables with the ECU code.

Measurement variable Any variable type that can be measured but not calibrated. The term *measurement variable* is independent of a variable type's dimension.

Memory mapping The process of mapping variables and functions to different memory sections.

Memory section A memory location to which the linker can allocate variables and functions.

Message Browser A TargetLink component for handling fatal (F), error (E), warning (W), note (N), and advice (A) messages.

MetaData files Files that store metadata about code generation. The metadata of each code generation unit (CGU) is collected in a DD Subsystem object that is written to the file system as a partial DD file called `<CGU>_SubsystemObject.dd`.

Method Behavior subsystem An atomic subsystem used to generate code for a method implementation. From the TargetLink perspective, this is an Adaptive AUTOSAR Function that can take arguments.

It contains a Function block whose AUTOSAR mode is set to `Adaptive` and whose Role is set to `Method Behavior`.

Method Call subsystem An atomic subsystem that is used to generate a method call in the code of an Adaptive AUTOSAR Function. The subsystem contains a Function block whose AUTOSAR mode is set to `Adaptive` and whose Role is set to `Method Call`. The subsystem interface is used to generate the function interface while additional model elements that are contained in the subsystem are only for simulation purposes.

Microcontroller family (MCF) A group of microcontroller units with the same processor, but different peripherals.

Microcontroller unit (MCU) A combination of a specific processor with additional peripherals, e.g. RAM or AD converters. MCUs with the same processor, but different peripherals form a microcontroller family.

MIL simulation A simulation method in which the function model is computed (usually with double floating-point precision) on the host computer as an executable specification. The simulation results serve as a reference for SIL simulations and PIL simulations.

MISRA Organization that assists the automotive industry to produce safe and reliable software, e.g., by defining guidelines for the use of C code in automotive electronic control units or modeling guidelines.

Mode An operating state of an ECU, a single functional unit, etc..

Mode declaration group Contains the possible operating states, for example, of an ECU or a single functional unit.

Mode manager A piece of software that manages modes. A mode manager can be implemented as a software component (SWC) of the application layer.

Mode request type map An entity that defines a mapping between a mode declaration group and a type. This specifies that mode values are instantiated in the software component (SWC)'s code with the specified type.

Mode switch event An RTE event that specifies to start or continue the execution of a runnable as a result of a mode change.

Model Compare A dSPACE software tool that identifies and visualizes the differences in the contents of Simulink/TargetLink models (including Stateflow). It can also merge the models.

Model component A model-based code generation unit (CGU).

Model element A model in MATLAB/Simulink consists of model elements that are TargetLink blocks, Simulink blocks, and Stateflow objects, and signal lines connecting them.

Model port A port used to connect a behavior model in ConfigurationDesk. In TargetLink, multiple model ports of the same kind (data in or data out) can be grouped in a model port block.

Model port block A block in ConfigurationDesk that has one or more model ports. It is used to connect the behavior model in ConfigurationDesk.

Model port variable A DD Variable object that represents a model port of a behavior model in ConfigurationDesk.

Model-dependent code elements Code elements that (partially) result from specifications made in the model.

Model-independent code elements Code elements that can be generated from specifications made in the Data Dictionary alone.

Modular unit A submodel containing functionality that is reusable and can be integrated in different integration models. The production code for the modular unit can be generated separately.

Module A DD object that specifies code modules, header files, and other arbitrary files.

Module specification The reference of a DD Module object at a Function Block (TargetLink Model Element Reference) block or DD object. The resulting code elements are generated into the module. See also production code and stub code.

ModuleOwnership A DD object that specifies an owner for a module (module owner) or module group, i.e. the owning code generation unit (CGU) that generates the production code for it or declares the module as external code that is not generated by TargetLink.

N

Nested bus A nested bus is a bus that is a subordinate bus element of another bus.

Nested struct A nested struct is a struct that is a subordinate struct component of another struct.

Non-scalar signal Collective term for vector and matrix signals.

Non-standard scaling A scaling whose LSB is different from 2^0 `or whose Offset is not 0.`

Nv receiver port A require port in NvData communication as described by Classic AUTOSAR. In the Data Dictionary, nv receiver ports are represented as DD NvReceiverPort objects.

Nv sender port A provide port in NvData communication as described by Classic AUTOSAR. In the Data Dictionary, nv sender ports are represented as DD NvSenderPort objects.

Nv sender-receiver port A provide-require port in NvData communication as described by Classic AUTOSAR. In the Data Dictionary, nv sender-receiver ports are represented as DD NvSenderReceiverPort objects.

NvData Data that is exchanged between an atomic software component (atomic SWC) and the ECU's NVRAM.

NvData interface An interface used in NvData communication.

NVRAM Abbreviation of *non volatile random access memory*.

NVRAM manager A piece of software that manages an ECU's NVRAM. An NVRAM manager is part of the basic software.

O

Offline simulation application (OSA) An application that can be used for offline simulation in VEOS.

Online parameter modification The modification of parameters in the production code before or during a SIL simulation or PIL simulation.

Operation Defined in a client-server interface. A software component (SWC) can request an operation via a client port. A software component can provide an operation via a server port. Operations are implemented by server runnables.

Operation argument Specifies a C-function parameter that is passed and/or returned when an operation is called.

Operation call subsystem A collective term for synchronous operation call subsystem and asynchronous operation call subsystem.

Operation call with runnable implementation subsystem An atomic subsystem that contains a Function block whose AUTOSAR mode property is set to `Classic` and whose Role is set to `Operation call with runnable implementation`.

Operation invoked event An RTE event that specifies to start or continue the execution of a runnable as a result of a client call. A runnable that is related to an operation invoked event represents a server.

Operation result provider subsystem A subsystem used when modeling *asynchronous* client-server communication. It is used to generate the call of the `Rte_Result` API function and for simulation purposes.

See also asynchronous operation call subsystem.

Operation subsystem A collective term for operation call subsystem and operation result provider subsystem.

OSEK Implementation Language (OIL) A modeling language for describing the configuration of an OSEK application and operating system.

P

Package A structuring element for grouping elements of software components in any hierarchy. Using package information, software components can be spread across or combined from several software component description (SWC-D) files during AUTOSAR import/export scenarios.

Parent model A model containing references to one or more other models by means of the Simulink Model block.

Partial DD file A DD file that contains only a DD subtree. If it is included in a DD project file, it is called Included DD file. The partial DD file can be located on a central network server where all team members can share the same configuration data.

Per instance memory The definition of a data prototype that is instantiated for each atomic software component instance by the RTE. A data type instance can be accessed only by the corresponding instance of the atomic SWC.

Per instance parameter A parameter for measurement and calibration unique to the instance of a software component (SWC) that is instantiated multiple times.

Physical evaluation board (physical EVB) A board that is equipped with the same target processor as the ECU and that can be used for validation of the generated production code in PIL simulation mode.

PIL simulation A simulation method in which the TargetLink control algorithm (production code) is computed on a microcontroller target (physical or virtual).

Plain data type A data type that is not struct, union, or pointer.

Platform A specific target/compiler combination. For the configuration of platforms, refer to the Code generation target settings in the TargetLink Main Dialog Block block.

Pool area A DD object which is parented by the DD root object. It contains all data objects which can be referenced in TargetLink models and which are used for code generation. Pool data objects allow common data specifications to be reused across different blocks or models to easily keep consistency of common properties.

Port (AUTOSAR) A part of a software component (SWC) that is the interaction point between the component and other software components.

Port-defined argument values Argument values the RTE can implicitly pass to a server.

Preferences Editor A TargetLink tool that lets users view and modify all user-specific preference settings after installation has finished.

Production code The code generated from a code generation unit (CGU) that owns the module containing the code. See also stub code.

Project folder A folder in the file system that belongs to a TargetLink code generation project. It forms the root of different artifact locations that belong to this project.

Property Manager The TargetLink user interface for conveniently managing the properties of multiple model elements at the same time. It can consist of menus, context menus, and one or more panes for displaying property–related information.

Provide calprm port A provide port in parameter communication as described by Classic AUTOSAR. In the Data Dictionary, provide calprm ports are represented as DD ProvideCalPrmPort objects.

R

Read/write access function An access function (AF) that *encapsulates the instructions* for reading or writing a variable.

Real-time application An application that can be executed in real time on dSPACE real-time hardware such as SCALEXIO.

Receiver port A require port in sender-receiver communication as described by Classic AUTOSAR. In the Data Dictionary, receiver ports are represented as DD ReceiverPort objects.

ReplaceableDataItem (RDI) A ReplaceableDataItem (RDI) object is a DD object that describes an abstract interface's basic properties such as the data type, scaling and width. It can be referenced in TargetLink block dialogs and is generated as a global macro during code generation. The definition of the RDI macro can then be generated later, allowing flexible mapping to an implementation variable.

Require calprm port A require port in parameter communication as described by Classic AUTOSAR. In the Data Dictionary, require calprm ports are represented as DD RequireCalPrmPort objects.

RequirementInfo An object of a DD RequirementInfo object. It describes an item of requirement information and has the following properties: Description, Document, Location, UserTag, ReferencedInCode, SimulinkStateflowPath.

Restart function A production code function that initializes the global variables that have an entry in the RestartfunctionName field of their variable class.

Reusable function definition The function definition that is to be reused in the generated code. It is the code counterpart to the reusable system definition in the model.

Reusable function instance An instance of a reusable function definition. It is the code counterpart to the reusable system instance in the model.

Reusable model part Part of the model that can become a reusable system definition. Refer to Basics on Function Reuse (TargetLink Customization and Optimization Guide).

Reusable system definition A model part to which the function reuse is applied.

Reusable system instance An instance of a reusable system definition.

Root bus A root bus is a bus that is not a subordinate part of another bus.

Root function A function that represents the starting point of the TargetLink-generated code. It is called from the environment in which the TargetLink-generated code is embedded.

Root model The topmost parent model in the system hierarchy.

Root module The module that contains all the code elements that belong to the production code of a code generation unit (CGU) and do not have their own module specification.

Root step function A step function that is called only from outside the production code. It can also represent a non-TargetLink subsystem within a TargetLink subsystem.

Root struct A root struct is a struct that is not a subordinate part of another struct.

Root style sheet A root style sheet is used to organize several style sheets defining code formatting.

RTE event The abbreviation of run-time environment event.

Runnable A part of an atomic SWC. With regard to code execution, a runnable is the smallest unit that can be scheduled and executed. Each runnable is implemented by one C function.

Runnable execution constraint Constraints that specify runnables that are allowed or not allowed to be started or stopped before a runnable.

Runnable subsystem An atomic subsystem that contains a Function block whose AUTOSAR mode property is set to `Classic` and whose Role is set to `Runnable`.

Run-time environment (RTE) A generated software layer that connects the application layer to the basic software. It also interconnects the different SWCs of the application layer. There is one RTE per ECU.

Run-time environment event A part of an internal behavior. It defines the situations and conditions for starting or continuing the execution of a specific runnable.

S

Scaling A parameter that specifies the fixed-point range and resolution of a variable. It consists of the data type, least significant bit (LSB) and offset.

Sender port A provide port in sender-receiver communication as described by Classic AUTOSAR. In the Data Dictionary, sender ports are represented as DD SenderPort objects.

Sender-receiver interface An interface that describes the data elements and event messages that are provided or required by a software component (SWC) via a port (AUTOSAR).

Sender-receiver port A provide-require port in sender-receiver communication as described by Classic AUTOSAR. In the Data Dictionary, sender-receiver ports are represented as DD SenderReceiverPort objects.

Server port A provide port in client-server communication as described by Classic AUTOSAR. In the Data Dictionary, server ports are represented as DD ServerPort objects.

Server runnable A runnable that provides an operation via a server port. Server runnables are triggered by operation invoked events.

Shared parameter A parameter for measurement and calibration that is used by several instances of the same software component (SWC).

Shared variable A variable that is accessed by several reusable system instances. Typically, shared variables are used for parameters whose values are the same across instances. They increase code efficiency.

SIC runnable function A void (void) function that is called in a task. Generated into the Simulink implementation container (SIC) to call the root function that is generated by TargetLink from a TargetLink subsystem. In ConfigurationDesk, this function is called *runnable function*.

SIL simulation A simulation method in which the control algorithm's generated production code is computed on the host computer in place of the corresponding model.

Simple TargetLink model A simple TargetLink model contains at least one TargetLink Subsystem block and exactly one MIL Handler block.

Simplified initialization mode The initialization mode used when the Simulink diagnostics parameter Underspecified initialization detection is set to `Simplified`.

See also classic initialization mode.

Simulation application An application that represents a graphical model specification (implemented control algorithm) and simulates its behavior in an offline Simulink environment.

Simulation code Code that is required only for simulation purposes. Does not belong to the production code.

Simulation S-function An S-function that calls either the root step functions created for a TargetLink subsystem, or a user-specified step function (only possible in test mode via API).

Simulink data store Generic term for a memory region in MATLAB/Simulink that is defined by one of the following:

- A Simulink.Signal object
- A Simulink Data Store Memory block

Simulink function call The location in the model where a Simulink function is called. This can be:

- A Function Caller block
- The action language of a Stateflow Chart
- The MATLAB code of a MATLAB function

Simulink function definition The location in the model where a Simulink function is defined. This can be one of the following:

- Simulink Function subsystem
- Exported Stateflow graphical function
- Exported Stateflow truthtable function
- Exported Stateflow MATLAB function

Simulink function ports The ports that can be used in a Simulink Function subsystem. These can be the following:

- TargetLink ArgIn and ArgOut blocks

  These ports are specific for each Simulink function call.
- TargetLink InPort/OutPort and Bus Inport/Bus Outport blocks

  These ports are the same for all Simulink function calls.

Simulink Function subsystem A subsystem that contains a Trigger block whose Trigger Type is `function-call` and whose Treat as Simulink Function checkbox is selected.

Simulink implementation container (SIC) A file that contains all the files required to import production code generated by TargetLink into ConfigurationDesk as a behavior model with model ports.

Slice A section of a vector or matrix signal, whose elements have the same properties. If all the elements of the vector/matrix have the same properties, the whole vector/matrix forms a slice.

Software component (SWC) The generic term for atomic software component (atomic SWC), compositions, and special software components, such as calprm software components. A software component logically groups and encapsulates single functionalities. Software components communicate with each other via ports.

Software component description (SWC-D) An XML file that describes software components according to AUTOSAR.

Stateflow action language The formal language used to describe transition actions in Stateflow.

Struct A struct (short form for structure) consists of subordinate struct components. A struct component can be a struct itself.

Struct component A struct component is a part of a struct and can be a struct itself.

Structure A structure (long form for struct) consists of subordinate struct components. A struct component can be a struct itself.

Stub code Code that is required to build the simulation application but that belongs to another code generation unit (CGU) than the one used to generate production code.

Subsystem area A DD object which is parented by the DD root object. This object consists of an arbitrary number of Subsystem objects, each of which is the result of code generation for a specific code generation unit (CGU). The Subsystem objects contain detailed information on the generated code, including C modules, functions, etc. The data in this area is either automatically generated or imported from ASAM MCD-2 MC, and must not be modified manually.

Supported Simulink block A TargetLink-compliant block from the Simulink library that can be directly used in the model/subsystem for which the Code Generator generates production code.

SWC container A container for files of one SWC.

Synchronous operation call subsystem A subsystem used when modeling *synchronous* client-server communication. It is used to generate the call of the `Rte_Call` API function and for simulation purposes.

T

Table function A function that returns table output values calculated from the table inputs.

Target config file An XML file named `TargetConfig.xml`. It contains information on the basic data types of the target/compiler combination such as the byte order, alignment, etc.

Target Optimization Module (TOM) A TargetLink software module for optimizing production code generation for a specific microcontroller/compiler combination.

Target Simulation Module (TSM) A TargetLink software module that provides support for a number of evaluation board/compiler combinations. It is used to test the generated code on a target processor. The TSM is licensed separately.

TargetLink AUTOSAR Migration Tool A software tool that converts classic, non-AUTOSAR TargetLink models to AUTOSAR models at a click.

TargetLink AUTOSAR Module A TargetLink software module that provides extensive support for modeling, simulating, and generating code for AUTOSAR software components.

TargetLink Base Suite The base component of the TargetLink software including the ANSI C Code Generator and the Data Dictionary Manager.

TargetLink base type One of the types used by TargetLink instead of pure C types in the generated code and the delivered libraries. This makes the code platform independent.

TargetLink Blockset A set of blocks in TargetLink that allow production code to be generated from a model in MATLAB/Simulink.

TargetLink Data Dictionary The central data container thats holds all relevant information about an ECU application, for example, for code generation.

TargetLink simulation block A block that processes signals during simulation. In most cases, it is a block from standard Simulink libraries but carries additional information required for production code generation.

TargetLink subsystem A subsystem from the TargetLink block library that defines a section of the Simulink model for which code must be generated by TargetLink.

Task A code section whose execution is managed by the real-time operating system. Tasks can be triggered periodically or based on events. Each task can call one or more SIC runnable functions.

Task function A function that implements a task and calls the functions of the subsystems which are assigned to the task by the user or via the TargetLink Code Generator during multirate code generation.

Term function A function that contains the code to be executed when the simulation finishes or the ECU application terminates.

Terminate function A runnable that finalizes a SWC, for example, by calling code that has to run before the application shuts down.

Timing event An RTE event that specifies to start or continue the execution of a runnable at constant time intervals.

tllib A TargetLink block library that is the source for creating TargetLink models graphically. Refer to How to Open the TargetLink Block Library (TargetLink Orientation and Overview Guide).

Transformer The Classic AUTOSAR entity used to perform a data transformation.

TransformerError The parameter passed by the run-time environment (RTE) if an error occurred in a data transformation. The `Std_TransformerError` is a struct whose components are the transformer class and the error code. If the error is a hard error, a special runnable is triggered via the TransformerHardErrorEvent to react to the error.

In AUTOSAR releases prior to R19-11 this struct was named `Rte_TransformerError`.

TransformerHardErrorEvent The RTE event that triggers the runnable to be used for responding to a hard TransformerError in a data transformation for client-server communication.

Type prefix A string written in front of the variable type of a variable definition/declaration, such as `MyTypePrefix Int16 MyVar`.

U

Unicode The most common standard for extended character sets is the Unicode standard. There are different schemes to encode Unicode in byte format, e.g., UTF-8 or UTF-16. All of these encodings support all Unicode characters. Scheme conversion is possible without losses. The only difference between these encoding schemes is the memory that is required to represent Unicode characters.

User data type (UDT) A data type defined by the user. It is placed in the Data Dictionary and can have associated constraints.

Utility blocks One of the categories of TargetLink blocks. The blocks in the category keep TargetLink-specific data, provide user interfaces, and control the simulation mode and code generation.

V

Validation Summary Shows unresolved model element data validation errors from all model element variables of the Property View. It lets you search, filter, and group validation errors.

Value copy AF An access function (AF) resulting from DD AccessFunction objects whose AccessFunctionKind property is set to `READ_VALUE_COPY` or `WRITE_VALUE_COPY`.

Variable access function An access function (AF) that *encapsulates the access* to a variable for reading or writing.

Variable class A set of properties that define the role and appearance of a variable in the generated production code, e.g. CAL for global calibratable variables.

VariantConfig A DD object in the Config area that defines the code variants and data variants to be used for simulation and code generation.

VariantItem A DD object in the DD Config area used to variant individual properties of DD Variable and ExchangeableWidth objects. Each variant of a property is associated with one variant item.

V-ECU implementation container (VECU) A file that consists of all the files required to build an offline simulation application (OSA) to use for simulation with VEOS.

V-ECU Manager A component of TargetLink that allows you to configure and generate a V-ECU implementation.

Vendor mode The operation mode of RTE generators that allows the generation of RTE code which contains vendor-specific adaptations, e.g., to reduce resource consumption. To be linkable to an RTE, the object code of an SWC must have been compiled against an application header that matches the RTE code generated by the specific RTE generator. This is the case because the data structures and types can be implementation-specific.

See also compatibility mode.

VEOS A dSPACE software platform for the C-code-based simulation of virtual ECUs and environment models on a PC.

Virtual ECU (V-ECU) Software that emulates a real ECU in a simulation scenario. The virtual ECU comprises components from the application and the basic software, and provides functionalities comparable to those of a real ECU.

Virtual ECU testing Offline and real-time simulation using virtual ECUs.

Virtual evaluation board (virtual EVB) A combination of an instruction set simulator (ISS) and a simulated periphery. This combination can be used for validation of generated production code in PIL simulation mode.

W

Worst-case range limits A range specified by calculating the minimum and maximum values which a block's output or state variable can take on with respect to the range of the inputs or the user-specified constrained range limits.