TargetLink Reference Guide

Version: Initial

Table of Contents

[1. Purpose 4](#_Toc467491453)

[2. Scope 4](#_Toc467491454)

[3. Acronyms 4](#_Toc467491455)

[4. Some Important Terms 4](#_Toc467491456)

[5. TargetLink Code Generation 5](#_Toc467491457)

[5.1. Pre-requisites: 5](#_Toc467491458)

[5.2. Converting a Simulink system to TargetLink system 5](#_Toc467491459)

[5.2.1. Steps in converting the Simulink system to TargetLink system 5](#_Toc467491460)

[5.3. Data Dictionary 8](#_Toc467491461)

[5.3.1. DD Creation Methods 8](#_Toc467491462)

[5.3.2. Saving the DDs 10](#_Toc467491463)

[5.4. Specifying TargetLink properties for all blocks 10](#_Toc467491464)

[5.4.1. Syncing Simulink Properties to TargetLink 10](#_Toc467491465)

[5.4.2. Modifying Block Properties via Block Dialogs 10](#_Toc467491466)

[5.4.3. Modifying Properties via the TargetLink Property Manager 12](#_Toc467491467)

[5.5. Creation of frame model for the component 13](#_Toc467491468)

[5.5.1. Integrating Component level library to frame model 13](#_Toc467491469)

[5.6. Incremental Code Generation (ICG) 13](#_Toc467491470)

[5.7. Module Ownership 14](#_Toc467491471)

[5.8. Code Generation Basis 14](#_Toc467491472)

[5.9. Best Practices 15](#_Toc467491473)

[5.9.1. Function Handling 15](#_Toc467491474)

[5.9.1.1. Why/When a function is needed? 15](#_Toc467491475)

[5.9.1.2. TargetLink function block properties 15](#_Toc467491476)

[5.9.1.3. Stateflow functions and properties 16](#_Toc467491477)

[5.9.1.4. How to optimize functions 19](#_Toc467491478)

[5.9.1.5. Handling of Function Arguments vs. Global Variables 20](#_Toc467491479)

[5.9.1.6. Reuse of Functions 20](#_Toc467491480)

[5.9.2. Variable Handling 21](#_Toc467491481)

[5.9.2.1. Naming conventions for DD variables 21](#_Toc467491482)

[5.9.2.2. How to optimize variables 21](#_Toc467491483)

[5.9.2.3. Elimination of Temporary Variables related to bus objects 22](#_Toc467491484)

[5.9.2.4. Dealing with Measurement Variables 23](#_Toc467491485)

[5.9.2.5. Optimized Vector Processing 24](#_Toc467491486)

[5.9.3. Others 24](#_Toc467491487)

[5.9.3.1. Dealing with NaN values 24](#_Toc467491488)

[5.9.3.2. Usage of custom code blocks 25](#_Toc467491489)

[5.9.3.3. Protection against division by zero 25](#_Toc467491490)

[5.9.3.4. Structure Types 26](#_Toc467491491)

[5.9.3.5. Bus Assignment Block 26](#_Toc467491492)

[5.9.3.6. Naming conventions for blocks 27](#_Toc467491493)

[5.9.3.7. 64 Bit Operations 27](#_Toc467491494)

[5.9.3.8. Consistency of Scaling Parameters 27](#_Toc467491495)

[5.9.3.9. Avoidance of Saturation Specification 28](#_Toc467491496)

[5.9.3.10. Usage of Enabled subsystems 28](#_Toc467491497)

[5.9.3.11. TL properties for Charts, Triggered/ Enabled subsystems/ If\_action subsystems/ Atomic subsystems 29](#_Toc467491498)

[6. Reference Documents 29](#_Toc467491499)

# Purpose

The purpose of this document is to define the workflow and guidelines for TargetLink that shall serve as a reference guide for a user to establish the understanding of code generation process using TargetLink.

# Scope

The document covers the TargetLink (Version 3.4) code generation process and guidelines to be followed for Model based Projects of WABCO.

# Acronyms

MIL – Model In Loop

SIL – Software In Loop

# Some Important Terms

Simulink:

Simulink, developed by [MathWorks](http://en.wikipedia.org/wiki/MathWorks), is a [data flow](http://en.wikipedia.org/wiki/Data_flow) graphical programming language tool for modeling, simulating and analyzing multidomain [dynamic systems](http://en.wikipedia.org/wiki/Dynamic_systems)

Stateflow:

Stateflow, developed by [MathWorks](http://en.wikipedia.org/wiki/MathWorks), is a control logic tool used to model [reactive systems](http://en.wikipedia.org/wiki/Reactive_systems) via state machines and [flow charts](http://en.wikipedia.org/wiki/Flow_charts) within a [Simulink](http://en.wikipedia.org/wiki/Simulink) model

TargetLink:   
dSPACE's production code generation software. TargetLink is a software for automatic code generation, based on a subset of [Simulink](http://en.wikipedia.org/wiki/Simulink)/[Stateflow](http://en.wikipedia.org/wiki/Stateflow) models, produced by [dSPACE GmbH](http://en.wikipedia.org/wiki/DSPACE_GmbH).  
Data Dictionary (DD):   
A central data container that holds all the relevant information for code generation, keeps data consistent throughout all stages of the development process, and facilitates multi-model applications

# TargetLink Code Generation

# Pre-requisites:

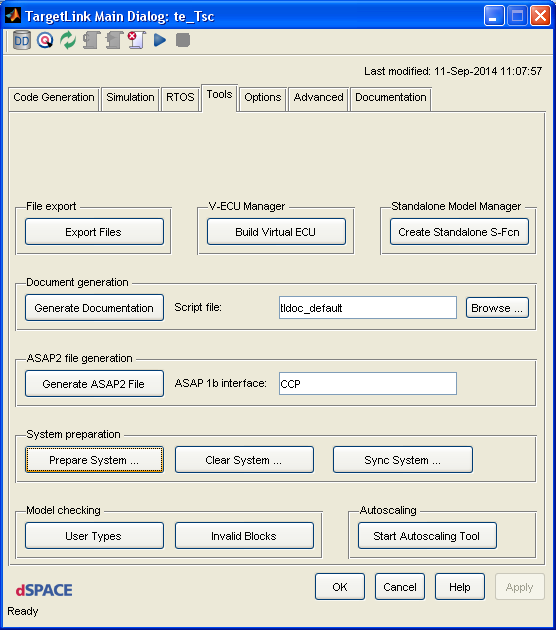
* Functional requirements of the module should have been implemented using MATLAB SIMULINK/STATEFLOW
* Test environment of the module should be compilable
* MIL testing should have been performed to ensure correctness of model as per requirements

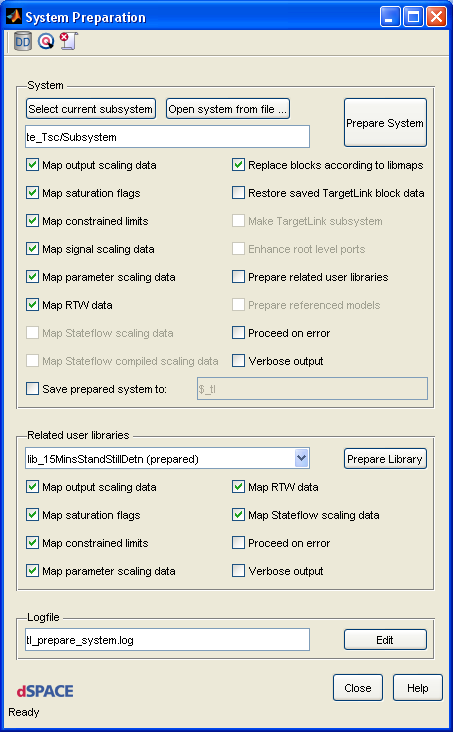
# Converting a Simulink system to TargetLink system

* Preparing a Simulink system (block, subsystem, library, or model) for TargetLink means that TargetLink can generate production code for it, but it is still fully compatible with Simulink.
* TargetLink can enhance Simulink blocks by adding TargetLink production code options to them, hence the Simulink blocks become TargetLink blocks which TargetLink can generate production code from. This can be done with single Simulink blocks as well as complete subsystems, user libraries and referenced models.
* TargetLink blocks are nothing but Simulink blocks enhanced with TargetLink production code options, they hold both Simulink-specific and TargetLink-specific block properties.

# Steps in converting the Simulink system to TargetLink system

* Place the ‘TargetLink Main Dialogue’ block from the TargetLink block set in the test environment which contains the subsystem to be prepared  
  
* Also place a MIL Handler block that performs the overflow checks though it has nothing directly to do with the conversion
* Double click and Open the TargetLink Main Dialogue block
* Select the Tools page



* Select the ‘Prepare System’ option
* A ‘System Preparation’ window as shown below is opened. Select the subsystem for code generation   
  
* Select the ‘unprepared’ user libraries one by one from the ‘Related user libraries’ list and prepare them using the ‘Prepare Library’ option.
* If a library has further sub-libraries, prepare the sub-libraries first.
* After preparing all the related user libraries of the subsystem successfully, prepare the entire subsystem using the ‘Prepare System’ option.
* The subsystem will be prepared successfully if it contains all enhanceable and supported Simulink blocks
* Points to be noted:
  + As a general rule, the blocks from WABCO Simulink Blockset shall only be used in the models
  + It does not matter if a system already contains TargetLink blocks. System preparation leaves them untouched.
  + After system preparation, the input and output ports of the internal subsystems will still be Simulink ports. Only the ports below selected subsystem are enhanced to TargetLink. Other subsystems with Simulink In and Out ports represent just a graphical hierarchy which has no influence on the code generation.

# Data Dictionary

Every module shall have a data dictionary (DD) that contains all Input/ Output variables, Datatypes and Scaling definitions for the module. For basic information on DD and different sections inside a DD refer the document [ChildDD\_MasterDD.docx](http://170.205.207.224:7001/si/viewrevision?projectName=d:/MKS_proj/T3_public/Tools/MbE/Matlab_Simulink/Misc/GuidelineDocuments/project.pj&revision=:member&selection=TargetLink_ReferenceDocs/ChildDD_MasterDD.docx)

# DD Creation Methods

Method 1: From RM

* The DD and frame model for a function can be created from RM having all the input and output interfaces specified in a defined format.
* Refer the Document [Workflow\_RqM2DD](http://170.205.207.224:7001/si/viewrevision?projectName=d:/MKS_proj/T3_public/Tools/MbE/Matlab_Simulink/Misc/GuidelineDocuments/project.pj&revision=:member&selection=TargetLink_ReferenceDocs/Workflow_RqM2DD.docx).docx

Method 2: From Simulink Model

NOTE:

Not officially supported by TEC team. This is a work around for scenarios where specification documents for models don’t exist

Step 1: Model2Xls

* Run the startup\_modulename.m script (In order to include the T\_Tools path to Matlab)
* Run the script model2xls.m from MATLAB command window
* The script will prompt to choose the subsystem. Choose the module library in the test environment. Similarly DD can be created for any subsystem in the model by selecting it.
* The script will search the selected subsystem and list the variables with constants blocks.
* Choose the variables for which the DD entries need to be created
* An interface xls file containing the list of input and output interfaces along with the interface properties which need to be edited will be generated

Output:

* Interface xls with all in ports, out ports and selected constant variables

Step 2: Xls2DD

Procedure:

* Run the script xls2dd.m from MATLAB command window
* The script will prompt to choose the xls sheet. Choose the xls file created in step1
* The script asks if your project is an Autosar project or not. It will select the MasterDD template based on your input
* The script also asks if a library or model shall be generated

Output:

* Child DD for variables and typedefs
* New library or model(based on the selection) containing all TargetLink ports
* Rename the child DD to Module name and Check in inside the module level lib folder
* In case of buses, an M file containing all bus object information is generated.

This shall be called during the initialization of modules or components

Remarks:

* When running the Xls2DD script, at first run select the output format as ‘Library’.

In case an existing frame has to be updated, choosing the output format as ‘Model’ would be helpful. It is then possible to copy ports from that generated ‘Model’ into the existing library without the risk of invalid library links.

* In case of bus ports, an IMPLICIT\_STRUCT will be created for every bus in- and out port. Structure types are not reused.
* The generated M file for bus objects contains information of all Simulink bus objects available in the base workspace, instead of bus objects that are needed only for the selected subsystem
* Also refer to the document below for more information on XLS2DD process

<http://170.205.207.224:7001/si/viewrevision?projectName=d:/MKS_proj/T3_public/Tools/MbE/Matlab_Simulink/Misc/XLS2DD/project.pj&revision=:member&selection=Workflow_RM_IfSpec.docx>

# Saving the DDs

* All Child DDs shall be saved separately
* A component level DD shall be saved after removing all the child DDs

# Specifying TargetLink properties for all blocks

When a Simulink block has been enhanced or taken from the TargetLink library, its TargetLink production code options are set to the TargetLink defaults. However, some of the TargetLink production code options correspond to Simulink properties.

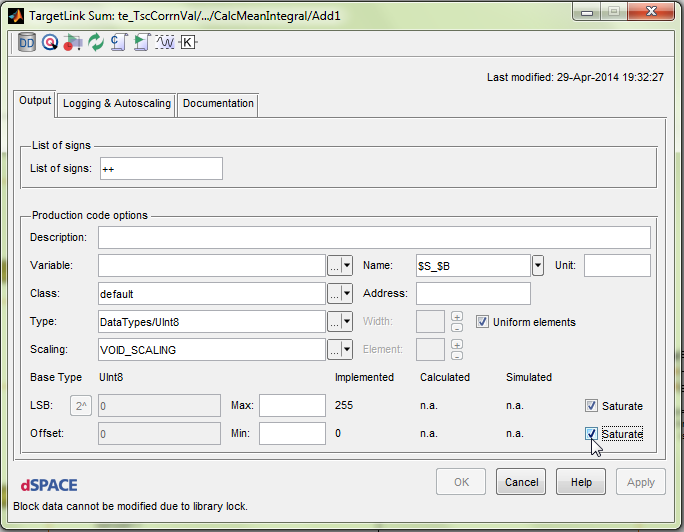
The TargetLink properties of all blocks shall be specified appropriately in order to avoid any underflow, overflow or loss of data.

# Syncing Simulink Properties to TargetLink

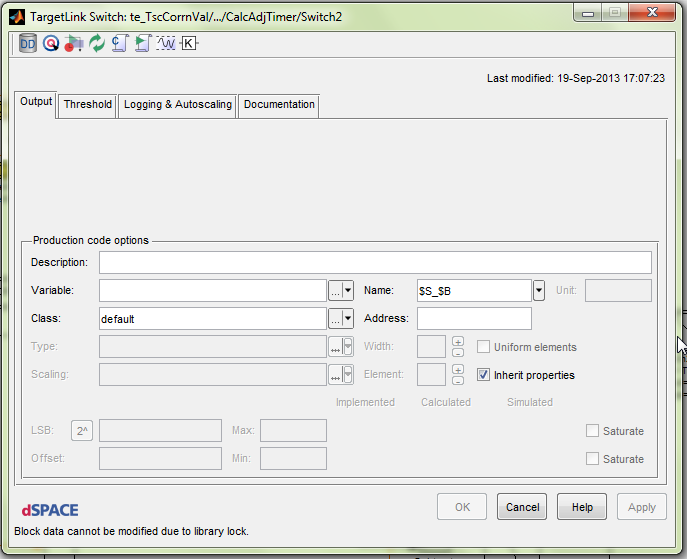
* TargetLink allows us to synchronize the values of TargetLink production code options with the values of their corresponding Simulink properties, and vice versa, automatically.
* The Sync System button on the Tools page of the TargetLink Main Dialog block(Or tl\_sync\_system in the MATLAB Command Window), lets you access the ‘System Synchronization’ dialog through which the TargetLink property values matching to the values of corresponding Simulink properties can be set, and vice versa
* Please refer to the document [Sync\_System.docx](http://170.205.207.224:7001/si/viewrevision?projectName=d:/MKS_proj/T3_public/Tools/MbE/Matlab_Simulink/Misc/GuidelineDocuments/project.pj&revision=:member&selection=TargetLink_ReferenceDocs/Sync_System.docx)
* For more Information on ‘Synchronization of Simulink and TargetLink Data’, please refer TLProductionCodeGeneration Guide.pdf 🡪 Preparing Simulink Systems for TargetLink 🡪 Synchronization of Simulink and TargetLink Data. (Page No: 109-115)

# Modifying Block Properties via Block Dialogs

* Modifying TargetLink block properties via block dialogs is convenient if you want to modify properties of individual TargetLink blocks.
* When a TargetLink block is double clicked, the corresponding block dialog opens as below.



* In the block dialog, all suitable TargetLink settings shall be specified
* The Min/Max entries let you specify constraint range limits for an output variable where you know that the limits will never be exceeded. Refer to [Min\_Max\_Saturation.pdf](http://170.205.207.224:7001/si/viewrevision?projectName=d:/MKS_proj/T3_public/Tools/MbE/Matlab_Simulink/Misc/GuidelineDocuments/project.pj&revision=:member&selection=TargetLink_ReferenceDocs/Min_Max_Saturation.pdf)
* Reference: <https://www.dspace.com/en/pub/home/support/kb/faq/supfaqtl.cfm>
* In case of fixed point operations, appropriate ‘Scaling’ shall be specified for all TL blocks
* The TL properties can be inherited for certain blocks like Switch, Unit delay. In those cases the ‘Inherit properties’ option shall be checked



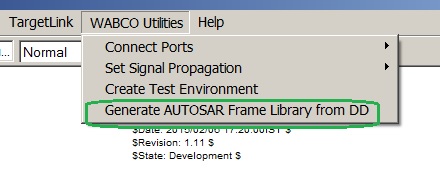
* It is possible to generate code without enhancing the datatype conversion blocks that are used inside a model. But in some cases it would result in code generation error. So it is always recommended to enhance all datatype conversion blocks
  + Refer to lesson LL0104 of Lessons\_Learnt.xls for more information  
    <http://wabco-pe/Method_Tools/targetlink/MbSWDDocuments/LessonsLearned/Lessons_Learnt.xls>
* All the constant blocks shall be mapped to DD variables to avoid any unwanted code optimization by TargetLink, that may lead to undesired behavior
  + Refer to lesson LL0090 of Lessons\_Learnt.xls

# Modifying Properties via the TargetLink Property Manager

* Changing the properties of large models via block dialogs manually is time-consuming and error-prone. TargetLink provides the Property Manager, a graphical user interface that allows you to view, filter and modify several properties of TargetLink blocks and Stateflow objects in a model simultaneously
* Refer to the section on ‘Modifying Properties via the TargetLink Property Manager’ in TargetLink Production Code Generation Guide

# Creation of frame model for the component

* Script to automate the frame model generation is created and available under ‘WABCO Utilities’



* Detailed steps on creation of frame model can be found at the below link

<http://170.205.207.224:7001/si/viewrevision?projectName=d:/MKS_proj/MBSP_SW_Components/D12_Tooling/Documents/project.pj&revision=:member&selection=AUTOSAR_FrameModelGeneration_HowTo.ppt>

# Integrating Component level library to frame model

* + The Component library shall be placed within the frame model (at the Runnable layer) and the inputs & outputs shall be connected to corresponding frame model ports
  + The component library need not be an Atomic subsystem (no function block needed)
  + The inports and outports within the component library need not be enhanced to TargetLink ports
* NOTE:

The current frame model generation script is updated such that the integration of component library into the frame model is automatically taken care. Issues if any to be notified

# Incremental Code Generation (ICG)

* TargetLink allows you to generate code for nested subsystems, in contrast to code generation for a root level TargetLink subsystem. This allows more tailored code generation, that is, one can create smaller code fragments that can be tested separately and thus may reduce development time. After a modification, only the modified model part must be regenerated, which also facilitates the review and testing process.
* To be able to generate code incrementally, the subsystems must be configured for incremental code generation and must contain a TargetLink Function block with the ‘Enable Incremental Code Generation’ checkbox selected on the Incremental page

# Code Generation @SWC level & Code Generation @ICG module level

During Code generation at component level, the C files of ICG modules are resynchronized (not regenerated) from module folders. This is acceptable unless the function definition of .c file (after resynchronization) from ICG module folder is not compatible with the .mdl file of ICG module that is referenced by component level test environment.

Developers may think a change in .mdl file of ICG module requires code generation at its module level alone. But this doesn’t hold true always. In some cases, in addition to code generation at module level, we need to generate code at SWC level even though the change involves only in module level .mdl files.

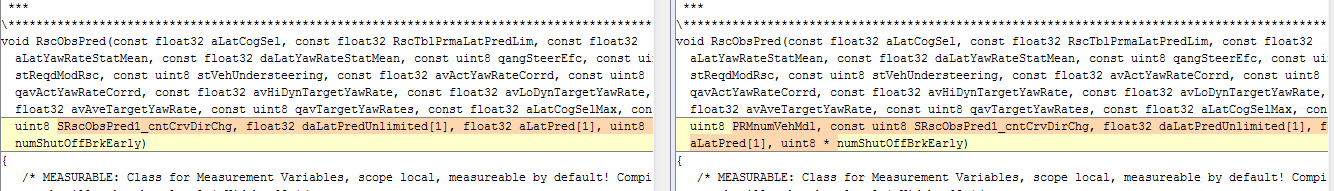
A typical case might be,

1) Change in ICG module interface like datatype, interface order etc. Note: For this case, the developer eventually needs to edit SWC .mdl file in order to adapt the interface change.

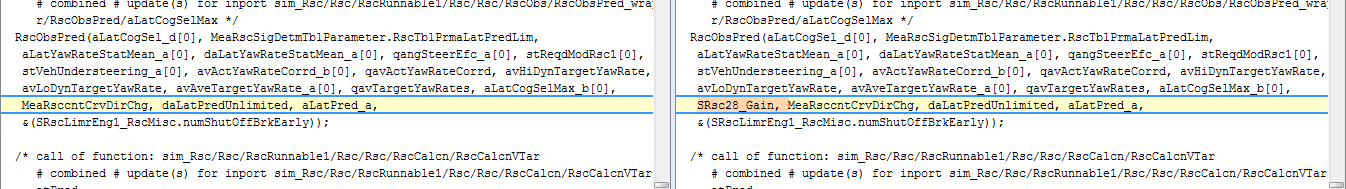
2) Some TL properties of blocks that affect function definition (mainly arguments) in .c file of ICG module, though there are no specific interface changes in .mdl file of ICG module. One such TL property is name template of ports set to $S\_$L. This is a potential case which may result in code generation at SWC level not possible, though code generation is possible at module level.

A general suggestion would be, even if you have changes specifically within an ICG module, in addition to code generation at module level, it is always safe to generate code at SWC level using fcn\_generatecode() and verify the code differences before giving CP.

Code difference at Module level:



Code difference at SWC level:



For more information on ICG, refer to [CodegenerationWithTargetLink.docx](http://170.205.207.224:7001/si/viewrevision?projectName=d:/MKS_proj/T3_public/Tools/MbE/Matlab_Simulink/Documents/SoftwareDevelopmentWithTargetLink/project.pj&revision=:member&selection=CodegenerationWithTargetLink.docx)

# Module Ownership

* If a model comprises several code generation units, for example, TargetLink subsystems, it is not sufficient for a module to have a general owner (defined by the module's Responsibility property in the DD). When code is generated for multiple code generation units, the same number of production code files is generated, each of which is stored in a CGU-specific production code folder. However, these files may differ, which leads to problems during code compilation and building the application. You should therefore refine the general owner by defining a specific owner, i.e., a certain code generation unit
* A specific owner can be defined for a module by using a module owner object in the DD. The module must exist as a DD module object
* Defining the owner of a module forces the TargetLink Code Generator to generate a module as a part of the production code only if code is generated for the code generation unit that owns the module.
* The module ownership concept can be used in cases where calibration variables, type definitions and reusable functions are used across different code generation units
* For more information on how to use the concept of module ownership, refer to
  + ‘Module Ownership’ section of document [CodegenerationWithTargetLink.docx](http://170.205.207.224:7001/si/viewrevision?projectName=d:/MKS_proj/T3_public/Tools/MbE/Matlab_Simulink/Documents/SoftwareDevelopmentWithTargetLink/project.pj&revision=:member&selection=CodegenerationWithTargetLink.docx)
  + ‘Introduction to Module Ownership’ section of [TLAdvancedGuide.pdf](http://170.205.207.224:7001/si/viewrevision?projectName=d:/MKS_proj/T3_public/Tools/MbE/Matlab_Simulink/Misc/GuidelineDocuments/project.pj&revision=:member&selection=TargetLink_ReferenceDocs/TLAdvancedGuide.pdf)

# Code Generation Basis

* TargetLink provides the following CodeGenerationBasis options for a module
  + ModelBased
  + ModelAndDDBased
  + DDBased
* For more information on the differences between these code generation options, refer to the document [CodeGenerationBasis.docx](http://170.205.207.224:7001/si/viewrevision?projectName=d:/MKS_proj/T3_public/Tools/MbE/Matlab_Simulink/Misc/GuidelineDocuments/project.pj&revision=:member&selection=TargetLink_ReferenceDocs/CodeGenerationBasis.docx)

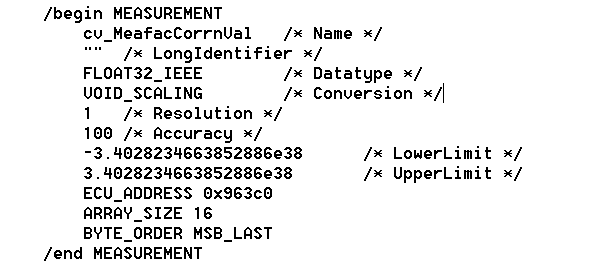
# Generation of A2L and Diary files

A2L Files:

* A2L is a standard commonly used for describing ECU-internal data.

An A2L file contains the following data -

* + Measurement and Calibration(MC) variable names
  + MC Descriptions, Datatypes, Scaling, Min/Max limits…etc.



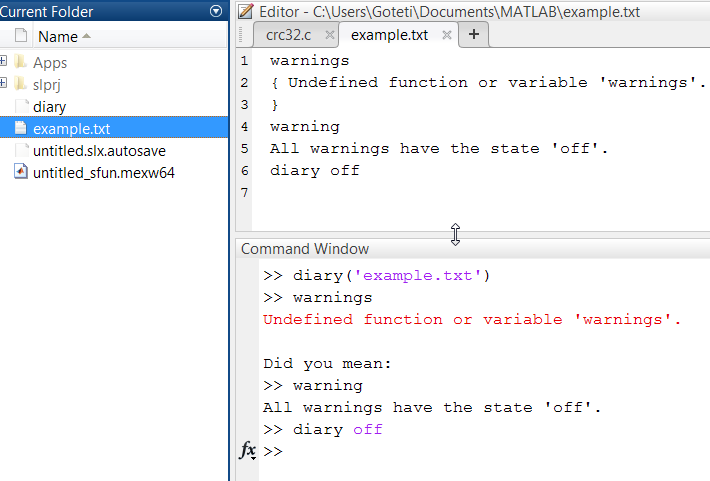
* During Production code generation for a component, such an a2l file containing all MC variable info. specific to the component gets generated (by calling fcn\_generateA2L.m inside fcn\_generateCode.m).

The user shall ensure that the generated a2l file is checked in and delivered along with the component CP.

* During SW integration, an overall project level a2l file can be created, taking information from all these component level a2l files. This a2l file would then be helpful in easy understanding of MC variables available in the ECU SW and their measurement and calibration.
* For more information on the A2L file, refer to the document [A2L\_FileGeneration.docx](http://170.205.207.224:7001/si/viewrevision?projectName=d:/MKS_proj/T3_public/Tools/MbE/Matlab_Simulink/Misc/GuidelineDocuments/project.pj&revision=:member&selection=TargetLink_ReferenceDocs/A2L_FileGeneration.docx)

Dairy Files:

* The Diary files get generated during production code generation and contain the content of the MATLAB command window during the code generation. These code generation logs are created by the Matlab command '*diary()’* [Called inside fcn\_generateCode.m]
* diary('filename.txt') enables writing the MATLAB command window into the text file, and ‘diary off’ stops that procedure. For eg., refer the screenshot below -



* The diary is stored as <model\_name>.txt and the main information of interest is the warnings that appear, which are also collected and written at the end of the diary. This way we have a proof for the absence of warnings, if there are none.
* Unlike A2L files, which shall be generated only on component level, Diary files shall be generated at all levels of code generation

# Best Practices

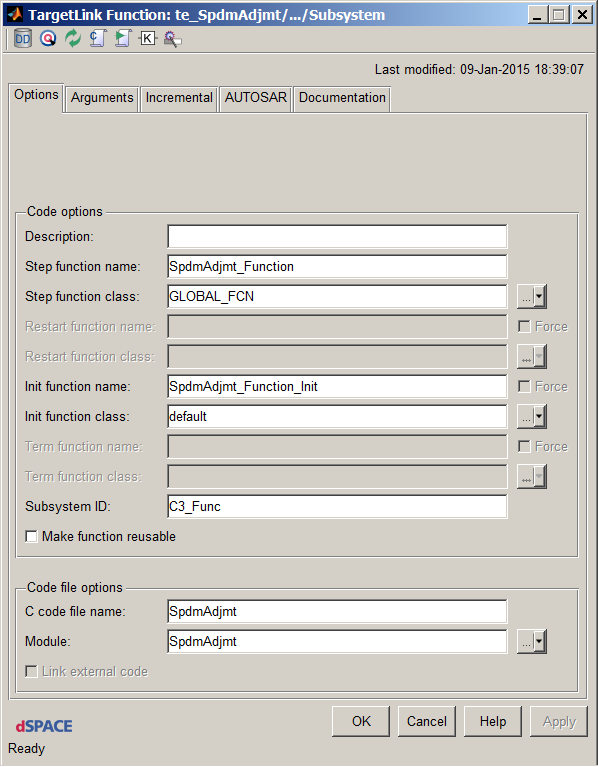
# Function Handling

# Why/When a function is needed?

* + - * + A component shall be broken down into modules and submodules based on functionality, maintainability and testability. Parts separated this way shall be made as libraries
        + Each library shall be placed as a function – A TargetLink function block shall be placed
        + Libraries which are very small and simple, and shall be tested within the module that is using it, can be used without function blocks, Example: EdgeDetection.
        + All re-used libraries shall be created as reusable functions – Make ‘Function Reusable’ shall be checked in the function block

# TargetLink function block properties

* + - * + On double clicking the function block, a dialog box will appear (as shown in below fig) where the function parameters have to be specified to create a reusable function in a separate C file.



* + - * + The Subsystem ID shall be configured to the MKS RM ID of the heading of the module

# Stateflow functions and properties

* + - * + Stateflow charts as reusable functions:

In order to make a Stateflow chart as a reusable function, go to TargetLink property manager of the model and navigate to the corresponding chart, right click the chart and then click “open this sf chart” option. You will now see a dialog box similar to   
Fig 5.2.1.1.2, where the function properties can be specified similar to Simulink.

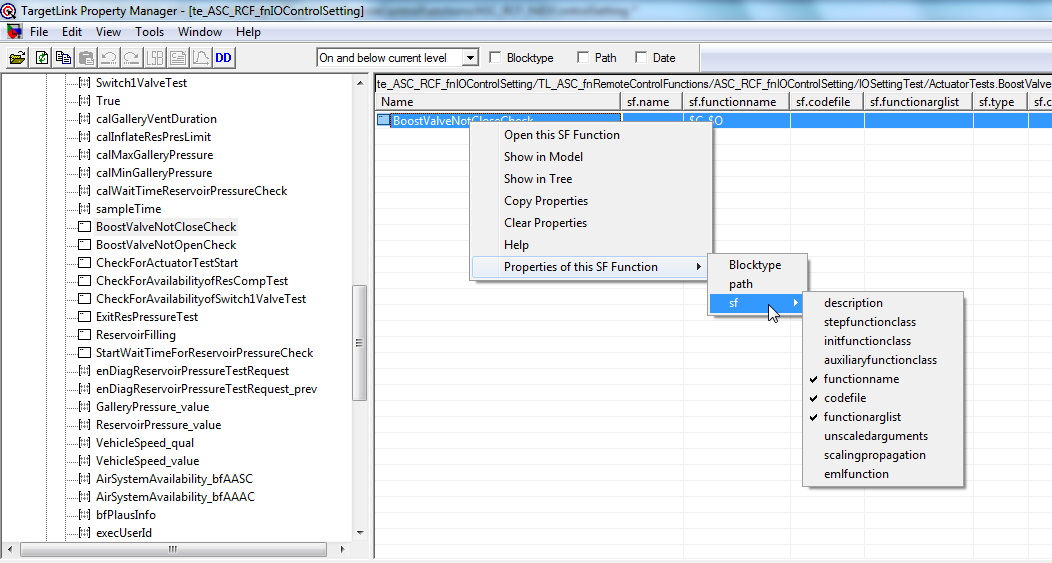
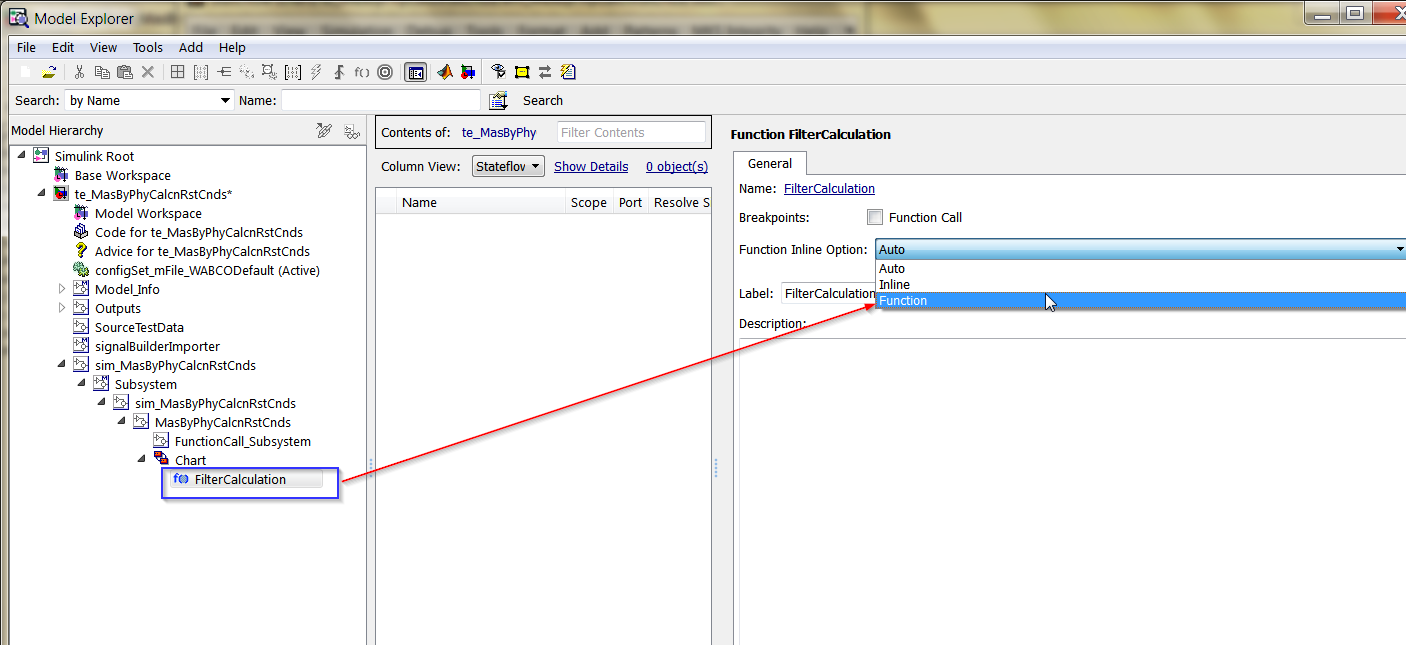


Fig 5.2.1.1.2

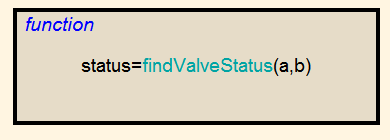
* + - * + Functions inside Stateflow chart as reusable:
* Any function inside Stateflow can be made reusable just like Simulink / Stateflow charts. Go to TL property manager 🡪 Navigate to your SF function 🡪 Cconfigure the required fields to create a reusable function
* It is also possible to create reusable C functions for the available Stateflow functions defined in a chart through Model explorer.

Open Model explorer 🡪 Navigate to your SF function 🡪 Select “Function” in Function Inline option



* Optimized code for SF functions:

Optimized code can be achieved by using SF functions with arguments.



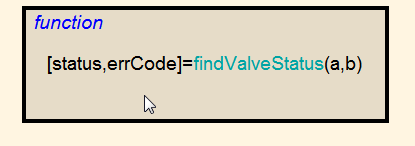
SF functions with no arguments shall be avoided. Otherwise static variables will get created in the generated code increasing the RAM consumption.



* Note - TargetLink Limitation (version 3.4P6 ):

Though SF function supports any number of input and output arguments, TargetLink is not able to generate code for SF functions whose Function Inline option is set as ‘Function’ and has more than one output.

In the below SF function, we have 2 outputs. To generate code, either we shall set Function Inline option as ‘Inline’ or avoid this kind of SF function design



# How to optimize functions

* NOTE:

This section gives basic information on the possibilities of function optimization through function classes. In general, one shall use generic WABCO code generation and optimization settings.

In case of project specific needs, the Technology team shall be involved.

* Function optimization can be achieved by changing the optimization properties of the respective function class



* Supported property values for the Optimization property of function classes are the following:
* SIDE\_EFFECT\_FREE
* MOVABLE
* CG\_AUTO\_INLINE
* CG\_FORCE\_INLINE
* CG\_NO\_INLINE
* The CompilerInline and Optimization properties for function classes in the Data Dictionary determine whether function calls should be replaced by the expansion of the function code
* Refer to section ‘Optimizing Functions via Property Values’ of [TLAdvancedGuide.pdf](http://170.205.207.224:7001/si/viewrevision?projectName=d:/MKS_proj/T3_public/Tools/MbE/Matlab_Simulink/Misc/GuidelineDocuments/project.pj&revision=:member&selection=TargetLink_ReferenceDocs/TLAdvancedGuide.pdf)

# Handling of Function Arguments vs. Global Variables

* The variables for TargetLink Inports and Outports are created as either function arguments or global variables, depending on the specified variable class.
* Depending on the choice of function parameters or global variables, either RAM or stack consumption can be reduced at the expense of the other resource. Passing values as function arguments usually increases the execution time.
* Since maintainability and risk of wrong connections is higher for global variables, usage of function arguments is preferable over global variables.
* Global Buses
  + When buses are passed across subsystems in a model, there is a possibility of unnecessary data copies getting created resulting in an inefficient code. Usage of global buses could resolve this issue. Though from design point of view this is not a good solution, the demands on efficient resource usage require the use of global buses.
  + Refer the document [SoftwareDevelopmentWithTargetLink\_GlobalBusses.ppt](http://170.205.207.224:7001/si/viewrevision?projectName=d:/MKS_proj/T3_public/Tools/MbE/Matlab_Simulink/Documents/SoftwareDevelopmentWithTargetLink/project.pj&revision=:member&selection=SoftwareDevelopmentWithTargetLink_GlobalBusses.ppt) on how global buses shall be used and the limitations in their use

# Reuse of Functions

* If a TargetLink model contains multiple identical subsystems/state charts, code for them should be generated in the form of one function only, which should be reused for each instance (Function Reuse)
* In case the function is used more than once, or at least intended to be used more than once, it has to be configured as “reusable function”. This can be done by checking the ‘Make function reusable’ option in the function block dialog.
* For more information on Function Reuse, refer to [CodegenerationWithTargetLink.docx](http://170.205.207.224:7001/si/viewrevision?projectName=d:/MKS_proj/T3_public/Tools/MbE/Matlab_Simulink/Documents/SoftwareDevelopmentWithTargetLink/project.pj&revision=:member&selection=CodegenerationWithTargetLink.docx)

# Variable Handling

# Naming conventions for DD variables

* The naming convention for variables shall be aligned with mBSP naming guidelines that will be the basis for generic WABCO conventions in future.

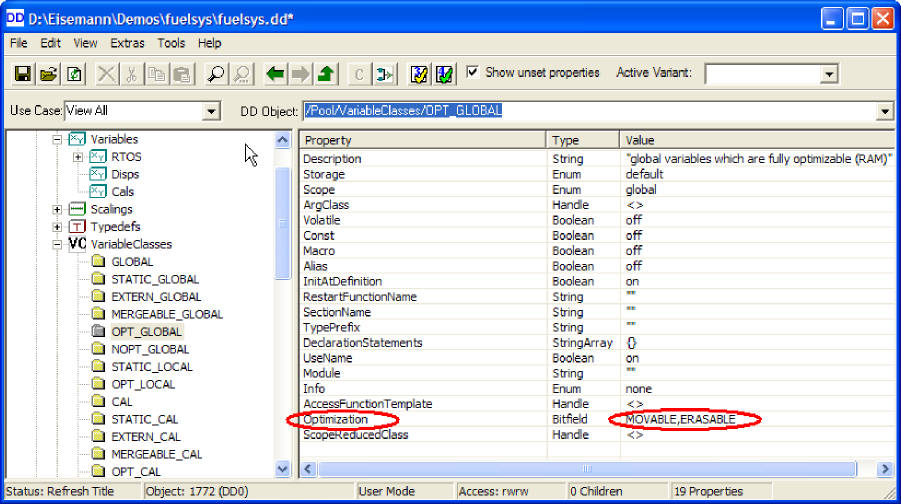
# How to optimize variables

* NOTE:

This section gives basic information on the possibilities of variable optimization through variable classes. In general, one shall use generic WABCO code generation and optimization settings.

In case of project specific needs, the Technology team shall be involved.

* TargetLink provides an efficient method for optimizing variables
* The variable classes stored in the Pool/VariableClasses folder in the TargetLink Data Dictionary contain an Optimization property which lets you specify property values (see below) for determining the optimization of variables.



* A variable class other than *default* should only be specified if this is really necessary, for instance to make a variable calibratable, measurable etc.
* Supported property values for the Optimization property of variable classes are the following:
  + MOVABLE
  + ERASABLE
  + SCOPE\_REDUCIBLE
  + MERGEABLE
* NOTE:

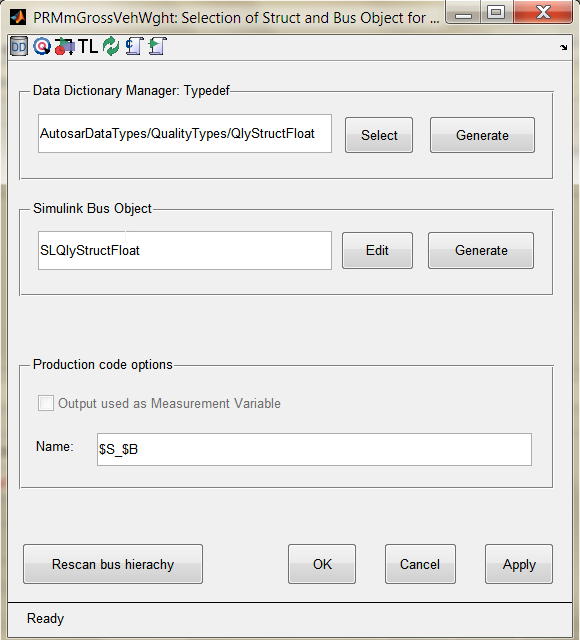
In general WABCO variable classes are defined in a separate Child-DD based on use cases. The developer just has to think of the use rather than the optimization.

The optimization is configured centrally by the Technology team, for example for Calibration variables.

# Elimination of Temporary Variables related to bus objects

* If a TL inport/ TL outport is a Bus port, then an appropriate BusObject should be assigned to that TL port.

Open TL GUI window of that TL BusInport/ BusOutport by clicking on it.



* Assign proper TypeDef and Simulink Bus Object (it’s automatically displayed if you have already assigned the busobject to Simulink port) and click Apply
* Click Rescan bus hierarchy so that the bus elements are updated correctly and then press OK
* With this approach, the creation of unnecessary Local and duplicate copy of structures is avoided
* Refer to the documents below for more information on bus objects  
  + [New Concept for Bus Ports and Module Interfaces.ppt](http://170.205.207.224:7001/si/viewrevision?projectName=d:/MKS_proj/MBSP_SW/D07_Design/Guidelines/TargetLink_CodeGeneration/project.pj&revision=:member&selection=New%20Concept%20for%20Bus%20Ports%20and%20Module%20Interfaces.ppt)
  + [SimulinkModelInterface\_modellingGuidelines.ppt](http://170.205.207.224:7001/si/viewrevision?projectName=d:/MKS_proj/MBSP_SW/D07_Design/Guidelines/SL_TL_handshake/project.pj&revision=:member&selection=SimulinkModelInterface_modellingGuidelines.ppt)

# Dealing with Measurement Variables

* For basic information on measuring function interface variables refer to [Optimization Concepts.doc](http://170.205.207.224:7001/si/viewrevision?projectName=d:/MKS_proj/T3_public/Tools/MbE/Matlab_Simulink/Documents/SoftwareDevelopmentWithTargetLink/dSPACE%20Workshop%20-%20Interface%20Concept/Modeling%20Hints/project.pj&revision=:member&selection=Optimization%20Concepts.doc)
* Below are the different WABCO specific variable classes provided as part of DD to support different scenarios of measurement variable configuration
  + FCN\_OUT\_MEASURABLE

Creates static variables for the arguments of called function with the scope limited to calling function

* + MEASURABLE

Creates function static variables with scope limited to the function

* + MEASURABLE\_GLOBAL

Creates global variables

* + FCN\_OUT\_MEASURABLE\_GLOBAL

Creates global variables for the arguments of called function.

This variable class in general is applied to the outports of an enabled subsystem to be able to hold their values when the system is not enabled.

* For further information on how the variables get generated with different measurable classes refer to [MeasurableVariableClasses.docx](http://170.205.207.224:7001/si/viewrevision?projectName=d:/MKS_proj/T3_public/Tools/MbE/Matlab_Simulink/Misc/GuidelineDocuments/project.pj&revision=:member&selection=TargetLink_ReferenceDocs/MeasurableVariableClasses.docx)

# Optimized Vector Processing

* NOTE:

This section just shows the possibility of optimized vector processing through code generation settings. In general, one shall use generic WABCO code generation and optimization settings.

In case of project specific needs, the Technology team shall be involved.

* You can enable and disable loop generation for vector signals via the following Code Generator options:
  + - EfficientVectorHandling
    - LoopUnrollThreshold
* The setting of the EfficientVectorHandling option takes priority over the setting of the LoopUnrollThreshold option.
* Refer to section ‘Optimized Vector Processing’ of [TLAdvancedGuide.pdf](http://170.205.207.224:7001/si/viewrevision?projectName=d:/MKS_proj/T3_public/Tools/MbE/Matlab_Simulink/Misc/GuidelineDocuments/project.pj&revision=:member&selection=TargetLink_ReferenceDocs/TLAdvancedGuide.pdf)

# Others

# Dealing with NaN values

* Values that are not real or complex numbers are represented with a special value called NaN, which stands for "Not a Number". Expressions like 0/0 and inf/inf result in NaN
* Standard header function isnan can be used to check for NaN.
* One other way to do this would be creating a custom code block with following lines to check if an input is a NaN value:

if (ToBeGuarded\_input == ToBeGuarded\_input)

{

Output = ToBeGuarded\_input;

}

else

{

Output = GuardValue;

}

* Note:

ToBeGuarded\_input should be of class ’ volatile’ to prevent compiler optimization

# Usage of custom code blocks

* The Custom Code block allows us to implement the logic in the form of C syntax in a Simulink model. It also serves as an interface between legacy C code and Simulink model. Hence the block can call a function of an existing C file.
* For more details on the usage of custom code block Refer to [Matlab\_CustomCode\_Reference.docx](http://170.205.207.224:7001/si/viewrevision?projectName=d:/MKS_proj/T3_public/Tools/MbE/Matlab_Simulink/Misc/GuidelineDocuments/project.pj&revision=:member&selection=TargetLink_ReferenceDocs/Matlab_CustomCode_Reference.docx)

# Protection against division by zero

* A common function (lib\_SafeDivsion) is available to provide protection against division by zero and is advised for use instead of a normal division block.  
    
  Below are the observations of enabling protection against division by zero in a normal division operation -
* The TargetLink properties of division block provide an option for ‘Protection against division by zero’ in production code. This option shall be set if denominator to division block has zero in its expected values range.
* When this option is set, production code checks the denominator against zero, and if the denominator is zero, based on the sign of numerator, the output of division block is set to either minimum or maximum value of its data type range
* In addition to this, TargetLink properties of division block also has a provision to specify min and max limits to the output of division block. If ‘Protection against division by zero’ is enabled and in case of division by zero, the output will be set to either min or max value based on the numerator sign
* If ‘Protection against division by zero’ is enabled, then the code generation is affected by the inputs to the division block as follows:
  + - If the numerator is a calibration variable, then the sign check for numerator is generated/optimized based on the min/max limits specified in the TargetLink properties of calibration variable
    - If the denominator is a calibration variable, then the check for division by zero itself is generated/optimized based on the min/max limits specified in the TargetLink properties of calibration variable
* TargetLink Limitation:

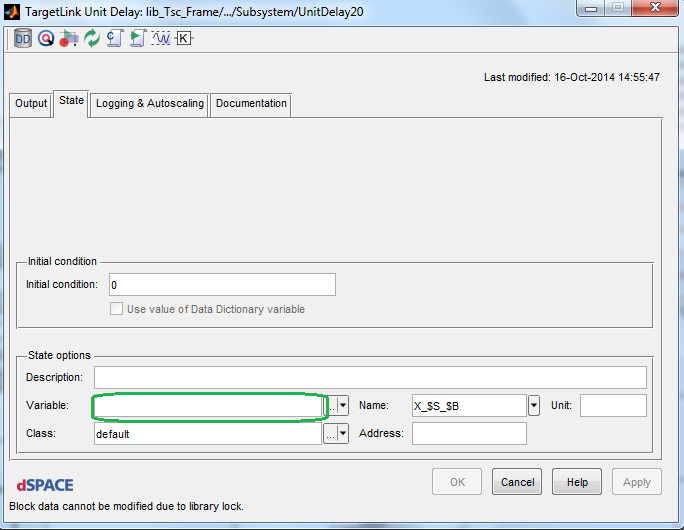
If the numerator is a calibration variable of signed integer or float data type, and if it is initialized to a positive value and if min/max limits are not specified, then the sign check for negative value of numerator is not generated

Solution: If the min/max values for the calibration variable are specified, then sign check for both negative and positive values of the numerator is generated

* The ‘Protection against division by zero’ option is not advisory for floating point operations, as it would generate equality checks for float variables which is against MISRA guidelines

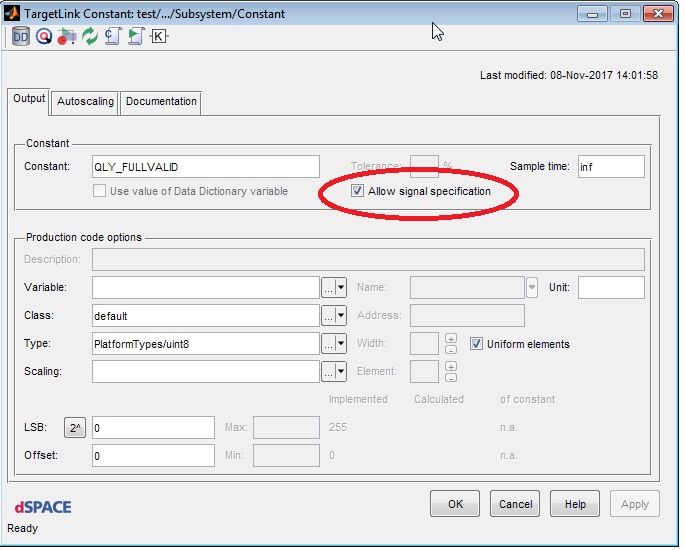
# Usage of unit delay blocks

* Usage of unit delay blocks in a TargetLink subsystem usually results in the generation of static variables and code for assignments
* The generation of these static variables and code for assignments can be avoided in few scenarios -
  + - When the output of a TargetLink subsystem is defined as a global variable and if this has to be looped back to that subsystem as an input too, then the state of the unit delay block can be mapped to the same DD variable as of the function output
    - When the Data of a unit delay block shall be measured, i.e. if the result shall be present in a global variable, the DD-variable shall not be assigned to the output but to the state of the 1/z block.



# Structure Types

* Structure types shall be defined explicitly in the DD (Typedefs) instead of using implicit structs. This avoids unnecessary copies of identical structures with just different type names.
* While using switch block with property set to inherit having bus as inputs and the bus created from constants, Signal specification should be provided inside constant block dialog.

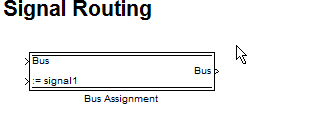


* Refer to lesson LL0105 of Lessons\_Learnt.xls for more information

<http://wabco-pe/Method_Tools/targetlink/MbSWDDocuments/LessonsLearned/Lessons_Learnt.xls>

# Bus Assignment Block

* When few of the signals from a large bus have to be processed and re-assigned to the same bus, the reassignment needs a lot of manual effort to model and also the model becomes complex
* The bus assignment block (a masked subsystem) from the TargetLink utilities can be used to perform this reassignment task



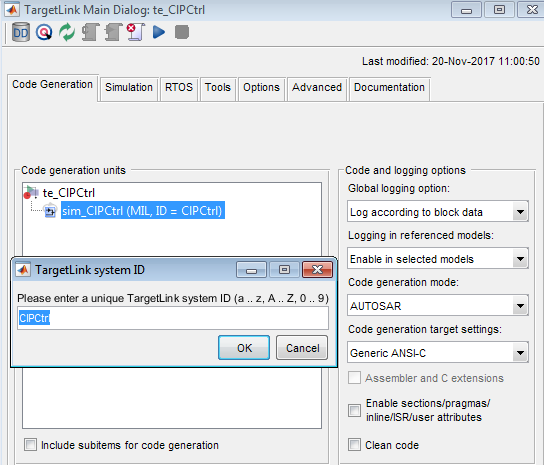
* Refer to lesson LL0108 of Lessons\_Learnt.xls for more information

<http://wabco-pe/Method_Tools/targetlink/MbSWDDocuments/LessonsLearned/Lessons_Learnt.xls>

# Naming conventions for blocks

* Avoid spaces for block names
* Give meaningful names for the blocks instead of default names. This would be helpful in tracking the variables from code to model or model to code

# Naming convention of header files for user defined typedefs

* The header file generated with user defined typedefs (udt) shall be named as udt\_<Swc.h>
  + In case if module reference creation is not required, the CG tool defines the user defined Typedefs in the file “udt\_<Swc.h>”. The CG tool uses the TargetLink system ID, which shall be specified as the “Swc” name and the typedef file shall be “udt\_<Swc.h>”. 
  + When module reference is created at Swc's level, the file name shall be specified as “udt\_<Swc.h>”.

# 64 Bit Operations

* 64 Bit Operations should be avoided wherever possible
* Their use leads to an increase in execution time and code size
* Example:

64-bit operations are generated by TargetLink if 32-bit data types are used as block inputs and due to enabled saturation or scaling, an intermediate 64-bit variable for the result must be introduced to prevent any undesired overflow or underflow.

See the figure below for example.

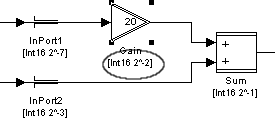


* Convert the variables manually to the same resolution using gain block(s) rather than leaving the conversion to the arithmetic blocks.
* 64 bit operations can also be avoided by explicitly specifying the min and max values of inports. Because in this case the calculations are performed with a smaller bit length

# Consistency of Scaling Parameters

* Consistent scaling parameters and data types within a model lead to efficient code with regard to execution time and code size.
* The combination of variables with different scaling parameters requires additional rescaling operations, leading to additional code.
* Example:

Below shows an example of improper scaling due to inconsistencies



# Avoidance of Saturation Specification

* Block outputs with specified saturation should be avoided whenever possible.
* It is often preferable to prevent over- and underflows by choosing wider data types 32 bit or modified scaling rather than by saturation.
* The saturation of block output variables is implemented via range checking using conditional statements. This leads to increased code size and execution time.

# Usage of Enabled subsystems

* Usage of enabled subsystems shall be reduced where possible
* Usage of an enabled subsystem creates static variables for all the outputs of the subsystem.
* NOTE:

The static variables can be avoided, in case the enabling is done via if-else-action and the outputs of if-else subsystems are connected to a merge block. With this approach, the outputs are calculated every cycle and TargetLink doesn’t create static variables to hold values until next activation

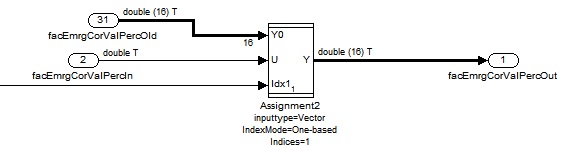
# TL properties for Charts, Triggered/ Enabled subsystems/ If\_action subsystems/ Atomic subsystems

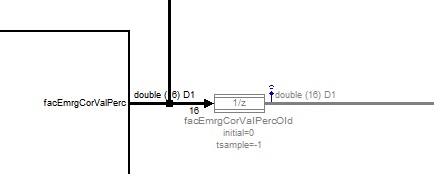
* All charts, Enabled, Triggered, If\_action and Atomic subsystems (with or without TL Function block) are created as functions in C code. If the variable classes for the inports and outports of these subsystems/charts are not specified, then unnecessary static variables may get created in the generated code.
* To avoid the generation of these unnecessary static variables, the Simulink inports and outports of these subsystems shall be enhanced to TargetLink ports and the appropriate variable classes shall be specified.

# Points on Resource Optimization

* Superfluous initializations may be created if 'Omit dispensable initializations' option is not activated for assignment blocks inside for loops
* In cases where Assignment, Selector and For Iterator blocks are operated with one-based index, Targetlink generates non-optimized code resulting in operations like 'vector[iteration - 1]'. Zero-based indexing is preferred.
* Usage of MERGEABLE class to avoid redundant RAM variables:
  + There would be scenarios where multiple RAM copies may get created for a single variable in a Simulink model. The user shall analyze the generated code in such scenarios and take possible measures to arrive at resource optimization. One way to address this scenario is by mapping the same DD variable to different elements of the model.
  + Use Case :

For example, in the case below the output of assignment operator goes as input to unit-delay. Now if the unit-delay value also needs to be measured using a variable with MEASURABLE\_GLOBAL class, TargetLink generates 3 RAM variables. One for the assignment operator, one for the unit-delay and one for the measurement variable.





A single RAM variable for unit-delay and measurement variable is possible if the measurement variable is mapped to the output state of unit-delay. Mapping the same measurement variable to assignment operator, would further optimize RAM consumption and create only a single RAM variable instead of three.

In this case the class for measurable variable shall also have the optimization property ‘MERGEABLE’. The variable class ‘REUSE\_STRUCT\_ELEMENT’ has this mergeable property and can be used for RAM optimization.

Usage of variables with mergeable class may also result in reduction of redundant lines of code for assignments resulting in ROM optimization.  
  
NOTE:   
Usage of mergeable variables always has a risk of unintended behavior. The user shall be careful w.r.t. their usage in the context of resource optimization

* Replacing math functions with customized implementation:
  + The user shall check if a customized implementation of a needed math function or any other function is already available, and shall use it for resource optimization

# Reference Documents

Few other reference documents on TargetLink guidelines/practices are available at below locations in MKS

* <http://170.205.207.224:7001/si/viewproject?projectName=d:/MKS_proj/T3_public/Tools/MbE/MatlabSimulink/Documents/ModellingGuidelines/TargetLink/project.pj>
* <http://170.205.207.224:7001/si/viewproject?projectName=d:/MKS_proj/T3_public/Tools/MbE/Matlab_Simulink/Documents/SoftwareDevelopmentWithTargetLink/project.pj>
* <http://170.205.207.224:7001/si/viewproject?projectName=d:/MKS_proj/MBSP_SW_Components/D12_Tooling/Documents/TargetLink_CodeGeneration/project.pj>