



Modeling Guidelines for MATLAB/Simulink/Stateflow and TargetLink



Version 1.0

TargetLink Product Management
dSPACE GmbH
33100 Paderborn
Germany
Contact:
ueisemann@dspace.de
mbeine@dspace.de

How to Contact dSPACE

Mail:	dSPACE GmbH Technologiepark 25 33100 Paderborn Germany
Tel.:	+4952511638-0
Fax:	+49525166529
E-mail:	info@dspace.de
Web:	http://www.dspace.com
General Technical Support:	support@dspace.de +4952511638-941 http://www.dspace.com/goto?support
TargetLink Support:	support.tl@dspace.de +4952511638-700

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/goto?support> for software updates and patches

Important Notice

This document contains proprietary information that is protected by copyright. All rights are reserved. Neither the documentation nor software may be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form, in whole or in part, without the prior written consent of dSPACE GmbH.

© Copyright 2006 by:
dSPACE GmbH
Technologiepark 25
33100 Paderborn
Germany

This publication and the contents hereof are subject to change without notice. dSPACE GmbH makes no warranty of any kind with regard to this publication, including but not limited to the implied warranties of merchantability or fitness for a particular purpose. dSPACE GmbH shall not be liable for errors contained herein or direct, indirect, special, incidental, or consequential damages in connection with the furnishing, performance, or use of this publication.

Brand names or product names are trademarks or registered trademarks of their respective companies or organizations.

Version and Change History

Version	Date	Originator	Modification
Draft	2005-09-01	Eisemann	Initial Version compiled
Draft	2005-10-18	Eisemann	Translations and Modifications
0.9	2005-10-31	Eisemann	Version 0.9 completed
0.9.1	2005-11-07	Eisemann	Version 0.9.1 under development
0.9.2	2005-11-20	Eisemann	Review Feedback partially incorporated
0.9.3	2005-12-15	Eisemann	Spelling and grammar corrected, rules added for MATLAB R14 SP2, specifically bus capable blocks (rule 2.1.5), duplicate Inports (rule 2.10.3), Data Type Conversion block (rule 2.11.2), Bus Creator block (rule 2.12.8), Stateflow limitations (rule 2.16.13 to 2.16.20),
0.9.4	2005-12-15	Eisemann	Additional rules for data types included (now rule 8.1 to 8.6, previous rules now 8.7 to 8.12), introduction for chapter 8 added, merging of Stateflow limitation 2.16.9 and 2.16.10.
0.9.5	2005-12-20	Eisemann	Minor layout corrections
0.9.9	2006-04-04	Eisemann	Minor corrections (spelling errors etc.) in diverse chapters
1.0	2006-04-06	Eisemann	Small corrections/modifications in 2.7.2, 2.7.3, 2.12.5 and other minor changes.

Table of Contents

I	Introduction	5
1	Exemplary Architectural Aspects	7
1.1	Process Blocks	7
1.2	Component Blocks	11
1.3	Execution Order Specification	13
1.4	General	16
2	Language Subset	18
2.1	General	18
2.2	Continuous Library	21
2.3	Discontinuities Library	22
2.4	Discrete Library	24
2.5	Logic and Bit Operations Library	27
2.6	Lookup Tables Library	29
2.7	Math Operations Library	31
2.8	Model Verification Library	35
2.9	Model-Wide Utilities Library	37
2.10	Ports & Subsystems Library	38
2.11	Signal Attributes Library	42
2.12	Signal Routing Library	44
2.13	Sinks Library	52
2.14	Sources Library	53
2.15	User-Defined Functions Library	54
2.16	Stateflow	55
3	Model Parameters	70
4	Name Spaces	72
5	Comments and Units	73
6	Layout Issues	77
7	Simulation Parameters	80
8	Data Types	83
9	Handling of Look-up Tables	95
10	Function Handling	99
11	Variable Handling	105
12	Stateflow Code	113
13	Fixed-Point Code Generation	124
14	MISRA Compliance	134

I Introduction

Purpose

The modeling guidelines contained in this document help to ensure a seamless development process from function development in MATLAB®/Simulink®/Stateflow® to automatic production code generation for electronic control units (ECUs) using TargetLink from dSPACE. The rules described are only relevant to the development of controllers, but do not apply to the design of plant models, for instance. Adherence to the guidelines helps to minimize the amount of rework necessary to implement the controller model in highly efficient code.

The issues covered in these guidelines are (among others):

- Specification of a structure for MATLAB/Simulink/Stateflow models to enable code generation with TargetLink in a subsequent process.
- Style issues for a transparent, well structured modeling style, simplifying model reviews and model exchange.
- Setting of TargetLink code generation options and implementation information for the generation of efficient code.
- Issues relating to the reliability and safety of models and the generated code, for instance, compliance with the MISRA C rules.

It should be noted that the guidelines in this document should not be regarded as a substitute for the TargetLink documentation. The rules in these guidelines do not cover all the above aspects in their entirety. Customer feedback on topics to be treated in future versions of the guidelines is highly welcome.

This publication and the contents hereof are subject to change without notice. dSPACE GmbH makes no warranty of any kind with regard to this publication, including but not limited to the implied warranties of merchantability or fitness for a particular purpose. dSPACE GmbH shall not be liable for errors contained herein or direct, indirect, special, incidental, or consequential damages in connection with the furnishing, performance, or use of this publication.

Version Dependency

The modeling guidelines in this document are primarily designed for a function development with the MATLAB releases R13 and R14, or more specifically:

- R13
- R13 SP1
- R13 SP2
- R14
- R14 SP1
- R14 SP2
- R14 SP3

The relevant TargetLink versions in the focus of these guidelines are all releases since TargetLink 2.0 as there are:

- TL 2.0
- TL 2.0.5

- TL 2.0.6
- TL 2.0.7
- TL 2.1
- TL 2.1.5
- TL 2.1.6

Where an individual rule depends on specific MATLAB or TargetLink versions, this is indicated in the text of the rule. For other MATLAB and TargetLink versions than those listed above, the guidelines might be a useful addition as well, although absolute compatibility cannot be assured.

Guideline Contents

As a whole, the guidelines are divided into two distinct parts:

- Rules relevant to the function development phase
- Rules relevant to the software development phase in which the model has to be implemented in highly efficient C code

Rules for function development deal with modeling in Simulink/Stateflow exclusively. No TargetLink-specific code generation settings or options are described here. However, since code generation with TargetLink depends on the modeling style in Simulink, TargetLink requirements are naturally taken into account. Rules for function development are described in chapters [1](#) to [8](#).

Rules for software development deal with target implementation information and special code generation options which are TargetLink-specific. The rules are described in the chapters [8](#) to [14](#).

Rule Structure

The rules in this document all have a common structure containing some or all of the following sections:

- A title
- A rule description, specifying the content of the rule
- A reason for the rule indicating the consequences of non-compliance with it
- An additional explanation providing some background information
- An optional reference to additional documents
- One or multiple optional examples for further explanation of the rule or highlighting related issues

In addition, each chapter or section in this document might contain an introduction providing some background information on the rules, some remarks on their relevance, or a short summary.

1 Exemplary Architectural Aspects

The rules in this section provide a partial description of a proper architecture for MATLAB/Simulink/Stateflow models. They are not mandatory in so far as the architecture described here is only an example. If an architecture is already in use, this chapter can be skipped.

The guidelines in this chapter introduce so-called process blocks (relating to real-time requirements) and component blocks (relating to functional requirements). Component blocks and process blocks ensure the construction of a hierarchical model architecture, with which external interfaces to the software, interfaces between software components, and software modules can be specified.

It should also be noted, that Target has built-in support for OSEK-compliant operating systems via the additional TMOS_OSEK module. An architecture based on TargetLink's OSEK support could be a partial alternative for the rules in this chapter but is not discussed here.

1.1 Process Blocks

Process blocks are special subsystems to structure the application algorithm according to real-time requirements. A process block represents a part of software which has a real-time requirement attached to it (a process). A series of processes with identical real-time requirements is combined in an operating system task, which is executed in a predefined order.

1.1.1 Realization of Process Blocks

Process blocks must be realized in the form of *Atomic Subsystems*. It is advisable to use a special color scheme for process blocks, e.g:

- green subsystem
- green Inports
- red Outports

Purpose

Subsystem encapsulation, proper graphical description.

Remark

Process blocks are executed as atomic units at specific time rates as a result of real-time requirements.

Example

A process block has to be realized as *Atomic Subsystem* by specifying the proper options in the *Block Parameters*, see fig. 1.

1.1.2 Contents of Process Blocks

Application algorithms must be modeled entirely within process blocks. Operations which have to be ported to the target later on must not be modeled outside of process blocks, where only blocks from the *Signal Attributes*, *Signal Routing*, *Sinks* and *Sources* library are permitted. Process blocks must not contain other process blocks.

Purpose

Proper architecture description.

Remark

Putting control functions in different process blocks enables the integration of the functionality in different operating system tasks. Since process blocks constitute atomic units, they cannot contain further process blocks themselves.

Example

A Model part with three process blocks is displayed in fig. 2. Each process block contains a certain functionality for the control algorithm, while no arithmetic operations are performed outside the process blocks.

1.1.3 Interfaces for Process Blocks

Signal busses crossing the border of process blocks are not permitted in TargetLink versions prior to TargetLink 2.1. In addition, trigger signals must not lead out and iteration limits for *For Iterator Subsystems* must not lead into a process block.

Purpose

Code generation with TargetLink.

Remark

A process block is transformed into a *TargetLink Root Subsystem* in subsequent code implementation steps. TargetLink imposes some restrictions on the signals crossing a *TargetLink Root Subsystem*.

References

- [TargetLink Production Code Generation Guide \[1\]](#), Limitations in Creating a TargetLink Subsystem, , bus signals
- [TargetLink Production Code Generation Guide \[1\]](#), Block-Specific Limitations

Example

Fig. 3 shows an example of impermissible use of busses for TargetLink versions prior to TargetLink 2.1.

1.1.4 Names of Process Blocks

The names of subsystems for process blocks and their respective *Outports* should be unique on a model-wide basis (globally unique). In addition, it is recommended to specify a particular naming convention for those subsystems and ports. The names of corresponding signal lines of *Outports* and *Inports* for process blocks should be identical.

Purpose

Naming conventions, proper graphical description.

Remark

Improper naming of process blocks leads to improper data signal communication during software integration.

Example

Fig. 4 shows an example of proper signal designation for process block signals.

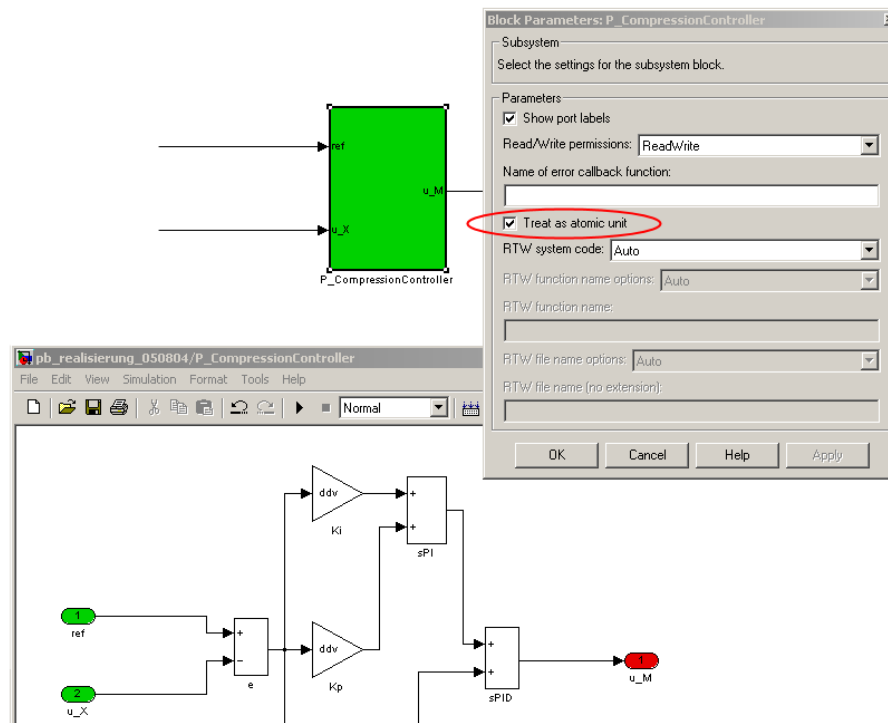


Fig. 1: A proper process block.

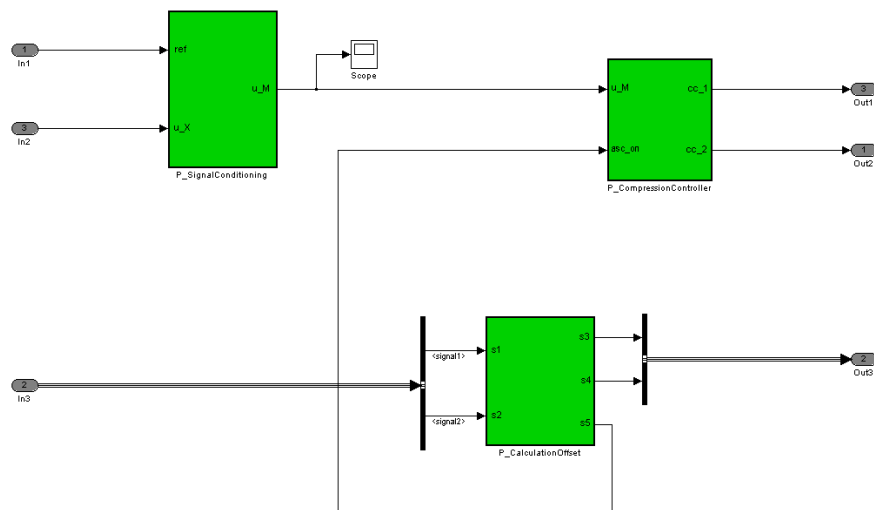


Fig. 2: Process blocks in a model.

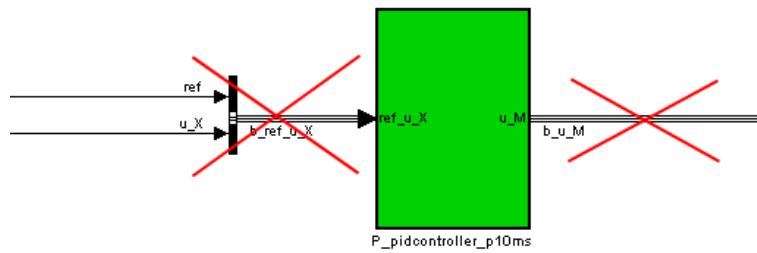


Fig. 3: Improper interface for process blocks for TargetLink versions prior to TargetLink 2.1.

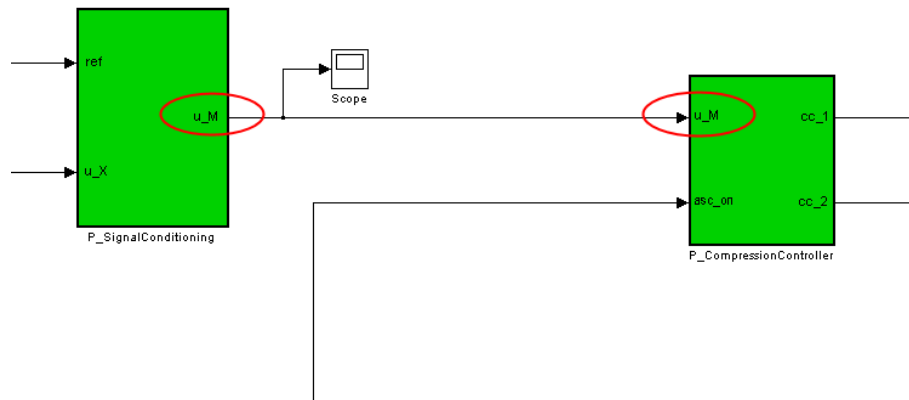


Fig. 4: Signal designation.

1.2 Component Blocks

Component blocks are specially designated subsystems which enable the logical grouping of one or multiple process blocks. The construction of component blocks shall be based entirely on functional aspects and not on real time requirements. A component block is the smallest unit which can be mapped to an ECU and thereby guarantees that all process blocks contained in it are mapped to the ECU.

A component block represents a logically separable, reusable software component. The set of all software components and processes forms the application software.

1.2.1 Realization of Component Blocks

The component blocks must be realized in the form of *(Virtual) Subsystems*. To distinguish between component blocks and other blocks, it is recommended to use a particular color scheme, for instance:

- Grey Subsystems
- Grey Inports and Outports

Purpose

Encapsulation and reuse, proper graphical notation.

Remark

The model architecture with different component blocks ensures the mapping onto the hardware architecture and the reuse of different components.

Example

Fig. 5 shows a component block.

1.2.2 Contents of Component Blocks

Components blocks must consist of process blocks. Multiple process blocks, which serve exactly one particular purpose, shall be combined in one component block. Component blocks must not contain other component blocks.

Purpose

Proper model structure.

Remark

The model architecture with different component blocks ensures the mapping to the hardware architecture and the reuse of different components.

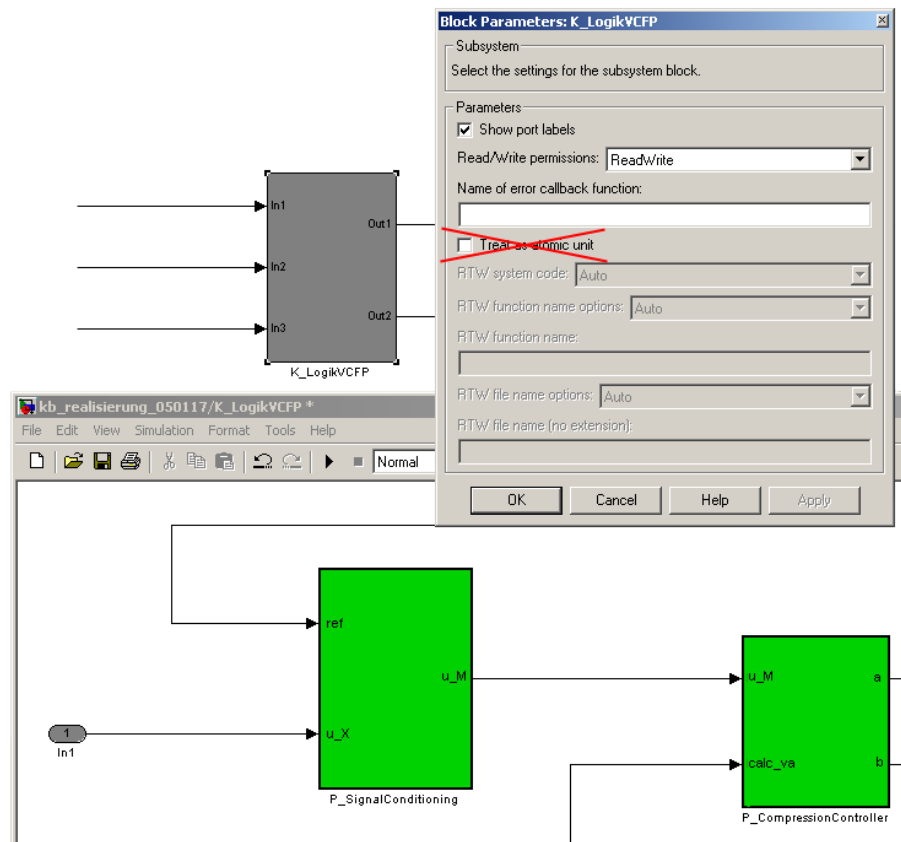


Fig. 5: Component block.

1.3 Execution Order Specification

The execution order (scheduling) for the simulation of a MATLAB/Simulink/Stateflow model can be defined by means of a control unit in the form of a Stateflow Chart within the model. Thereby, the execution order of individual process blocks is properly defined and documented for subsequent implementation in software. The control unit, which only serves simulation purposes in Simulink, is not to be ported to the target.

1.3.1 Control Unit for Proper Execution Order

The execution order for individual process blocks should be controlled by a central unit, for instance in the form of a Stateflow chart.

Purpose

Proper task and interface description, specification of the execution order.

Remark

The control unit properly defines the execution order for the process blocks and is entirely user-adaptable to specific requirements.

References

- [Controller Style Guidelines for Production Intent Using MATLAB, Simulink and Stateflow V1.0.0 \[8\], 4.4.1.1](#)

Example

Fig. 6 shows an example of a central control unit in the form of a Stateflow chart.

1.3.2 Activation of Process Blocks

The activation of process blocks by the control unit shall be performed in the form of *function-call subsystems* which shall be wrapped around the process blocks.

Purpose

Proper task and interface description, proper simulation order.

Remark

The separation of process blocks and *Function-Call Subsystem* ensures the simulation of models with or without schedulers using process block libraries.

References

- [Controller Style Guidelines for Production Intent Using MATLAB, Simulink and Stateflow V1.0.0 \[8\], 4.4.1.1](#)

Example

Fig. 7 shows the proper activation of a process block.

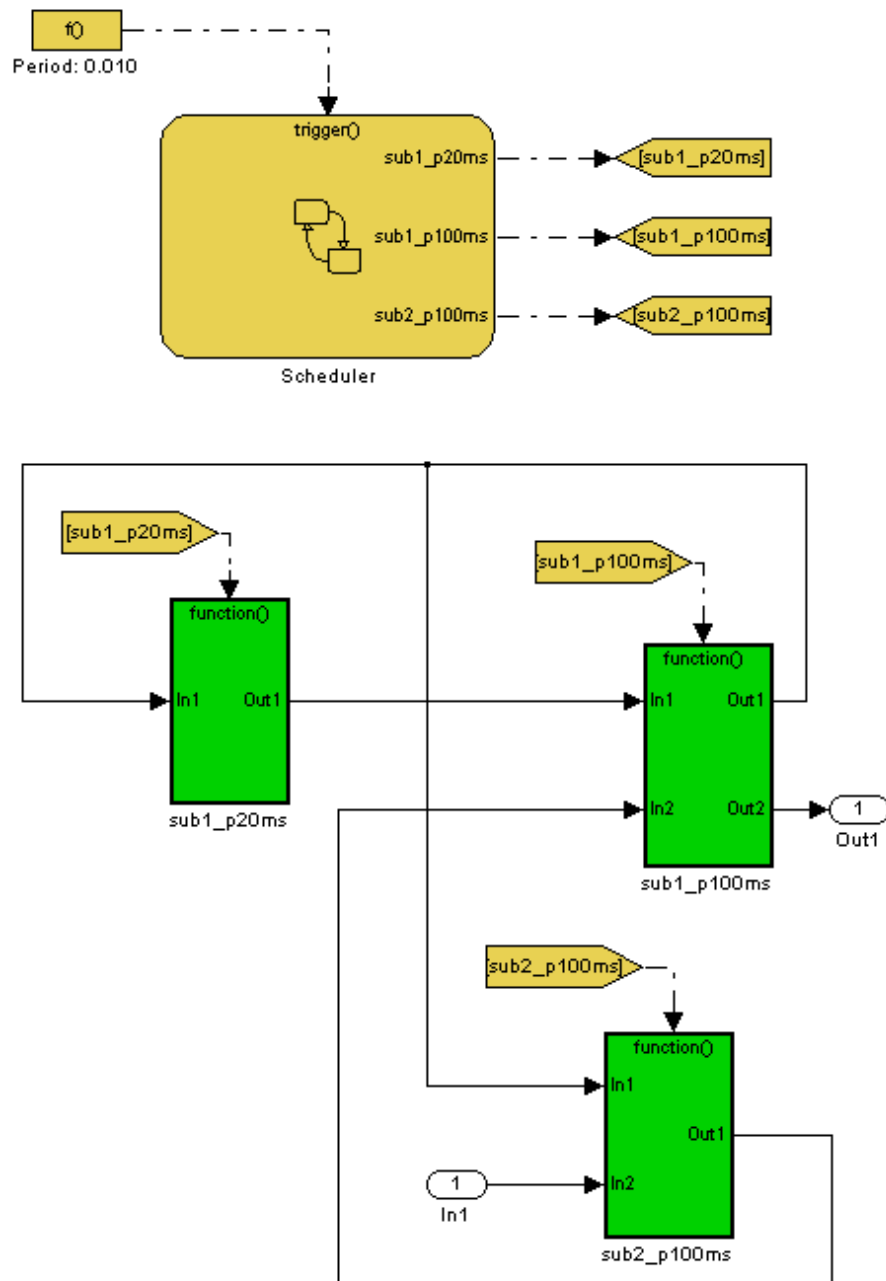


Fig. 6: Specification of the execution order.

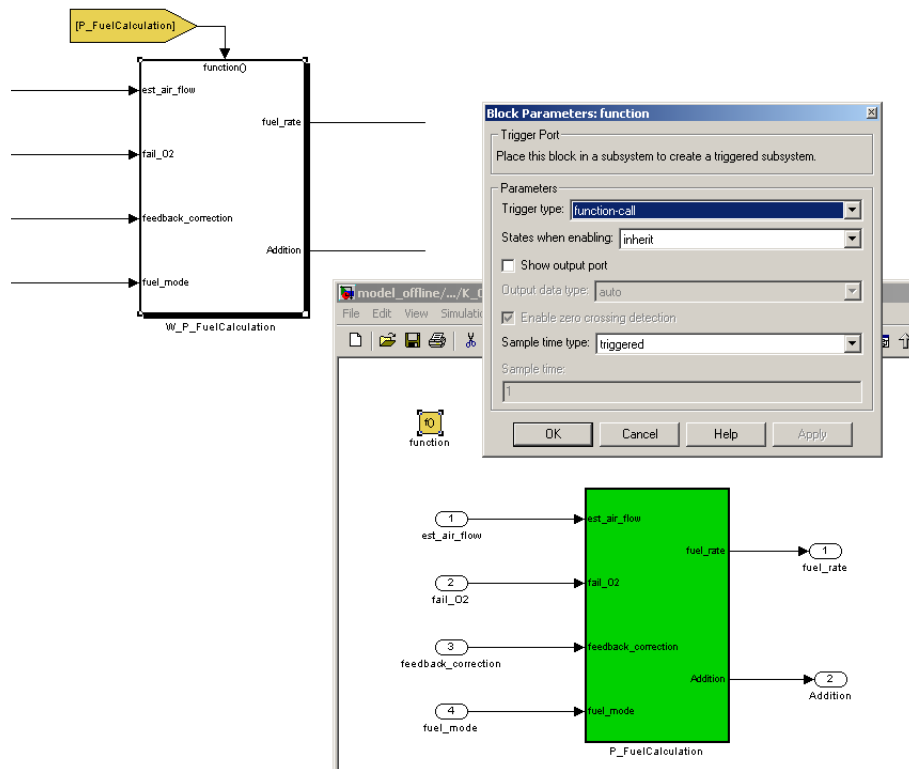


Fig. 7: The process block *P_FuelCalculation* is encapsulated by the *function-call* subsystem *W_P_FuelCalculation*.

1.4 General

The guidelines presented in this section ensure proper hierarchies and versioning of models and components.

1.4.1 Model Hierarchy

Besides component blocks and process blocks, *(virtual) subsystems* should be used for the purpose of proper model structuring whenever appropriate.

Purpose

Proper model structure and design.

Remark

Different hierarchies in the form of subsystems enable transparent modeling and easy inspections.

Example

Fig. 8 shows the grouping of two component blocks in a *(virtual) subsystem*.

1.4.2 Model Administration

It is generally recommended to store process blocks and component blocks as *library block* in a Simulink library whenever appropriate.

Purpose

Versioning and proper administration.

Remark

The rule enables proper versioning and reusability of component blocks and process blocks. The libraries containing them can be administered using a version control system.

Example

Fig. 9 shows the storage of individual process blocks in a Simulink library.

1.4.3 Model Versioning

It is recommended to use version numbers for each Simulink model/library and support a mechanism for displaying the change history.

Purpose

Traceability of changes.

Remark

Proper versioning of the models/libraries greatly enhances the traceability of changes.

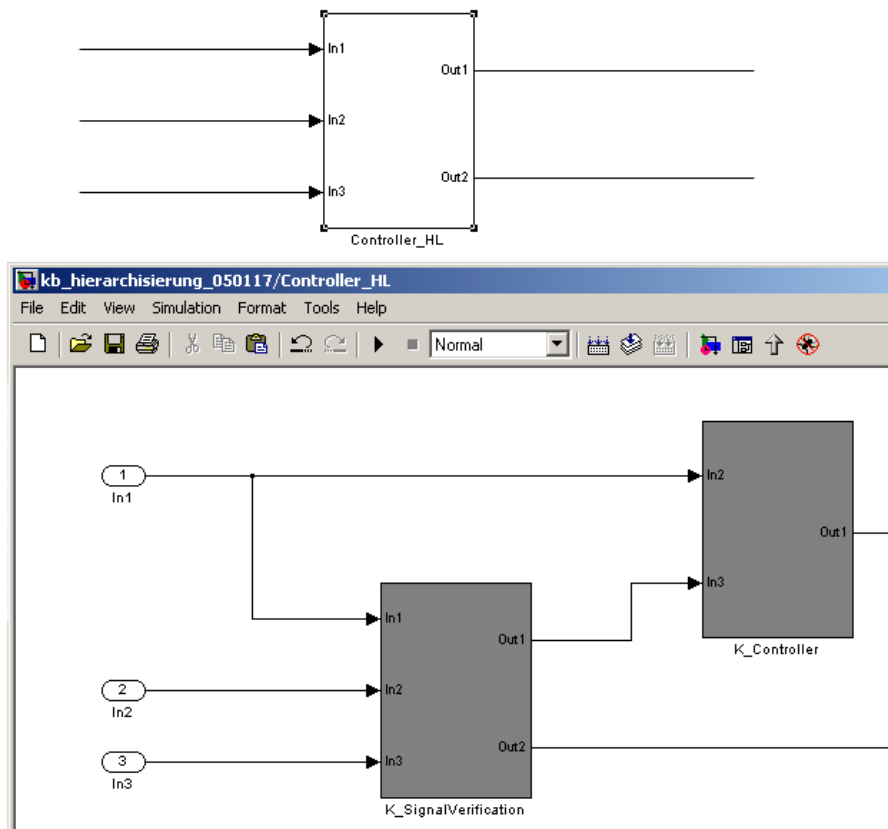


Fig. 8: Optional subsystems for proper model structuring.

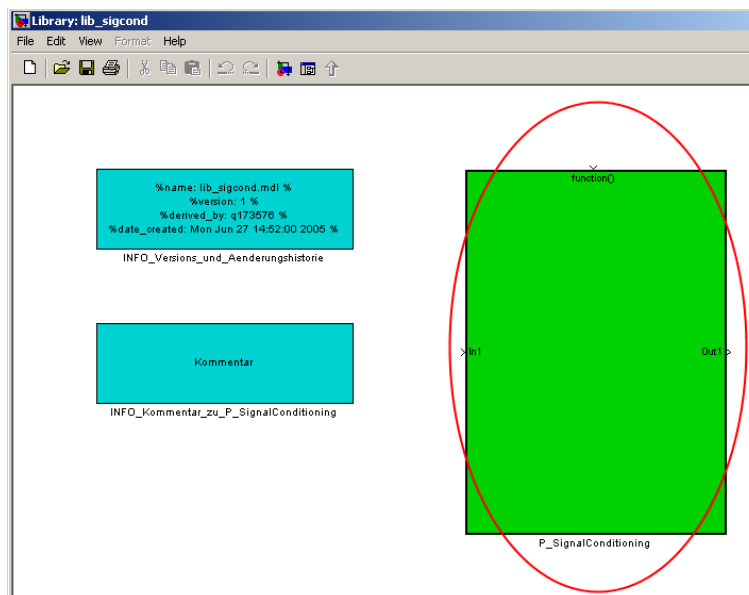


Fig. 9: Simulink libraries can be used for proper versioning of process blocks.

2 Language Subset

The rules in this chapter define a language subset of MATLAB/Simulink/Stateflow that helps to ensure successful subsequent code generation with TargetLink. The language subset contains a subset of the blocks supported in Simulink and some limitations with regard to block parameters and as signal types.

The restrictions listed in this chapter are only relevant for those parts of a model which shall be implemented in code using TargetLink. Other model parts like plant models, a central control logic etc., which are not subject to production code generation, can be modeled regardless of the TargetLink guidelines.

2.1 General

This section contains general rules which are relevant for almost all blocks and signals in the controller model.

2.1.1 Block Priorities

Block priorities must not be used to specify the block execution order in Simulink.

Purpose

Code generation with TargetLink.

Remark

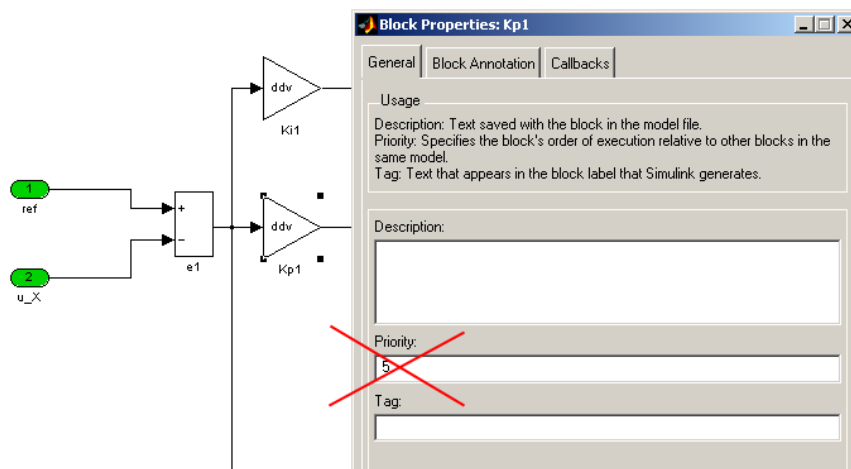
The use of block priorities leads to an intransparent modeling style and is therefore not supported (ignored) by TargetLink.

References

- [TargetLink Production Code Generation Guide \[1\]](#), General Limitations, Block priorities and block scheduling

Example

Inadmissible use of *priorities* in the *Block Properties*.



2.1.2 Permissible Signals and Parameters

Matrix parameters with dimensions larger than two (*n-dimensional arrays*) must not be used neither in Simulink nor in Stateflow. Two dimensional parameters (*two-dimensional arrays*) can be used only in the following blocks:

- Look-Up Table (2-D)
- Interpolation (n-D) using PreLook-Up
- Direct Look-Up Table (n-D)
- Discrete State-Space

Matrix signals (*two-dimensional signal arrays*) are only permitted for local signals in Stateflow. They must not be used in Simulink. In addition, all signals must be real not complex.

Purpose

Code generation with TargetLink.

Remark

Matrix signals are not supported by TargetLink and two-dimensional parameters only in the above mentioned cases.

2.1.3 Unconnected Signals

All signals which are not connected to further blocks shall be terminated using the *Terminator* block. Unconnected block input signals shall be terminated using a *Ground* block.

Purpose

Proper modeling style.

Remark

The use of *Terminator* and *Ground* blocks leads to a better modeling style and helps to avoid functional errors.

References

- [Controller Style Guidelines for Production Intent Using MATLAB, Simulink and Stateflow V1.0.0 \[8\], 4.5.1.4](#)

2.1.4 Signal Names

The names of signals should be unique on a model-wide (global) scale if the signals must be measurable later on.

Purpose

Proper signal designation, model readability.

Remark

Unique names are required for signals with global relevance.

Example

Fig. 10 shows one example of improper and one of permissible signal naming.

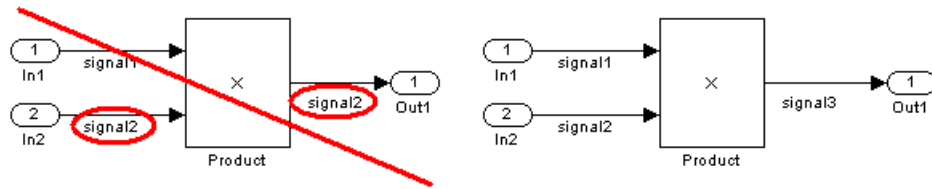


Fig. 10: Improper and proper signal designation.

2.1.5 Bus Capable Blocks.

The following nonvirtual blocks, which are bus-capable in Simulink since R14, should not be driven by a bus:

- Unit Delay
- Multipoint Switch
- Merge
- Switch

However, the following nonvirtual blocks can be driven by busses:

- Rate Transition
- Zero-Order Hold

Purpose

Code generation with TargetLink.

Remark

If busses are fed into the above blocks, the code generator treats the output like a vector with identical data types, not like a bus.

Example

Fig. 11 shows the impermissible use of a *Unit Delay* block, which is driven by a bus.

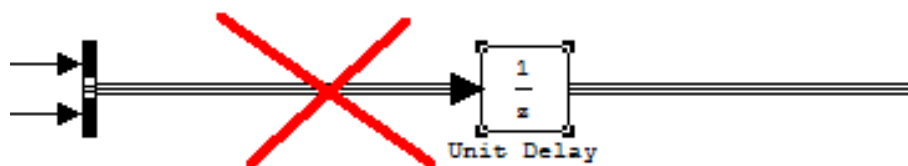


Fig. 11: TargetLink currently does not support *Unit Delay* blocks which are driven by busses.

2.2 Continuous Library

This section contains restrictions with regard to the use of blocks from the *Continuous* library.

2.2.1 No Blocks from the Continuous Library

No blocks from the *Continuous* library shall be used for the controller part of a model. However, blocks from the library can naturally be used for modeling plants.

Purpose

Code generation with TargetLink.

Remark

A model implementation in the form of code is discrete by nature. TargetLink therefore supports only time-discrete blocks, so that potential time-continuous parts of a controller model need to be discretized before production code generation.

2.3 Discontinuities Library

This section contains restrictions with regard to the use of blocks from the *Discontinuities* library.

2.3.1 Permissible Blocks of the Discontinuities Library

Only the following blocks from the *Discontinuities* library should be used:

- Backlash
- Dead Zone
- Dead Zone Dynamic
- Rate Limiter
- Relay
- Saturation
- Saturation Dynamic

Purpose

Code generation with TargetLink.

Remark

TargetLink supports only a restricted subset of blocks to ensure the generation of efficient code. The blocks *Backlash* and *Dead Zone* are not built-in TargetLink blocks but implemented as masked sub-systems and *Custom Code* blocks. This might lead to increased effort for fixed-point scaling.

Example

Fig. 12 shows the permissible blocks.

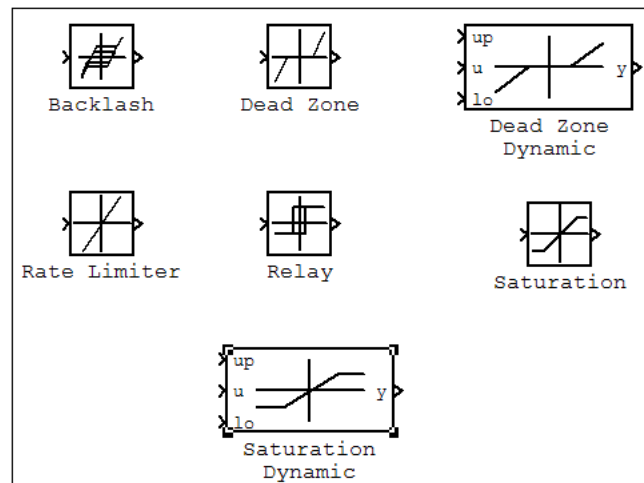


Fig. 12: Permissible blocks from the *Discontinuities* library.

2.3.2 Limitations with Regard to the Dead Zone Block

Vector signals (*one-dimensional signal arrays*) are not permitted as input signals for the *Dead Zone* block.

Purpose

Code generation with TargetLink.

Remark

TargetLink does not support vector signals for the *Dead Zone* block.

2.4 Discrete Library

This section contains restrictions with regard to the use of blocks from the *Discrete* library.

2.4.1 Permissible Blocks of the Discrete Library

Only the following blocks from the *Discrete* library should be used:

- Discrete Filter
- Discrete State-Space
- Discrete Transfer Fcn
- Discrete-Time Integrator
- Unit Delay
- Zero-Order Hold

Purpose

Code generation with TargetLink.

Remark

TargetLink supports only a restricted subset of blocks to ensure the generation of efficient code.

Example

Fig. 13 shows the permissible blocks.

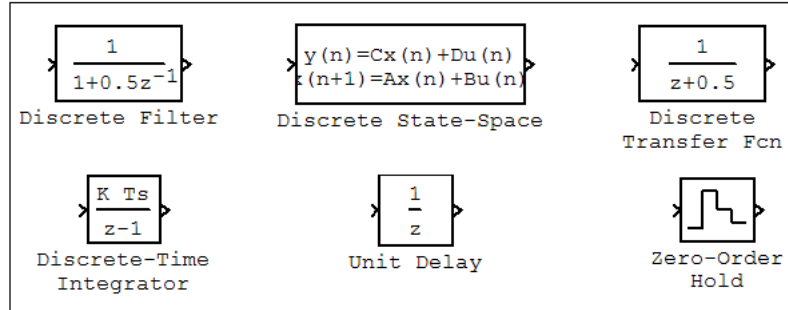


Fig. 13: Permissible blocks from the *Discrete* library.

2.4.2 Limitations with Regard to the Discrete-Time Integrator Block

Vector signals (*one-dimensional signal arrays*) are not permitted for the *Discrete-Time Integrator* block. In addition, the *level* option for the *External reset* property as well as the *Show state port* option are also not permitted.

Purpose

Code generation with TargetLink.

Remark

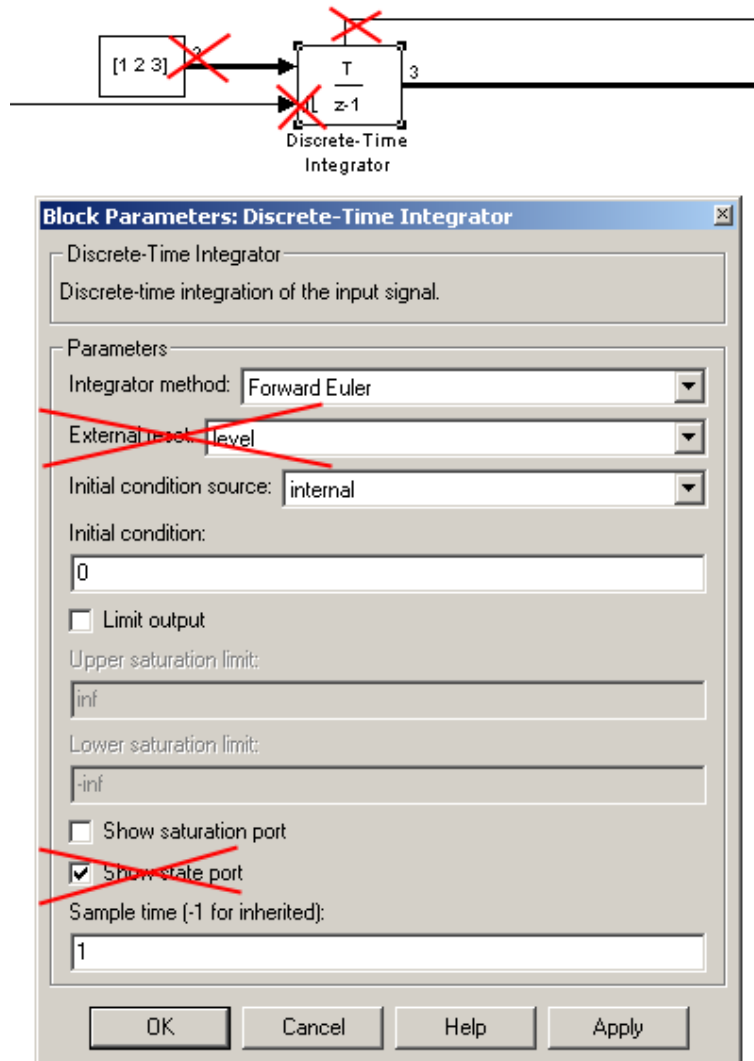
TargetLink does not support these options/signals for the *Discrete-Time Integrator* block.

References

- [TargetLink Production Code Generation Guide \[1\]](#), TargetLink Limitations, Block specific Limitations, Discrete-Time Integrator

Example

Fig. 14 shows the inadmissible *Block Parameters* and options for the *Discrete-Time Integrator* block.

Fig. 14: Options for the *Discrete Time Integrator* block.

2.5 Logic and Bit Operations Library

This section contains restrictions with regard to the use of blocks from the *Logic and Bit Operations* library, which was newly introduced with MATLAB release R14.

2.5.1 Permissible Blocks of the Logic and Bit Operations Library

Only the following blocks from the *Logic and Bit Operations* library should be used:

- Bitwise Operator
- Logical Operator
- Relational Operator

Purpose

Code generation with TargetLink.

Remark

TargetLink supports only a restricted subset of blocks to ensure the generation of efficient code. The *Bitwise Operator* is not a built-in TargetLink Block but realized via a *Custom Code* block.

Example

Fig. 15 shows the permissible blocks.

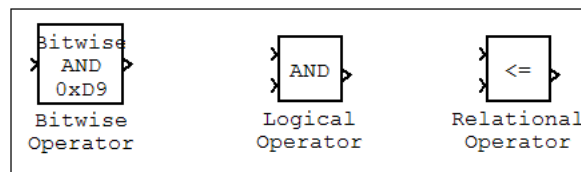


Fig. 15: Permissible blocks from the *Logic and Bit Operations* library.

2.5.2 Limitations with Regard to the Relational Operator and Logical Operator Blocks

For *Relational Operator* and *Logical Operator* blocks, the *Output data type mode* property must be set to *Boolean*, *Logical*, or *float('double')*.

Purpose

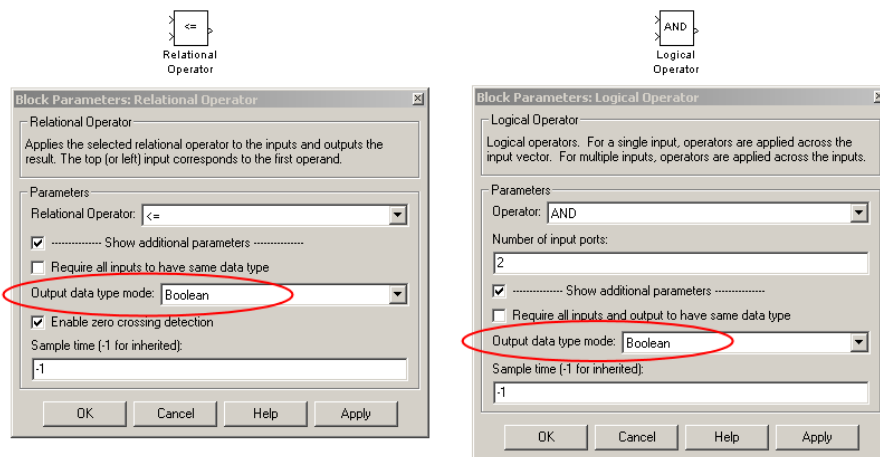
Code generation with TargetLink.

Remark

TargetLink supports only the boolean and double data types at the output of these blocks. The data type is determined by the global Simulink option *Implement logic signals as boolean data (vs. double)* available under *Configuration Parameters/Optimization*.

Example

Proper data type options for the *Relational Operator* and *Logical Operator* blocks.



2.6 Lookup Tables Library

This section contains restrictions with regard to the use of blocks from the *Lookup Tables* library.

2.6.1 Permissible Blocks of the Look-up Tables Library

Only the following blocks from the *Lookup Tables* library should be used:

- Direct Lookup Table (n-D)
- Interpolation (n-D) using PreLookup
- Lookup Table
- Lookup Table (2D)
- Lookup Table (n-D)
- PreLookup Index Search

Purpose

Code generation with TargetLink.

Remark

TargetLink supports only a restricted subset of blocks to ensure the generation of efficient code.

Example

Fig. 16 shows the permissible blocks.

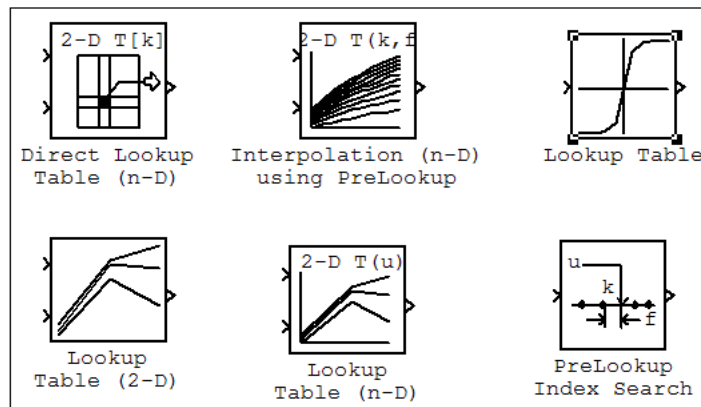


Fig. 16: Permissible blocks from the *Lookup Table* library.

2.6.2 Limitations with Regard to the Look-up Table (n-D) and Direct Look-up Table (n-D) Blocks

The blocks *Look-Up Table (n-D)* and *Direct Look-up Table (n-D)* must be used only in conjunction with dimensions smaller than three.

Purpose

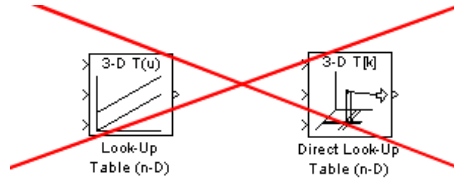
Code generation with TargetLink.

Remark

TargetLink does not support parameters with dimensions greater than two.

Example

Inadmissible configuration of a *Look-Up Table (n-D)* and *Direct Look-up Table (n-D)* block with three-dimensional maps.



2.7 Math Operations Library

This section contains restrictions with regard to the use of blocks from the *Math* library.

2.7.1 Permissible Blocks of the Math Library

Only the following blocks from the *Math* library should be used:

- Abs
- Add
- Assignment
- Divide
- Gain
- Math Function
- MinMax
- Product
- Product of Elements
- Rounding Function
- Sign
- Subtract
- Sum
- Sum of Elements
- Trigonometric Function
- Bitwise Logical Operator
- Logical Operator
- Relational Operator

Purpose

Code generation with TargetLink.

Remark

TargetLink supports only a restricted subset of blocks to ensure the generation of efficient code. The *Bitwise Logical Operator* and *Rounding Function* blocks are not built-in TargetLink blocks but realized as Custom Code blocks. Since Release R14, the blocks *Bitwise Logical Operator*, *Logical Operator*, and *Relational Operator* are part of the *Logic and Bit Operations* library.

Example

Fig. 17 shows the permissible blocks.

2.7.2 Limitations with Regard to Operand Numbers for the Product Block

If a fixed-point data type is specified for the output of the *Product* block, the number of inputs must not exceed two. If a vector signal is fed into a *product block*, the number of elements of the vector must not exceed two. However, no restrictions apply to the number of input signals (or element numbers) if a floating-point data type is specified for the output of the *Product* block.

Purpose

Code generation with TargetLink.



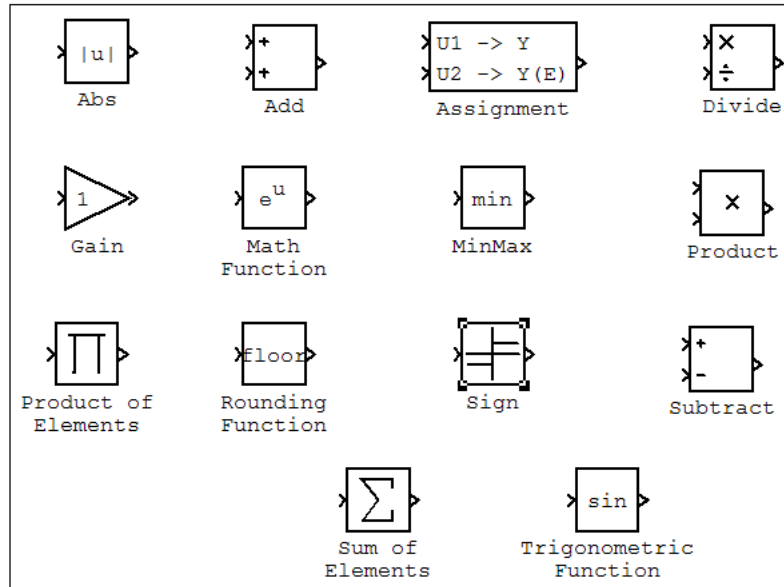


Fig. 17: Permissible blocks from the *Math* library.

Remark

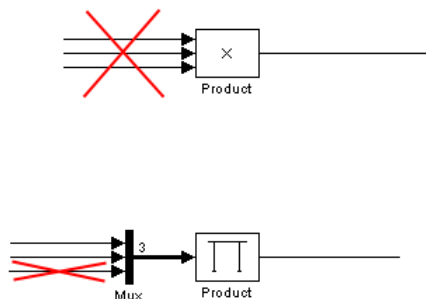
The generation of proper fixed-point code for the *Product* block with more than two operands requires the specification of scaling information for intermediate results. TargetLink therefore requires the number of input variables not to exceed two for fixed-point data types.

References

- [TargetLink Production Code Generation Guide \[1\]](#), TargetLink Limitations, Block-Specific Limitations, Product Block

Example

Inadmissible use of a *Product* block with more than two operands for fixed-point code. The block has to be transformed into a hierarchical structure of *Product* blocks for fixed-point code generation.



2.7.3 Limitations with Regard to Parameters for the Product Block

Calculating the reciprocal of an input via the *Number of inputs: /* setting for the *Block Parameters* is only permitted for scalar input signals. In addition, calculating the reciprocal value $1/(x \cdot y)$ of a product of input values via the settings *Number of inputs: //* in the *Block Parameters* is not permitted either.

Both restrictions only apply if a fixed-point data type has been specified for the output of the *Product* block. No restrictions apply in case of a floating-point output.

Purpose

Code generation with TargetLink.

Remark

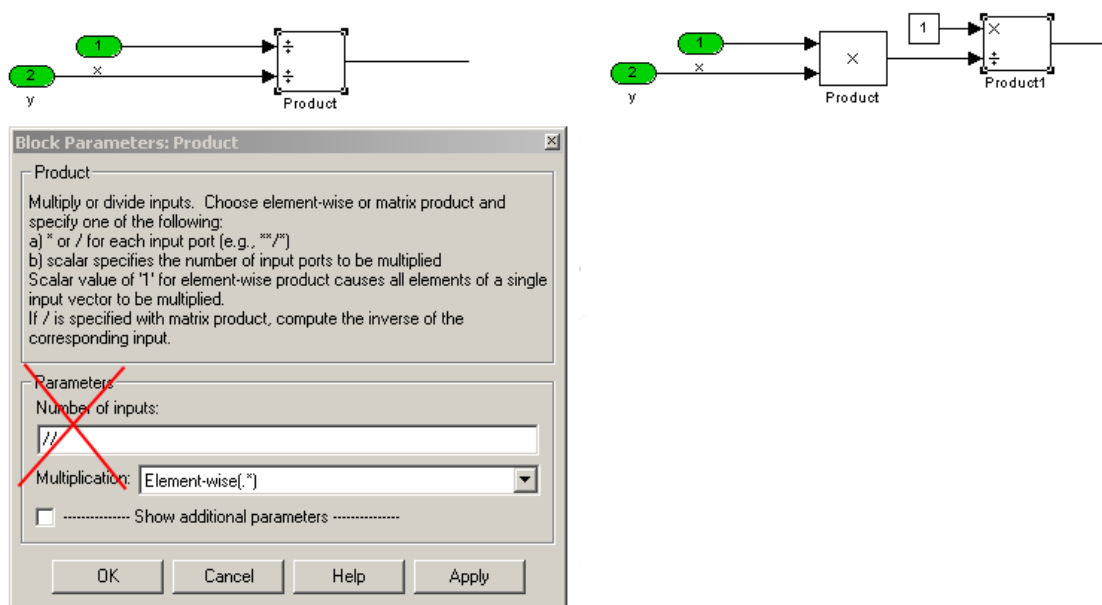
Proper fixed-point code generation requires the specification of scaling information for intermediate results.

References

- [TargetLink Production Code Generation Guide \[1\]](#), TargetLink Limitations, Block-Specific Limitations, Product Block

Example

The following images show how the reciprocal of a product of input values can be calculated in TargetLink.



2.7.4 Limitations with Regard to the Sum Block

The *Icon shape: rectangular* option should be used for the *Block Parameters* of the *Sum* block.

Purpose

Code generation with TargetLink.

Remark

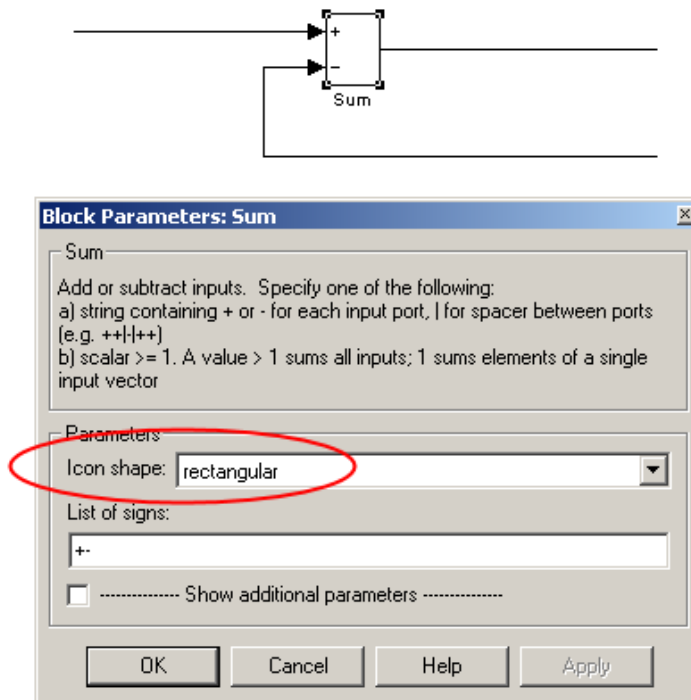
Round icon shapes for the *Sum* block are transformed into rectangular shapes during conversion from Simulink to TargetLink. On some occasions, this can lead to improper connections.

References

- [TargetLink Production Code Generation Guide \[1\]](#), TargetLink Limitations, Block-Specific Limitations, Sum Block

Example

Proper options for the *Block Parameters* of the *Sum* block.



2.7.5 Restrictions with Regard to the Math Function Block

For the *Math Function* block, the *Function: magnitude^2* option should not be specified. The *Function: square* option should be used as an alternative.

Purpose

Code generation with TargetLink.

Remark

The option is not supported by TargetLink.

2.8 Model Verification Library

This section contains the permitted language subset from the *Model Verification* library.

2.8.1 Permissible Blocks of the Model Verification Library

The following blocks from the *Model Verification* library can be used:

- Assertion
- Check Discrete Gradient
- Check Dynamic Gap
- Check Dynamic Lower Bound
- Check Dynamic Range
- Check Dynamic Upper Bound
- Check Input Resolution
- Check Static Gap
- Check Static Lower Bound
- Check Static Range
- Check Static Upper Bound

Purpose

Code generation with TargetLink.

Remark

The blocks listed above have an impact during model-in-the-loop simulation but are ignored during code generation and software-in-the-loop simulation.

Example

Fig. 18 shows the permissible blocks.

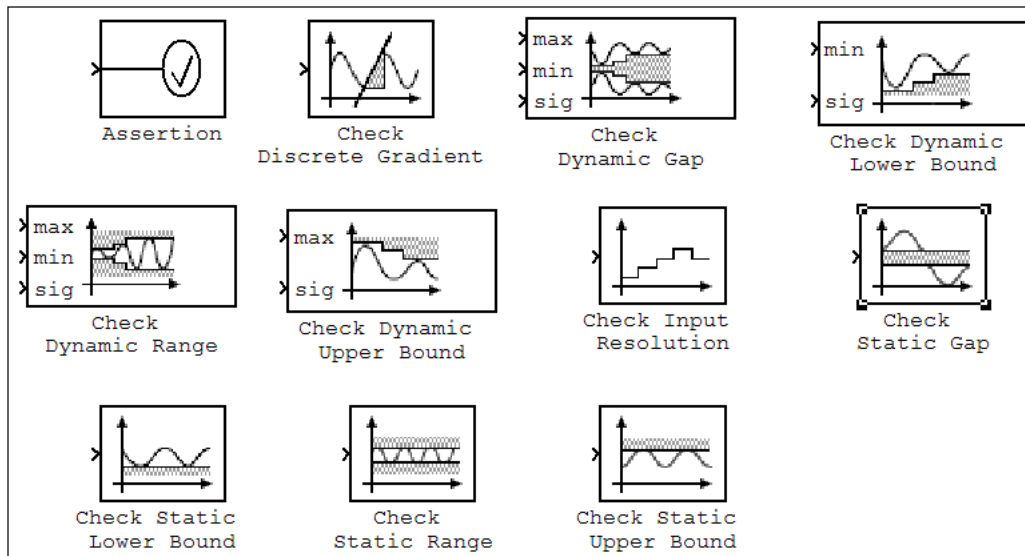


Fig. 18: Permissible blocks from the *Model Verification* library.

2.9 Model-Wide Utilities Library

This section contains restrictions with regard to the use of blocks from the *Model-Wide Utilities* library.

2.9.1 Permissible Blocks of the Model-Wide Utilities Library

Only the following blocks from the *Model-Wide Utilities* library should be used:

- Block Support Table
- Model Info
- Doc Block

Purpose

Code generation with TargetLink.

Remark

TargetLink supports only a restricted subset of blocks to ensure the generation of efficient code.

Example

Fig. 19 shows the permissible blocks.

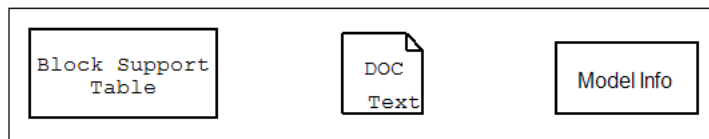


Fig. 19: Permissible blocks from the *Model-Wide Utilities* library.

2.10 Ports & Subsystems Library

This section contains restrictions with regard to the use of blocks from the *Ports & Subsystems* library.

2.10.1 Permissible Blocks of the Ports Subsystems Library

All blocks from the library can be used with the exception of the *Model* block, which was newly introduced with MATLAB release R14.

Purpose

Code generation with TargetLink.

Remark

TargetLink supports all combinations of subsystems in Simulink. The *Model* block however, which is used for the Model referencing feature, is currently not supported.

Example

Fig. 20 shows the permissible blocks from the *Ports & Subsystems* library.

2.10.2 Use of the Switch Case Block

The *Show default case* option for the *Switch Case* block should be set and the default output must be properly connected.

Purpose

Proper and transparent modeling.

Remark

All Switch-case instructions in the generated code should have a default branch to avoid undefined states during program execution.

Example

Figure 21 shows a *Switch Case* block with activated *Show default case* option.

2.10.3 Duplicate Inports

Duplicate Inports, which were introduced with Release R14, must not be used.

Purpose

A transparent modeling style, code generation with TargetLink.

Remark

Duplicate Inports lead to an intransparent modeling style prone to errors because the connection between individual ports gets lost. As a result, it is rather intransparent how many input variables a subsystem has and modifications can have unexpected side-effects. TargetLink does not support duplicate Inports.

Example

Fig. 22 and 23 show the impermissible use of duplicate Inports.

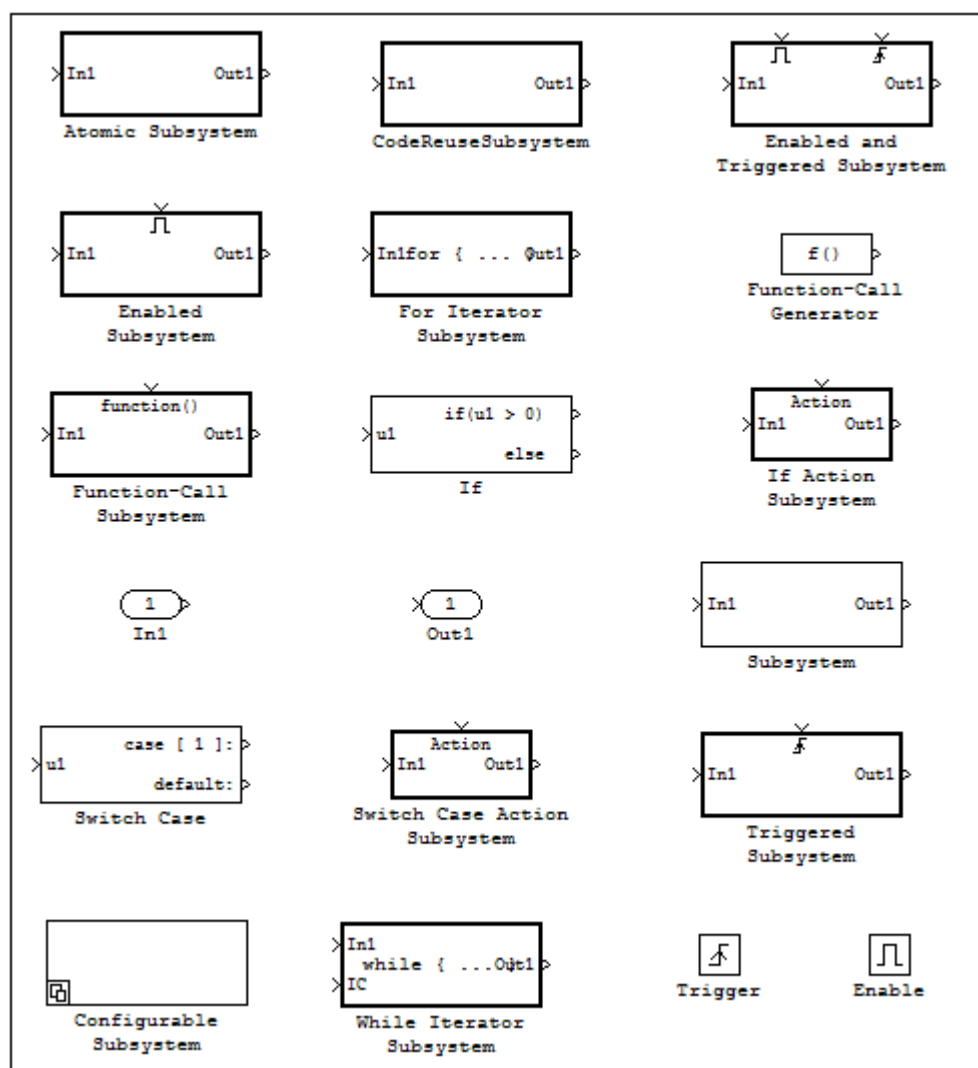


Fig. 20: Permissible blocks.

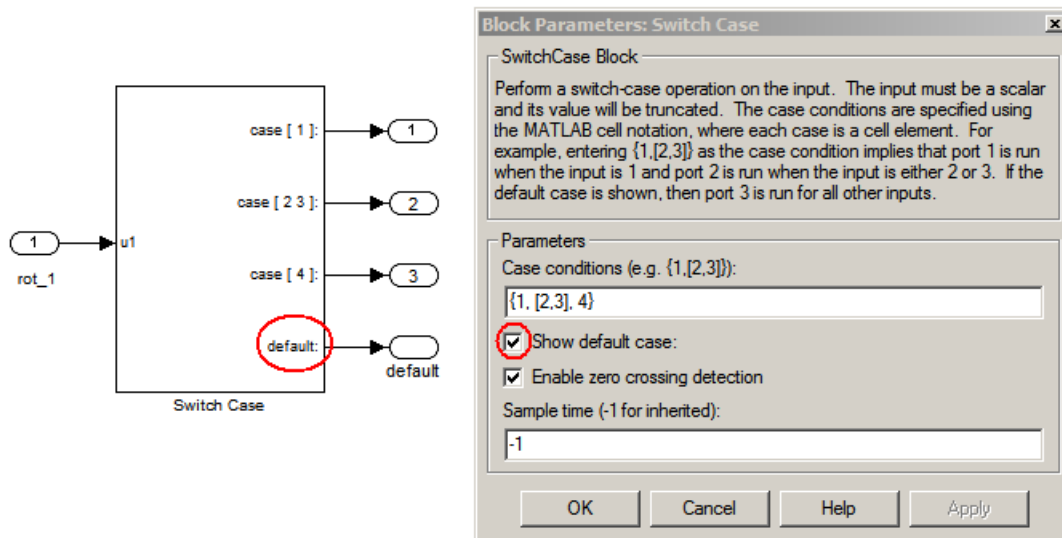


Fig. 21: Proper options.

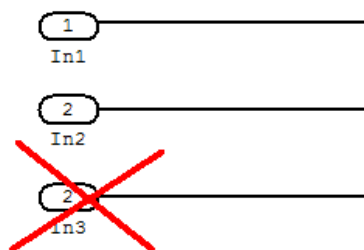


Fig. 22: Duplicate Inports are created by performing a copy operation on a Simulink *Inport* and a subsequent *Paste Duplicate Inport* operation. The duplicated Inport is not an independent signal but a reference to the original Inport. Despite being optically disconnected, both ports are identical as far as the simulation is concerned, which is indicated through the identical port number. Changes to the parameters of one Inport are immediately reflected in the other Inport also.

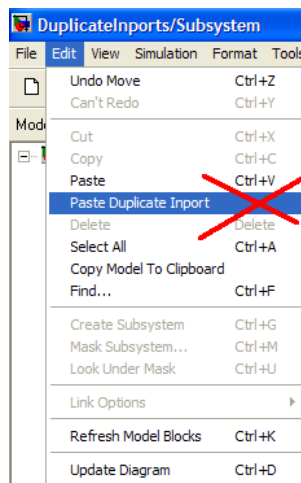


Fig. 23: Specification of duplicate imports, which are not supported by TargetLink.

2.11 Signal Attributes Library

This section contains restrictions with regard to the use of blocks from the *Signal Attributes* library.

2.11.1 Permissible Blocks of the Signal Attributes Library

Only the following blocks from the *Signal Attributes* library should be used:

- Data Type Conversion
- Rate Transition

Purpose

Code generation with TargetLink.

Remark

TargetLink supports only a restricted subset of blocks to ensure the generation of efficient code.

Example

Fig. 24 shows the permissible blocks from the library.

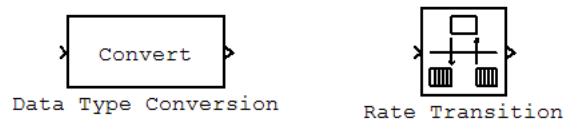


Fig. 24: Permissible blocks from the *Signal Attributes* library.

2.11.2 Use of the Data Type Conversion Block

The rounding functionality for the conversion of floating-point numbers to integers must not be used. Moreover, the *Saturate on integer overflow* option must not be used either. Hence, it has to be made sure that neither saturation nor rounding operations occur.

Purpose

Code generation with TargetLink.

Remark

The *Data Type Conversion* block is completely ignored during code generation and only influences the model-in-the-loop simulation. The saturation functionality as well as the different rounding modes, which were mainly introduced with R14, are therefore ignored either. The user has to make sure that neither rounding nor saturation nor overflows occur.

Example

Fig. 25 shows the critical functionality for the *Data Type Conversion* block, which is currently not supported by TargetLink. Regardless of the settings, the user has to make sure that neither rounding nor saturation nor overflows occur.

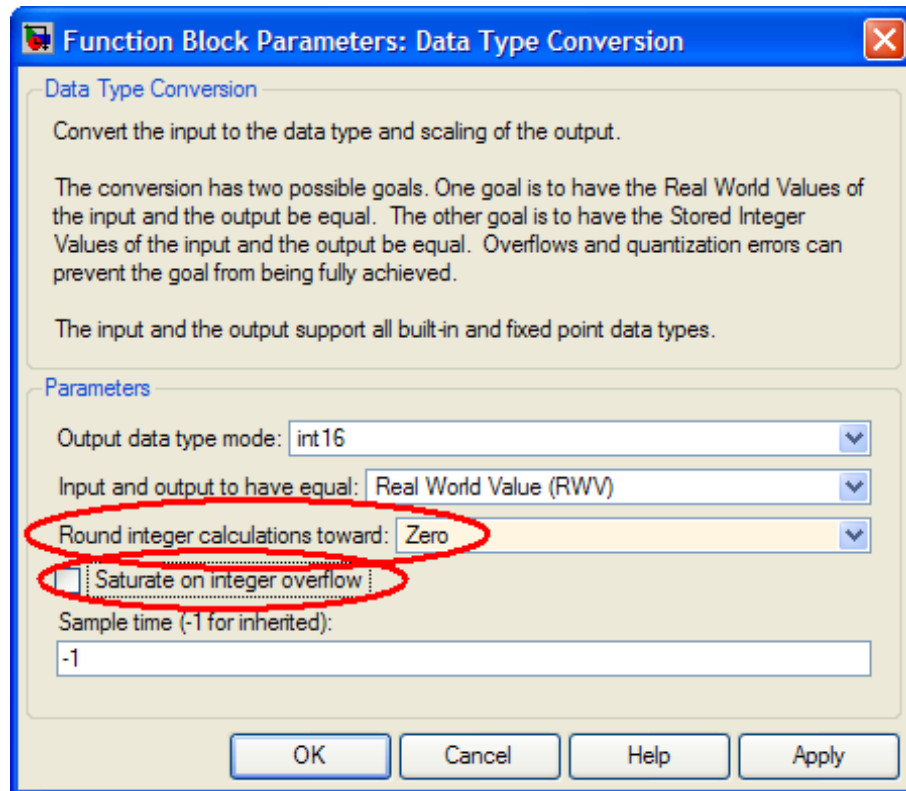


Fig. 25: The above properties of the *Data Type Conversion* block.

2.12 Signal Routing Library

This section contains restrictions with regard to the use of blocks from the *Signal Routing* library.

2.12.1 Permissible Blocks of the Signal Routing Library

All blocks from the *Signal Routing* library can be used with the exception of the *Manual Switch* block.

Purpose

Code generation with TargetLink.

Remark

The conversion of other blocks is not supported by TargetLink, partly because there is no reasonable counterpart in generated code.

Example

Fig. 26 shows the permissible blocks from the *Signal Routing* library.

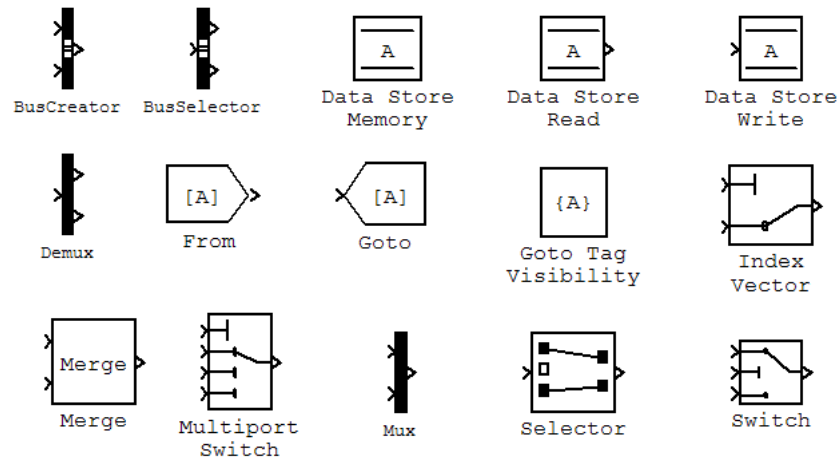


Fig. 26: Permissible blocks.

2.12.2 Restrictions with Regard to the Data Store Memory/Read/Write Blocks

The *Data Store Memory*, *Data Store Read*, and *Data Store Write* blocks must not be used to exchange data between different TargetLink subsystems. In addition, they should be used only in those cases where their relationship is still transparent, for instance, within the same subsystem.

Purpose

Code generation with TargetLink.

Remark

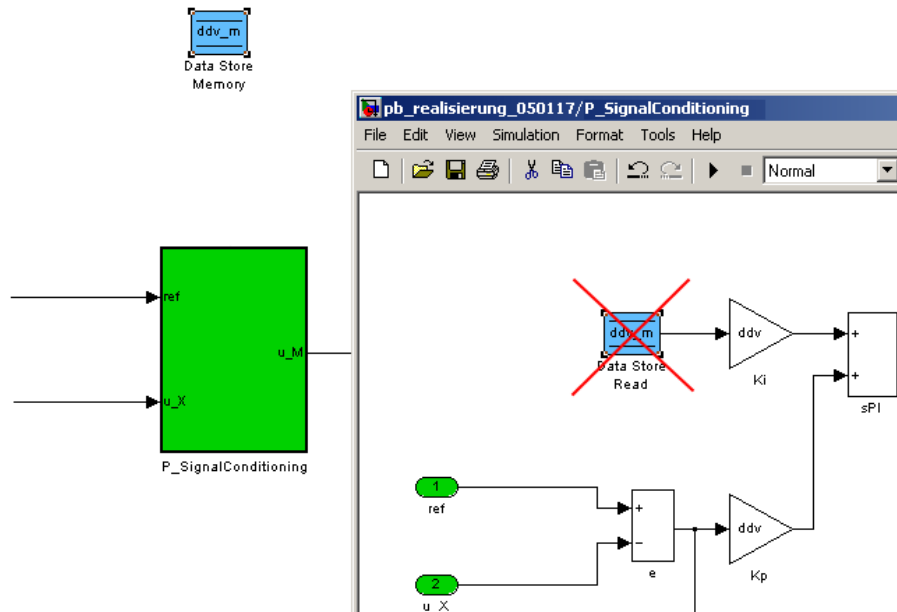
Data exchange via the above blocks across subsystems leads to intransparent models. In addition, code generation with TargetLink is performed for all TargetLink subsystems individually, requiring that interface variables at the boundary of a TargetLink subsystem must be properly defined.

References

- [TargetLink Production Code Generation Guide \[1\]](#), Block-Specific Limitations, Data Store

Example

Inadmissible use of *Data Store Memory* and *Data Store Read* blocks in combination with a subsystem to be converted into a TargetLink subsystem later on.



2.12.3 Restrictions with Regard to the From and Goto Block

The *From* and *Goto* blocks must not be used to connect blocks residing in different TargetLink subsystems. In addition, they should be used only in those cases where the relationships between the corresponding *From* and *Goto* block are transparent, for instance within a single subsystem and with *Tag visibility: local*.

Purpose

Code generation with TargetLink.

Remark

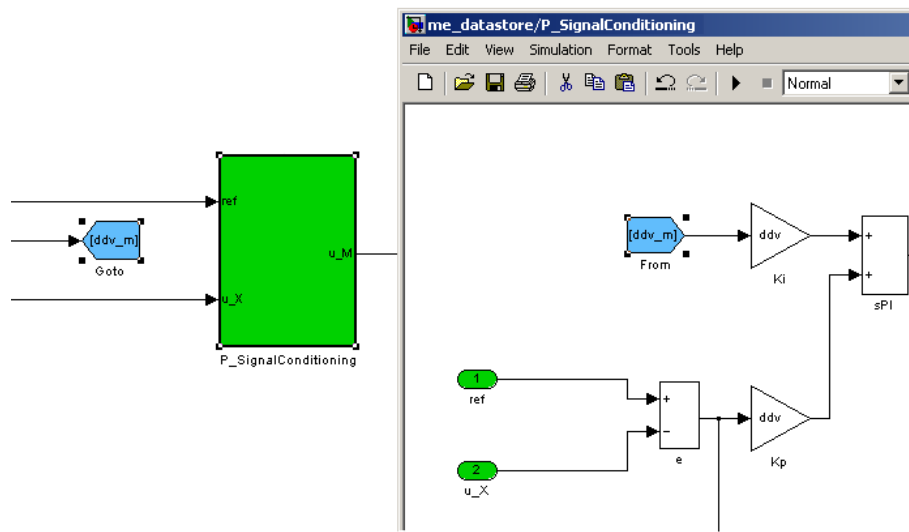
Data exchange via the above blocks across subsystems leads to intransparent models. In addition, code generation with TargetLink is performed for all TargetLink subsystems individually, requiring that interface variables at the boundary of a TargetLink subsystem must be properly defined.

References

- [Controller Style Guidelines for Production Intent Using MATLAB, Simulink and Stateflow V1.0.0 \[8\]](#), 4.5.1.5

Example

Illegal use of *From* and *Goto* blocks in combination with a block to be converted into a TargetLink subsystem later on.



2.12.4 Restrictions with Regard to the Merge Block

The *Allow unequal port widths* parameter must not be checked for the *Merge* block.

Purpose

Code generation with TargetLink.

Remark

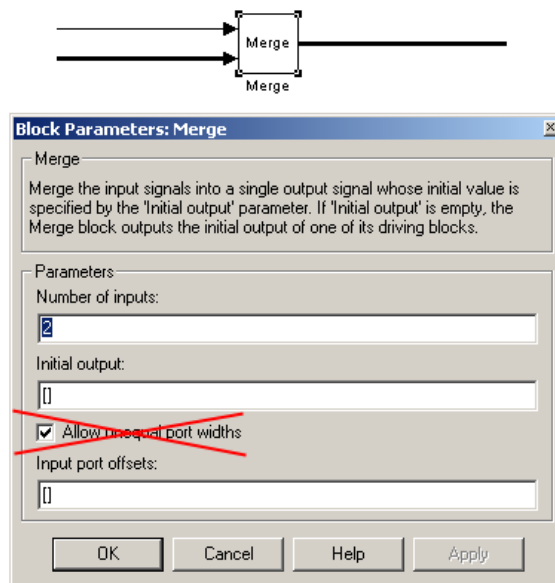
TargetLink does not support the option for the reason of code efficiency.

References

- [TargetLink Production Code Generation Guide \[1\]](#), TargetLink Limitations, Block-Specific Limitations, Merge block

Example

Unsupported *Block Parameters* for the *Merge* block.



2.12.5 No Cascades of Merge Blocks

In TargetLink versions prior to TargetLink 2.1.6, cascades of *Merge* blocks are not permitted. Instead, the hierarchy must be flattened using only one *Merge* block with multiple inputs. Since TargetLink 2.1.6, cascades of *Merge* blocks are fully supported.

Purpose

Code generation with TargetLink.

Remark

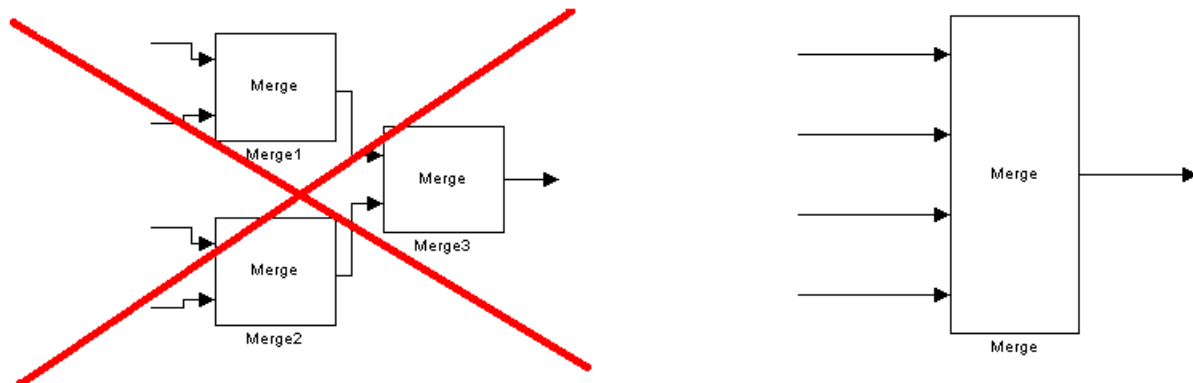
TargetLink does not support cascades of *Merge* blocks for the reason of code efficiency.

References

- [TargetLink Production Code Generation Guide \[1\]](#), TargetLink Limitations, Block-Specific Limitations, Merge block cascades

Example

Using a *Merge* block with multiple inputs instead of a cascade.



2.12.6 Use of the Multiport Switch Block

The *Multiport Switch* block in Simulink does not support a default path. Proper behavior of the *Multiport Switch* block under all circumstances should therefore be ensured by restricting the potential values of the *control port*, for instance, by employing some additional logic.

Purpose

Proper modeling, compliance with MISRA Rule 30: All automatic variables must have been assigned a value before being used.

Remark

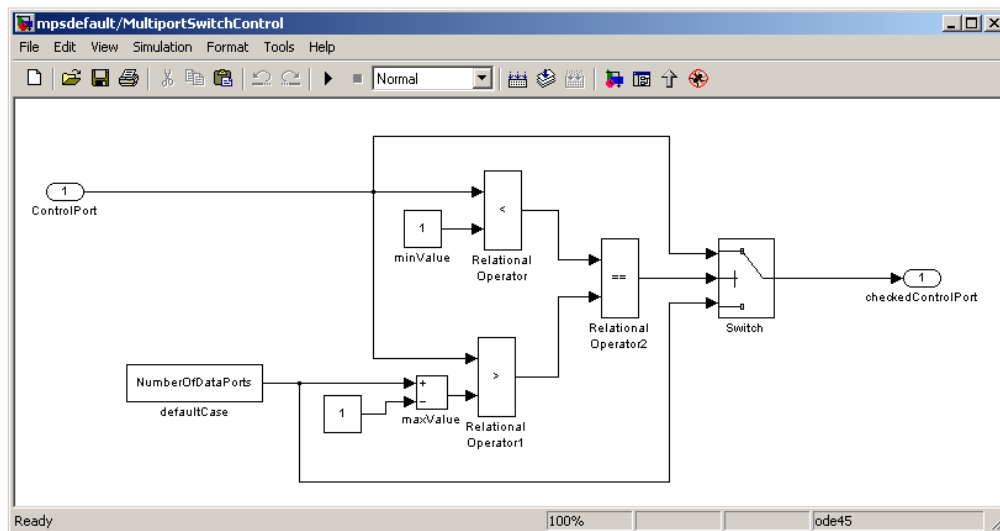
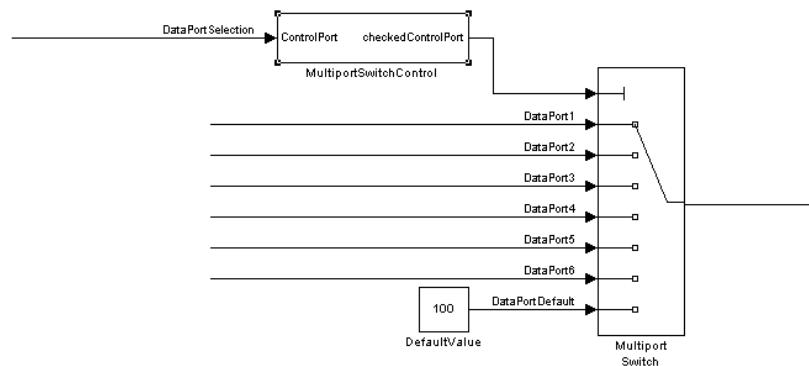
The rule makes sure that the output signal of a *Multiport Switch* block is properly defined under all circumstances.

References

- [TargetLink Production Code Generation Guide \[1\]](#), TargetLink Limitations, Block-Specific Limitations, Multiport Switch

Example

Restriction of the values of the control port of the *MultiportSwitch* block.



2.12.7 Restrictions with Regard to the Multiport Switch Block

The option *Use zero-based indexing* for the *Multiport Switch* block must not be used. This applies to TargetLink versions prior to TargetLink 2.1.

Purpose

Code generation with TargetLink.

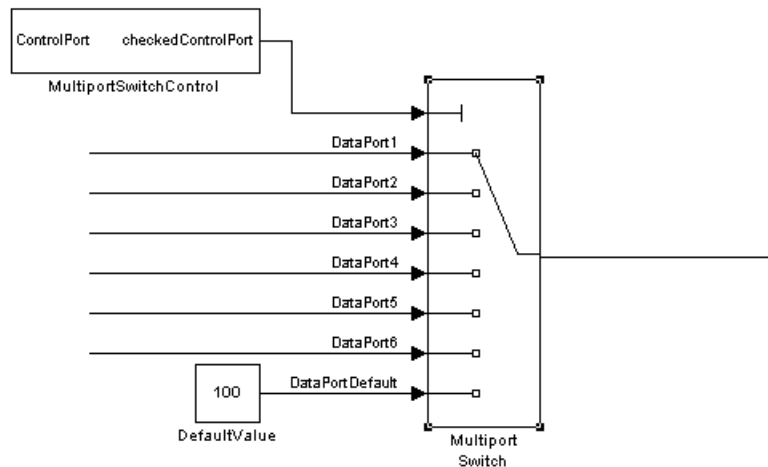
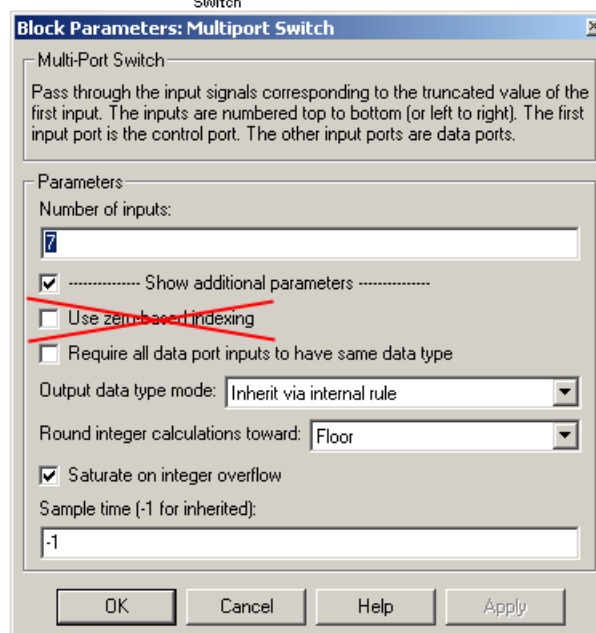
Remark

The option is not supported by TargetLink versions prior to TargetLink 2.1.

References

- [TargetLink Production Code Generation Guide \[1\]](#), TargetLink Limitations, Block-Specific Limitations, Multiport Switch

Example



2.12.8 Limitations with regard to the Bus Creator Block

The *Output as nonvirtual Bus* option of the *Bus Creator* block must not be used.

Purpose

Code generation with TargetLink.

Remark

The option serves to specify nonvirtual busses, which were introduced with Release R14 but are currently not supported by TargetLink.

Example

Fig. [27](#) shows the critical option of the *Bus Creator* block.

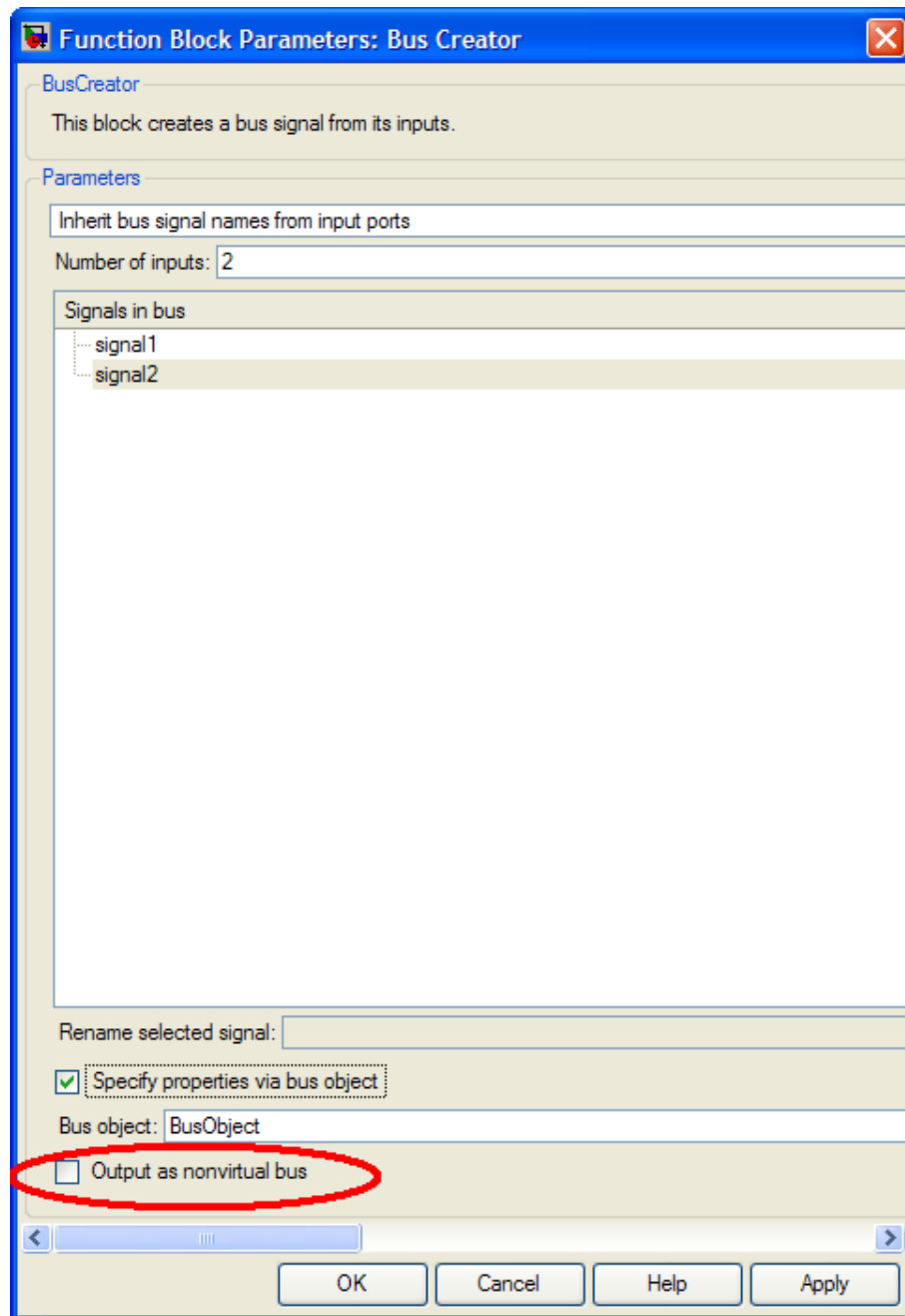


Fig. 27: Non-virtual busses, which can be generated exclusively using a *Bus Creator* block, must not be used.

2.13 Sinks Library

This section contains restrictions with regard to the use of blocks from the *Sinks* library.

2.13.1 Permissible Blocks of the Sinks Library

Only the following blocks from the *Sinks* library should be used:

- Display
- Out1
- Scope
- Terminator
- To File
- To Workspace

Purpose

Conversion to TargetLink.

Remark

TargetLink supports only a restricted subset of blocks to ensure the generation of efficient code. It should be noted that some of the supported blocks have no impact on code generation but work properly during a TargetLink model-in-the-loop simulation.

Example

Fig. 28 shows the permissible blocks.

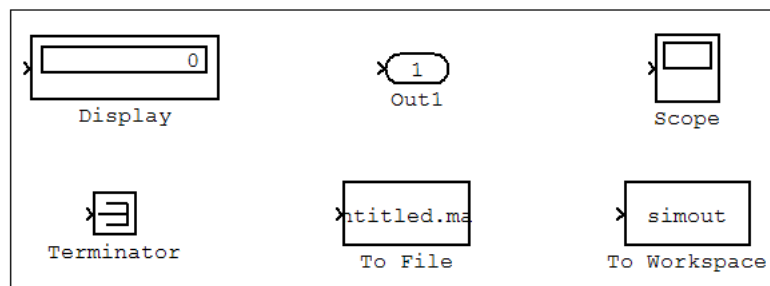


Fig. 28: Permissible blocks from the *Sinks* library.

2.14 Sources Library

This section contains restrictions with regard to the use of blocks from the *Sources* library.

2.14.1 Permissible Blocks of the Sources Library

Only the following blocks from the *Sources* library should be used:

- Constant
- Ground
- In1

Purpose

Code generation with TargetLink.

Remark

The unsupported blocks in this library are time-dependent sources which usually serve as input signals for a controller and are not part of the application algorithm itself.

Example

Fig. 29 shows the permissible blocks.

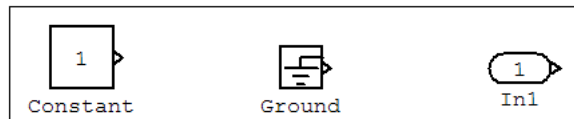


Fig. 29: Permissible blocks from the *Sources* library.

2.15 User-Defined Functions Library

This section contains restrictions with regard to the use of blocks from the *User-Defined Functions* library.

2.15.1 Permissible Blocks from the User-Defined Functions library

Only the following blocks from the *User-Defined Functions* library should be used:

- Fcn

Purpose

Code generation with TargetLink.

Remark

TargetLink supports only a restricted subset of blocks to ensure the generation of efficient code. If legacy code has to be integrated, the TargetLink *Custom Code* block should be used, as this also supports the generation of efficient production code.

Example

Fig. 30 shows the permissible blocks.

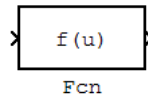


Fig. 30: Permissible blocks from the *User-Defined Functions* library.

2.15.2 Restrictions with Regard to the Fcn Block.

The *Fcn* block shall not be used in conjunction with long, complex expressions for the function to be implemented.

Purpose

Code generation with TargetLink.

Remark

Complex expressions for the function do not allow the specification of data types for intermediate variables, or any overflow or saturation handling. They should therefore be avoided.

2.16 Stateflow

This section contains restrictions with regard to modeling in Stateflow.

2.16.1 Initialization of a Stateflow Chart

The *Execute (enter) Chart At Initialization* option in the *Chart Properties* dialog must not be used.

Purpose

Code generation with TargetLink.

Remark

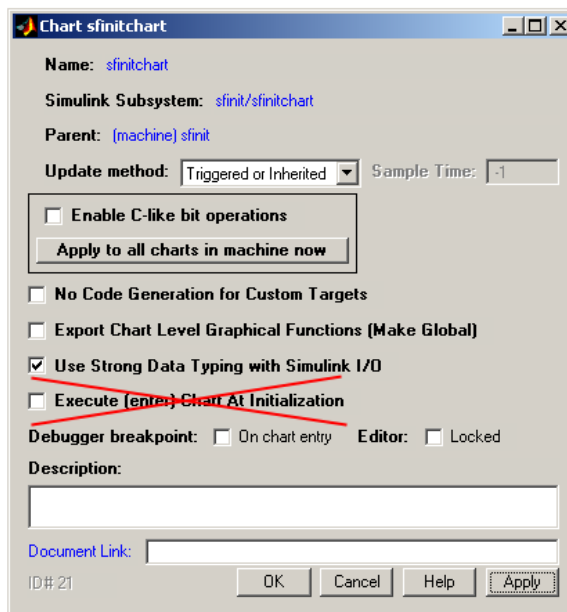
The use of this option would lead to execution of the default transition of a Stateflow chart whenever the Simulink model is initialized. TargetLink ignores this option so that default transitions are executed when the Stateflow chart is executed for the first time.

References

- [TargetLink Production Code Generation Guide \[1\]](#), TargetLink Limitations, Stateflow Limitations, Execute (enter) Chart At Initialization property

Example

Chart Properties of a Stateflow Chart.



2.16.2 Reserved Names in Stateflow

The following keywords are reserved in Stateflow and must not be used for names of variables, events, states or graphical functions in a Stateflow chart: *abs, acos, after, asin, at, atan, atan2, before, ceil, chg, change, cos, cosh, du, during, en, enter, entry, every, ex, exp, exit, fabs, floor, fmod, in, log, log10, max, min, matlab, ml, pow, send, sgn, sin, sinh, sqrt, sqrt, t, tan, tanh, tc<digits>, trigger_id*.

Purpose

Code generation with TargetLink.

Remark

The use of reserved keywords leads to erroneous code.

References

- [TargetLink Production Code Generation Guide \[1\]](#), TargetLink Limitations, Stateflow Limitations, Reserved names

2.16.3 Permitted Dimensionality at the Stateflow Chart Interface

Variables with *Input from Simulink* or *Output to Simulink* scope have to be scalar (*one-element arrays*) or vectors (*one-dimensional arrays*). Higher dimensions are not supported for the Simulink-Stateflow Interface.

Purpose

Code generation with TargetLink.

Remark

Since TargetLink does not support matrix signals in Simulink, the Simulink-Stateflow interface also has to be free of matrix signals.

Example

Inadmissible two-dimensional array with *Input from Simulink* scope in Stateflow.

Data: a

Name: a

Parent: ...signal_tl/Subsystem/Subsystem/Subsystem/Chart

Scope: Input from Simulink Port: 1

Type: double Units:

Fixed-Point

Stored Integer: int16

Fraction length: 0

☒ **Array**

Sizes: [4 4] (invalid)

First Index: 1

Limit Range

Min:

Max:

Initialize from: data dictionary 0

☐ Save final value to base workspace

☐ Watch in Debugger

Description:

Document Link:

ID# 19 OK Cancel Help Apply

2.16.4 Usage of Constants

The dimensionality of variables with *Constant* scope must be 2 at most (two-dimensional arrays).

Purpose

Code generation with TargetLink.

Remark

Arrays of three or more dimensions are not supported by TargetLink.

References

- [TargetLink Production Code Generation Guide \[1\]](#), TargetLink Limitations, Stateflow Limitations, General Stateflow Limitations

2.16.5 Permitted Data Types at the Stateflow Chart Interface

The *Use Strong Data Typing with Simulink I/O* option in the *Chart Properties* dialogs shall be checked.

Purpose

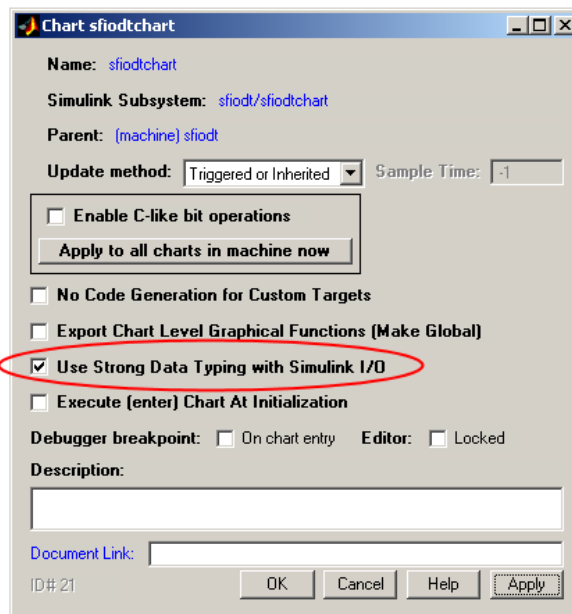
Avoidance of implicit type conversions.

Remark

Unless the option is checked, Stateflow performs implicit data type conversions at the Simulink-Stateflow interface, which may lead to data loss for variables of *Input from Simulink* or *Output to Simulink* scope.

Example

Chart Properties of a Stateflow chart.



2.16.6 Transition Actions

Transition actions generally should be avoided in favor of condition actions. Transition actions must not be used within loops.

Purpose

Code generation with TargetLink.

Remark

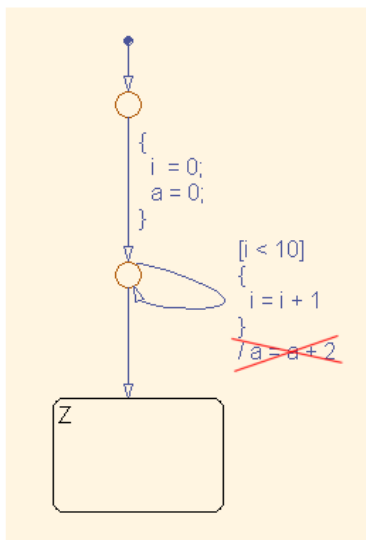
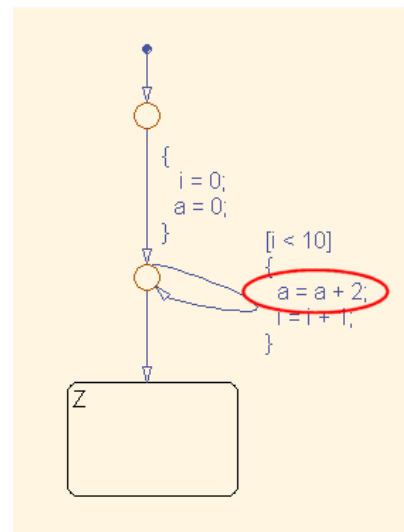
Transition actions can lead to inefficient code and should therefore be avoided. The behavior of transition actions within loops might be unexpected by many users, which is why they are not supported within loops.

References

- [TargetLink Production Code Generation Guide \[1\]](#), TargetLink Limitations, Stateflow Limitations, Transition Loops

Example

Left: Inadmissible use of a transition action. Right: Alternative realization using condition action.



2.16.7 Bind Actions

The Stateflow *Bind* feature to bind data and events to specific states must not be used in TargetLink versions prior to TargetLink 2.1.

Purpose

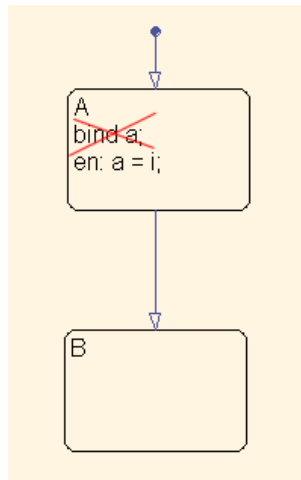
Code generation with TargetLink.

Remark

Bind actions are not supported in TargetLink versions prior to TargetLink 2.1.

Example

Inadmissible definition of a bind action.



2.16.8 Order Numbers for Parallel States

An AND state that broadcasts an event to its neighbouring states must have a higher activation order than the receivers.

Purpose

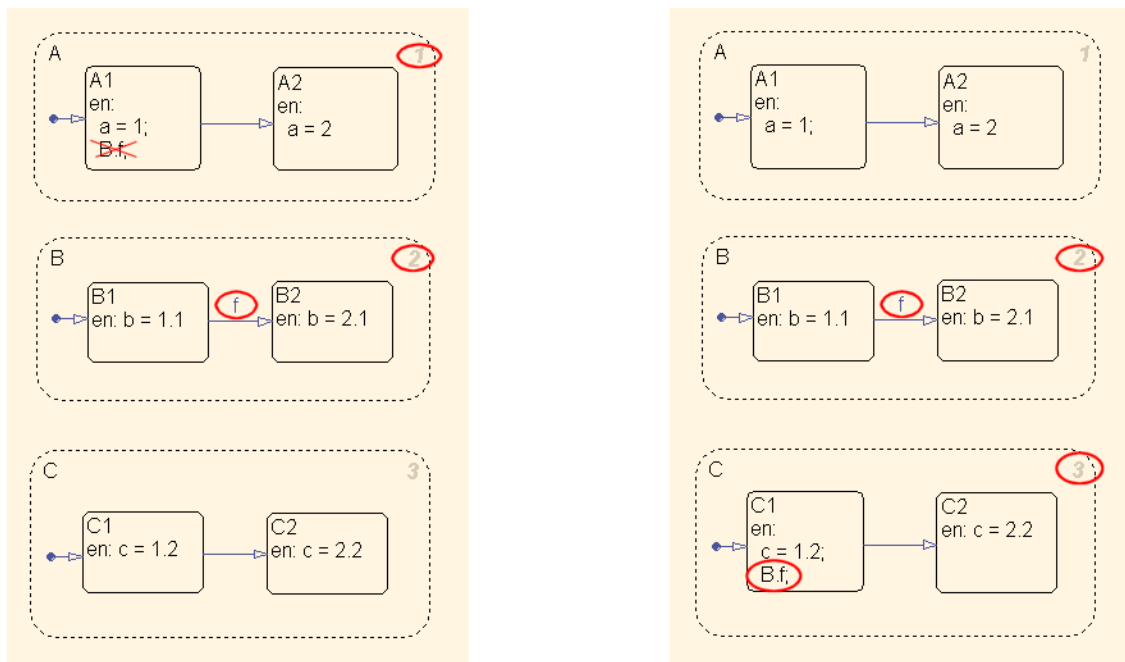
Code generation with TargetLink.

Remark

This ensures that the receivers of the event are already active at the time when they receive the event. The modeling of unreachable states and "dead code" is avoided.

Example

Left: *Directed event broadcast* with improper activation order of the states. Right: *Event broadcast* using a correct activation order.



2.16.9 Definition of Events and Variables

Events must not be defined on the *Stateflow machine level* for TargetLink versions prior to TargetLink 2.1. In addition, it is not recommended to specify data on the *Stateflow machine level*.

Purpose

Reusability of state charts and avoidance of deviations during the simulation, Code generation with TargetLink.

Remark

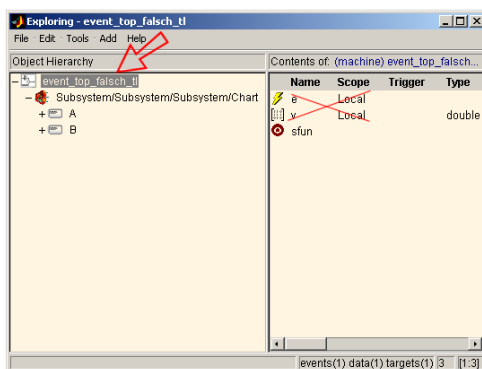
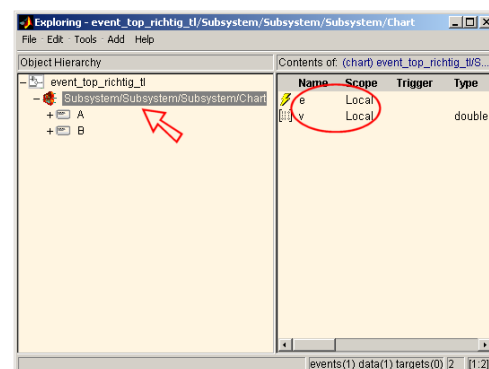
The definition of events at *Stateflow machine level* is not supported by TargetLink versions prior to TargetLink 2.1. By defining variables at the state machine level, TargetLink users risk a potential mix up. If one Stateflow chart within a TargetLink subsystem and outside it use a particular data item on the machine level, they use the identical data during MIL simulation. However, since TargetLink naturally generates a variable for machine level data, different variables are used during SIL simulation, which might lead to simulation differences.

References

- [TargetLink Production Code Generation Guide \[1\]](#), TargetLink Limitations, Stateflow Limitations, General Stateflow Limitations

Example

Left: Inadmissible definition of a variable and an event at *Stateflow machine level*. Right: Proper definition of variables and events using the *Stateflow chart level*.



2.16.10 No Use of Implicit Events

Implicit events (*change*, *enter*, *exit*) should not be used.

Purpose

Design of transparent models in Stateflow.

Remark

While the receivers of implicit event broadcast are defined through event specifiers (*change*, *chg*, *enter*, *en*, *exit*, *ex*) there is no such specification with regard to the origin of the implicit event. The usage of implicit events is therefore intransparent concerning potential side effects of variable assignments or the entering/exiting of states.

2.16.11 MATLAB Functionality

The MATLAB functionality

- *MATLAB functions*
- *MATLAB actions*
- time t for the absolute simulation time

must not be used within Stateflow.

Purpose

Code generation with TargetLink.

Remark

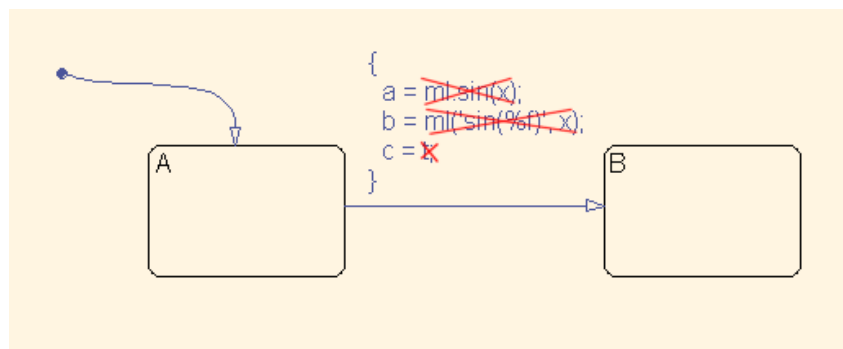
The MATLAB functionality listed above is not supported by TargetLink for reasons of code efficiency.

References

- [TargetLink Production Code Generation Guide \[1\]](#), TargetLink Limitations, Stateflow Limitations, Embedding MATLAB/Simulink functionality

Example

Inadmissible use of MATLAB functionality within Stateflow.



2.16.12 Alias Types

The usage of user-defined names for data types in Stateflow using Simulink *Alias Type* objects is not permitted.

Purpose

Code generation with TargetLink.

Remark

This feature, which was newly introduced with R14 SP2, is currently not supported by TargetLink.

Example

Fig. 31 and 32 show the impermissible use of *Alias Type* objects for data types.

2.16.13 Execution Order of Transitions

The Stateflow feature *User specified transition execution order* must not be used in conjunction with TargetLink versions prior to TargetLink 2.0.7.

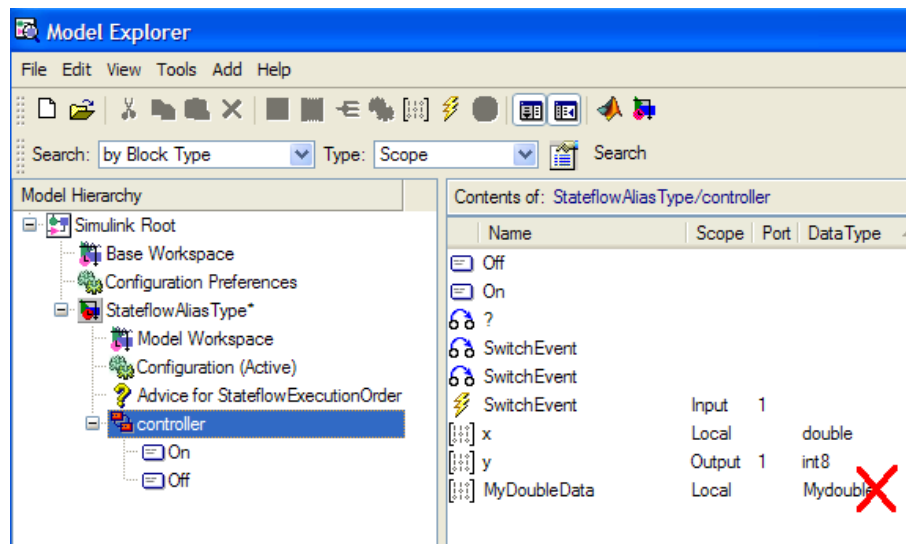


Fig. 31: *Alias Type* objects, which can be defined on the model level, must not be used to introduce user-defined names for data types in Stateflow. Here, an *Alias Type* named `Mydouble` is used to specify the data type of the `MyDoubleData` variable, which is not supported by TargetLink.

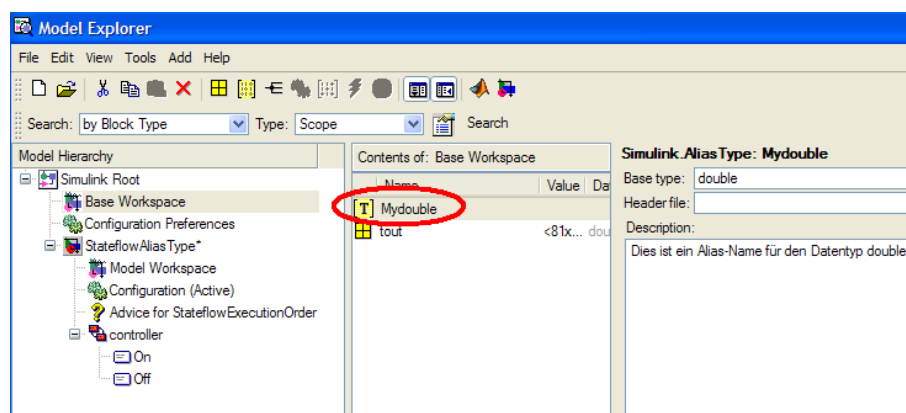


Fig. 32: The definition of an *Alias Type* object called `Mydouble` in the workspace.

Purpose

Code generation with TargetLink.

Remark

This feature, which was newly introduced with R14 SP2, is supported only for TL 2.0.7 and succeeding versions.

Example

Fig. 33 shows the *User specified transition execution order* feature in the *chart properties* dialog in Stateflow. In the default case, the option is not activated.

2.16.14 Arrays at Interfaces of Graphical Functions

Graphical Functions can only be used in conjunction with scalar input and output signals. Vectors and matrices are not supported as interface variables.

Purpose

Code generation with TargetLink.

Remark

This feature, which was newly introduced with R14, is currently not supported by TargetLink. However vector and matrix signals can be used as local variables in Graphical Functions.

Example

Fig. 35 shows the specification of input and output signals for a Graphical Function, see fig. 36.

2.16.15 Cast Operator for Data Type Conversions

The conversion of data types through means of the *cast*-operator in Stateflow's action language must not be used in conjunction with TargetLink versions prior to TargetLink 2.1.

Purpose

Code generation with TargetLink.

Remark

This feature, which was newly introduced with R14, is supported by TargetLink only since version 2.1. However, type casting in the original form can be used in all TargetLink versions.

Example

Fig. 37 and 38 show the use of the Stateflow *cast* operator, which is permitted since TargetLink 2.1.

2.16.16 Data on the Level of Stateflow Boxes

Data must not be defined on the level of Stateflow boxes. Instead the hierarchy level above can be used.

Purpose

Code generation with TargetLink.

Remark

This feature, which was newly introduced with Release R14, is currently not supported by TargetLink.

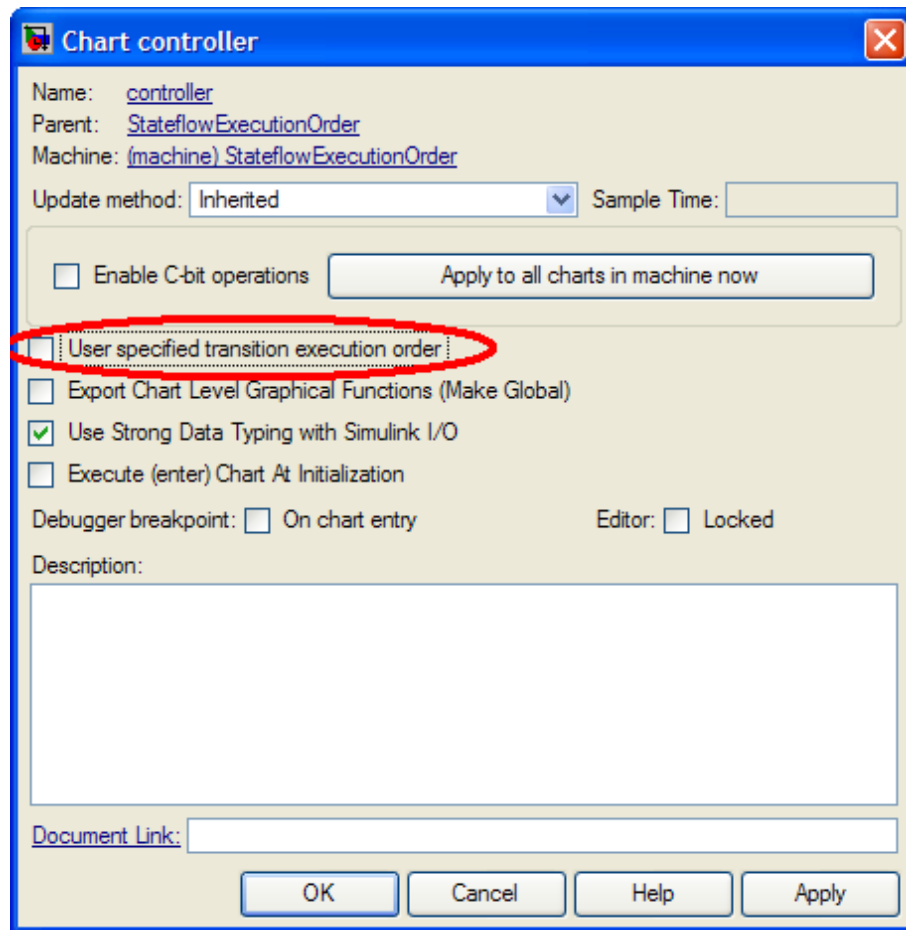


Fig. 33: By checking the *User specified transition execution order* feature, each transition can be assigned a value specifying the order in which it is checked for execution with regard to other potential transitions. The feature can only be used in conjunction with TargetLink 2.0.7 and succeeding versions.

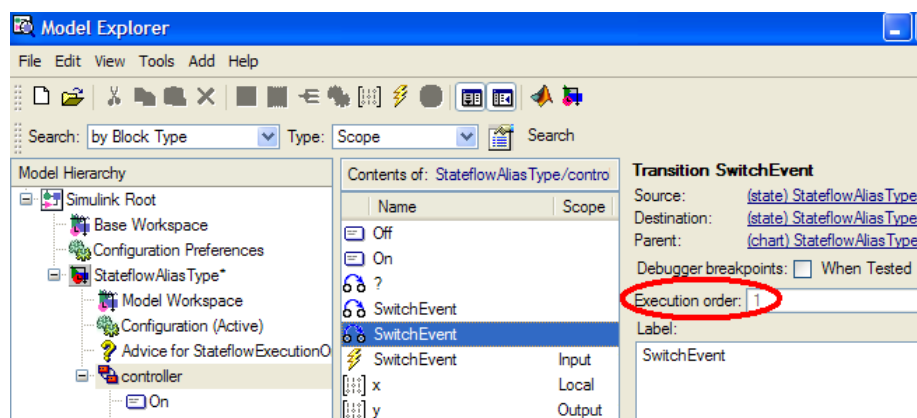


Fig. 34: The assignment of an execution order for each transition. The option is disabled, if the *User specified transition execution order* option is not checked.

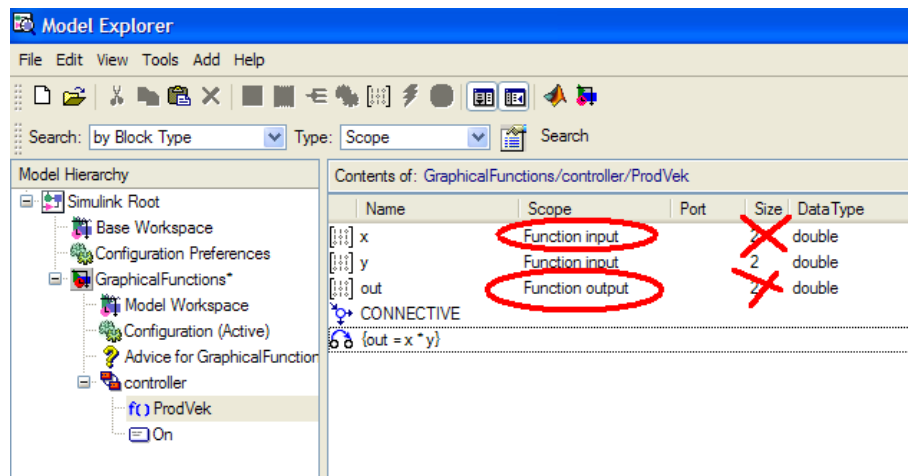


Fig. 35: Example of a Graphical Function called `ProdVek` with impermissible input and output signals (Scope *Function input* and *Function output*) with a *Size* of 2.

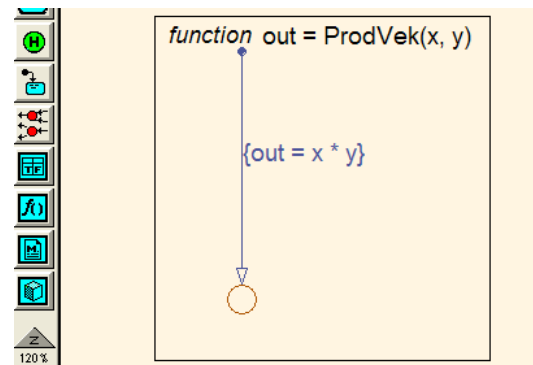


Fig. 36: Graphical Function called `ProdVek`.

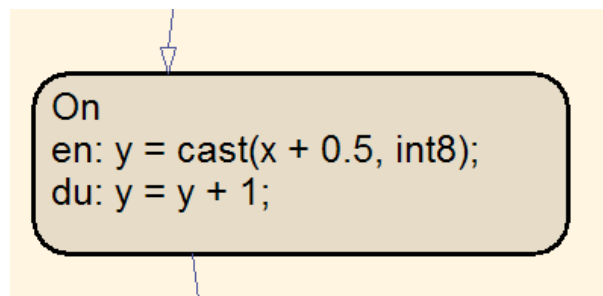


Fig. 37: The `cast` operator can be used in TargetLink 2.1 and succeeding versions to perform a typecast in Stateflow's action language. For other TargetLink versions, a typecast can be carried out explicitly in the usual manner, here for instance, `int8(x+0.5)`.

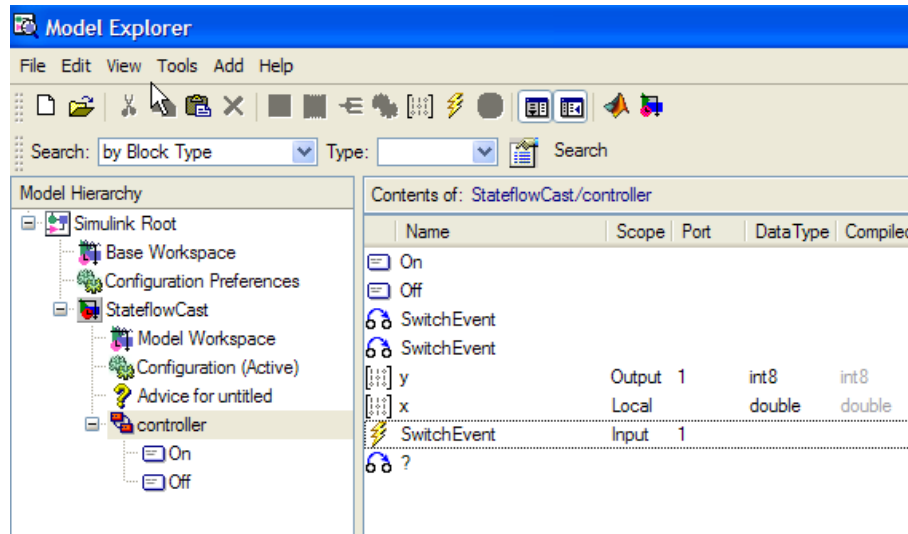


Fig. 38: Through the *cast* operator, explicit data type conversions can be specified, here involving the Stateflow variables x and y .

Example

Fig. 39 and 40 show the impermissible definition of data on the level of Stateflow boxes.

2.16.17 Embedded MATLAB Functionality

The *Embedded MATLAB* functionality in Stateflow must not be used.

Purpose

Code generation with TargetLink.

Remark

TargetLink serves to generate highly efficient production code. This is why only a limited subset of the functionality in Simulink/Stateflow is supported. The *Embedded MATLAB* functionality was introduced with Release R14 and is not supported by TargetLink.

Example

Fig. 41 shows the *Embedded MATLAB* functionality in Stateflow.

2.16.18 Complex Expressions for the Size Property

Complex expressions must not be used for the *Size* property in Stateflow.

Purpose

Code generation with TargetLink.

Remark

This feature, which was newly introduced with Release R14, is currently not supported by TargetLink.

Example

Fig. 42 shows an example for the improper specification of the *Size* property.

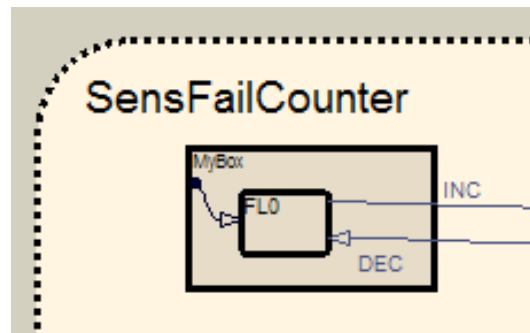


Fig. 39: A box called *MyBox* in Stateflow.

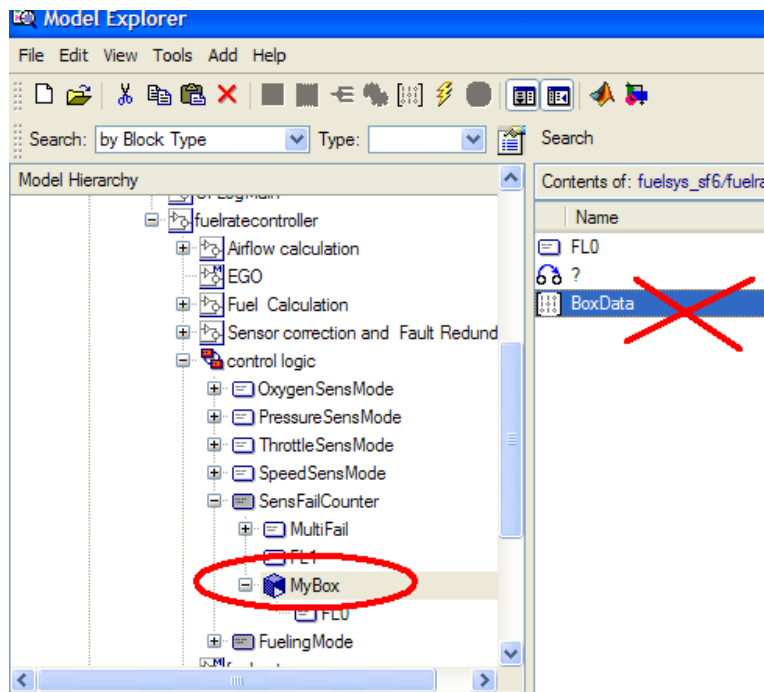


Fig. 40: The *BoxData* data item is specified on the hierarchy level of the *MyBox* box in Stateflow, which is not supported by TargetLink.

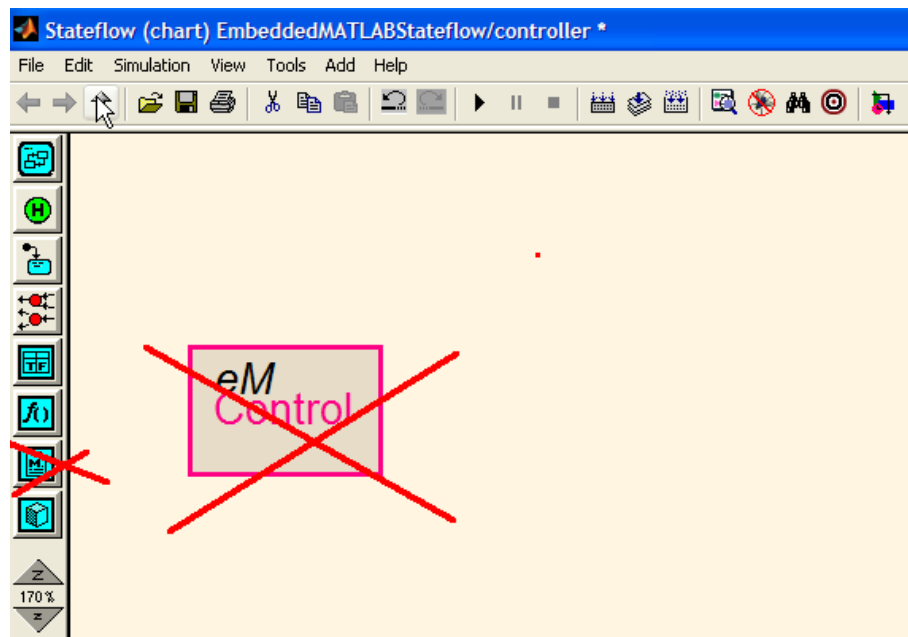


Fig. 41: The *Embedded MATLAB* modeling functionality must not be used.

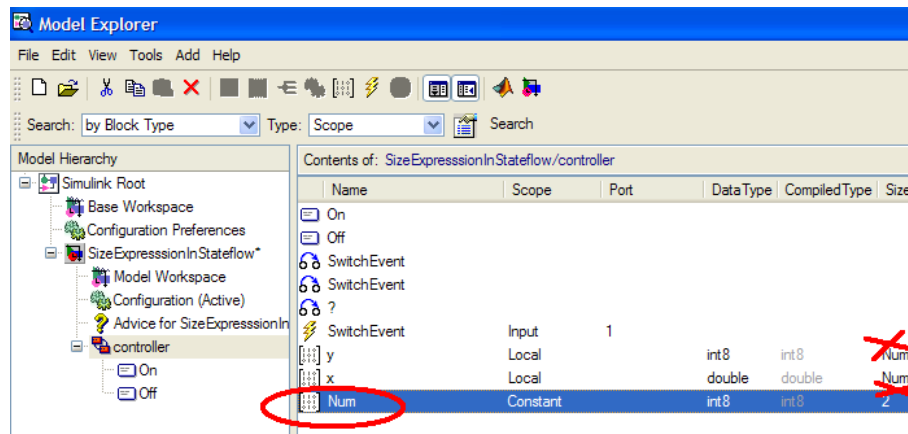


Fig. 42: The *Size* property for the Stateflow variables *x* and *y* is specified through another Stateflow constant called *Num*, which is currently not supported by TargetLink.

2.16.19 Simulink-Stateflow Inheritance

The inheritance mechanism for the *Size* and *Data Type* properties at the Simulink-Stateflow interface must not be used.

Purpose

Code generation with TargetLink.

Remark

This feature, which was newly introduced with Release R14, is currently not supported by TargetLink.

Example

Fig. 43 shows an example of the impermissible use of property inheritance for a Stateflow variable at the Simulink-Stateflow interface.

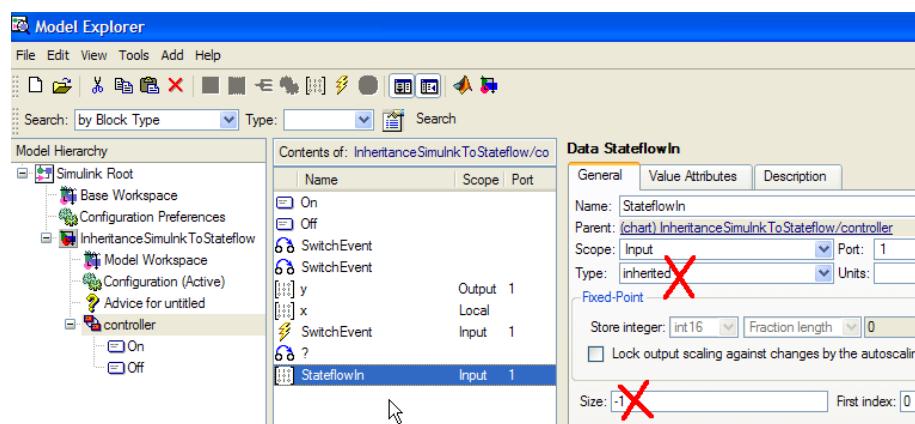


Fig. 43: The specification of -1 for the *Size* property or *inherited* for the *Type* property must not be used in conjunction with TargetLink.

3 Model Parameters

This section contains a few recommendations with regard to the handling of model parameters. This particularly applies to application parameters, which need to be calibratable.

3.1 Parameter Initialization

All application parameters and other parameters required for model initialization shall be properly defined, for instance via MATLAB M-files or the dSPACE Data Dictionary.

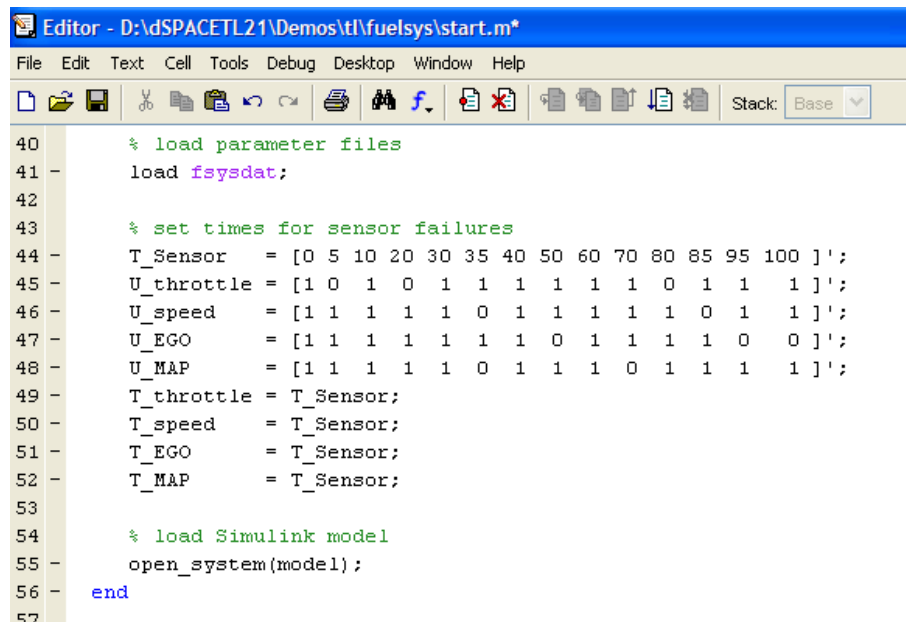
Purpose

Correct model initialization.

Remark

Parameters in Simulink blocks must be properly initialized. Handling the parameters in M-files is a common way of doing this. As an alternative, the dSPACE Data Dictionary can not only be used to specify parameter values but also code generation settings.

Example



```

Editor - D:\dSPACE\TL21\Demos\tl\fuelsys\start.m*
File Edit Text Cell Tools Debug Desktop Window Help
[Icons] Stack: Base
40      % load parameter files
41      load fsysdat;
42
43      % set times for sensor failures
44      T_Sensor = [0 5 10 20 30 35 40 50 60 70 80 85 95 100]';
45      U_throttle = [1 0 1 0 1 1 1 1 1 1 0 1 1 1]';
46      U_speed = [1 1 1 1 1 0 1 1 1 1 1 0 1 1]';
47      U_EGO = [1 1 1 1 1 1 1 0 1 1 1 1 0 0]';
48      U_MAP = [1 1 1 1 1 0 1 1 1 0 1 1 1 1]';
49      T_throttle = T_Sensor;
50      T_speed = T_Sensor;
51      T_EGO = T_Sensor;
52      T_MAP = T_Sensor;
53
54      % load Simulink model
55      open_system(model);
56  end
57

```

Fig. 44: Use of an M-file for initializing block parameters in Simulink models.

3.2 Definition/Display of Block Parameters

Block parameters in Simulink models should be specified without any algebraic expressions and preferably without data type conversions. In addition, it is recommended to indicate the presence of application parameters in a block by using a special color scheme, for instance using an orange background.

Purpose

Initialization of block parameters, proper graphical display.

Remark

Model reviews are simplified and the presence of application parameters is more transparent. The avoidance of complex expressions for application parameters ensures a proper calibration process later on.

Example

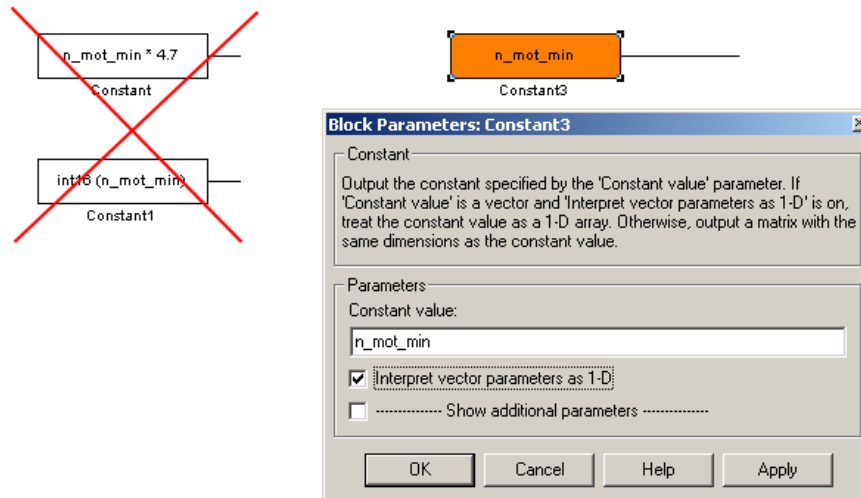


Fig. 45: The presence of application parameters in individual blocks should be indicated, for instance by using a special color scheme.

4 Name Spaces

In order to support a namespace for MATLAB/Simulink/Stateflow designators, it is useful to apply naming conventions for all identifiers. This chapter provides some recommendations for such naming conventions.

4.1 General Naming Convention

Identifiers within a model (in particular for parameters, subsystem names and port names) should be valid ANSI C identifiers, i.e. they should contain the letters a - z, A - Z, the digits 0 - 9 and an underscore. The first character has to be a letter. Names should be unique within the first 31 characters and different from the following reserved names: *auto*, *break*, *case*, *catch*, *char*, *const*, *continue*, *default*, *do*, *double*, *else*, *elseif*, *end*, *enum*, *extern*, *float*, *for*, *function*, *global*, *goto*, *if*, *int*, *long*, *otherwise*, *persistent*, *register*, *return*, *short*, *signed*, *sizeof*, *static*, *struct*, *switch*, *try*, *typedef*, *union*, *unsigned*, *void*, *volatile*, *while*.

Purpose

Naming convention, avoiding naming conflicts.

Remark

The rule enables seamless conversion and code generation for Simulink models with TargetLink. The generated code then contains only valid C identifiers that avoid name collisions.

References

- [TargetLink Production Code Generation Guide \[1\]](#), TargetLink Limitations, General Limitations, Reserved identifiers

4.2 Reserved System Headerfile Names

Subsystems names should be different from TargetLink system header files like *tl_types*, *tl_target_types*, *tllimits*, *dsfxp_a*, *dsfxp_c* or any other compiler or standard library.

Purpose

Naming convention, avoiding naming conflicts, compliance with MISRA rule115: Standard library function names shall not be reused.

Remark

This rule is required to ensure the correctness of the generated code.

References

- [TargetLink Production Code Generation Guide \[1\]](#), TargetLink Limitations, General Limitations, Reserved Identifiers

5 Comments and Units

The guidelines contained in this chapter should be regarded as examples of how model parts can be suitably commented. This ensures proper documentation of models and simplifies reviews and maintenance.

5.1 Comments on the Top Level

A comment should be provided on the top level for each Simulink model and Simulink library, describing its functionality. Additional comments can be placed at proper locations within the model if that makes the model more understandable.

Purpose

Proper documentation.

Remark

Comments serve to simplify reviews and maintenance, and increase the comprehensibility of the model.

Example

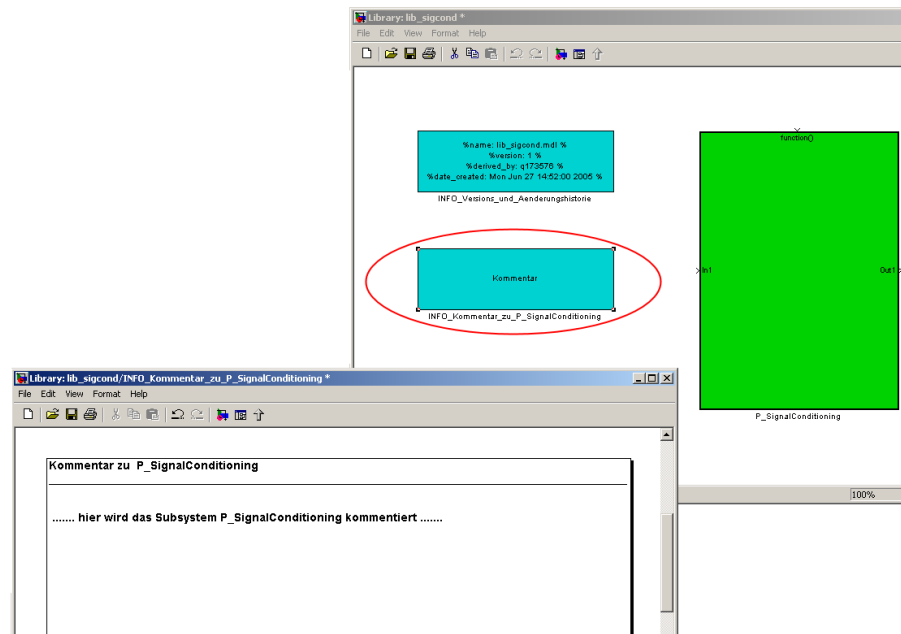


Fig. 46: Proper model comments.

5.2 Comments in Block Properties

All comments provided should be valid ANSI C comments without the enclosing `/*` and `*/` symbols. In particular, this rules out the use of `/*` and `*/` as well as umlaut and special characters.

Purpose

Subsequent code generation with TargetLink.

Remark

The rule ensures that comments on the block level can be transformed into valid C comments.

5.3 Comments for Interfaces

For easier understanding, information about the interfaces of individual subsystems can be described in the *Description* field of the *Block Properties* for *Inports* and *Outports*.

Purpose

Proper interface and data documentation.

Remark

The contents in the *Description* field help to simplify reviews and maintainability.

Example

Fig. 47 shows an example of proper interface description.

5.4 Comments for Individual Blocks and Signals

For easier understanding, additional comments regarding blocks and signals can be placed in the *Description* field of *Block Properties* or the *Code comment* and *Block comment* fields of TargetLink blocks, if they are being used.

Purpose

Proper documentation.

Remark

The contents in the *Description* field helps to simplify reviews and maintainability.

Example

Fig. 48 shows an example of proper signal description.

5.5 Units for Block Parameters and Signals

The units of block parameters and signals should be indicated in the *Description* field of the *Block Properties*.

Purpose

Proper documentation.

Remark

The specification of units helps to analyze the functionality of the model.

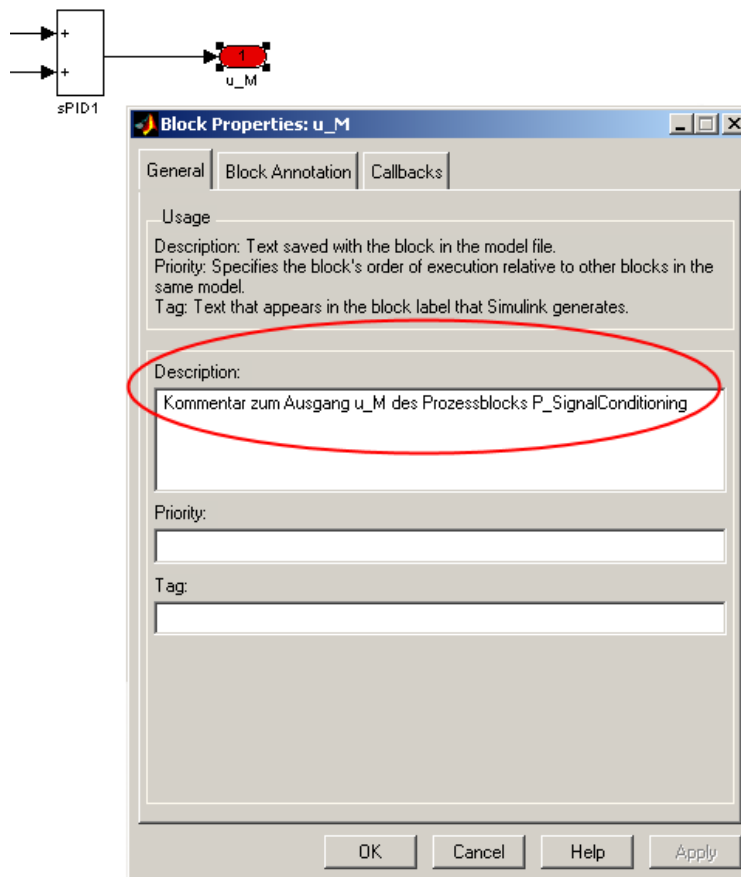


Fig. 47: Proper interface comments for an *Outport* block.

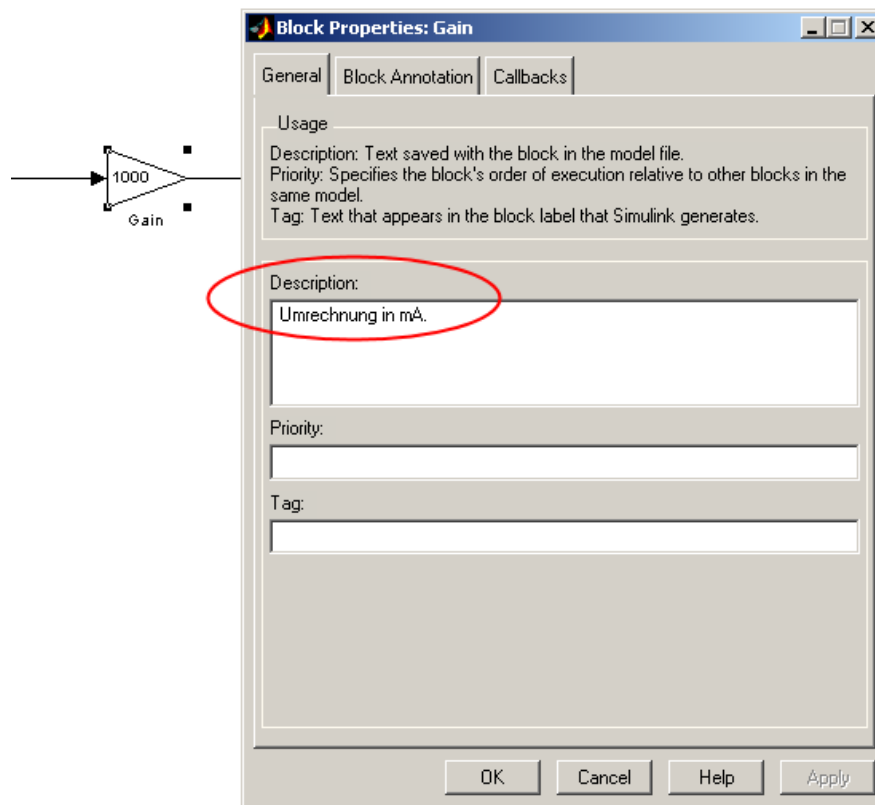


Fig. 48: Proper block commenting, e.g. for a *Gain* block.

6 Layout Issues

The guidelines contained in this chapter should be regarded as recommendations for layout issues. The rules deal with basic layout questions to help establish a common standard which guarantees simple model reviews and transparent models.

6.1 Signal Flow

Inports should be located on the left, *Outports* on the right. Signal flow is usually directed from left to right with the exception of feedback loops. The signal flow within loops should have clockwise orientation.

Purpose

Establishing a transparent, standardized layout.

Remark

The guideline helps to simplify reviews and maintenance.

6.2 Signal Lines

Signal lines should have a rectangular shape whenever possible and should not cross blocks or sub-systems. The number of intersections should be kept as low as possible.

Purpose

Establishing a transparent, standardized layout.

Remark

The guideline helps to simplify reviews and maintenance.

References

- [Controller Style Guidelines for Production Intent Using MATLAB, Simulink and Stateflow V1.0.0 \[8\], 4.3.1.3](#)

6.3 Display of Signals and Busses

Signals and busses should be provided with a name at their origin or other suitable locations to indicate names and corresponding signals.

Purpose

Establishing a transparent, standardized layout.

Remark

The guideline helps to simplify reviews, maintenance and further processes.

References

- [Controller Style Guidelines for Production Intent Using MATLAB, Simulink and Stateflow V1.0.0 \[8\], 4.6.1.1](#)
- [Controller Style Guidelines for Production Intent Using MATLAB, Simulink and Stateflow V1.0.0 \[8\], 4.6.1.2](#)
- [Controller Style Guidelines for Production Intent Using MATLAB, Simulink and Stateflow V1.0.0 \[8\], 4.6.1.3](#)

6.4 Nonscalar Signals

The option *Wide nonscalar lines* in the *Format* menu should be used to indicate the dimensionality of individual signals.

Purpose

Establishing a transparent, standardized layout.

Remark

The guideline helps to simplify reviews and maintenance.

References

- [Controller Style Guidelines for Production Intent Using MATLAB, Simulink and Stateflow V1.0.0 \[8\], 4.3.2.1.1](#)

6.5 Display of Blocks

Block symbols should be large enough so that all icons and labels are clearly visible. Important block parameters should be displayed via *Attribute Format String* or *Block Annotation*. Block names should be located below the blocks.

Purpose

Establishing a transparent, standardized layout.

Remark

The guideline helps to simplify reviews and maintenance.

References

- [Controller Style Guidelines for Production Intent Using MATLAB, Simulink and Stateflow V1.0.0 \[8\], 4.5.1.3](#)
- [Controller Style Guidelines for Production Intent Using MATLAB, Simulink and Stateflow V1.0.0 \[8\], 4.5.1.1](#)
- [Controller Style Guidelines for Production Intent Using MATLAB, Simulink and Stateflow V1.0.0 \[8\], 4.3.1.7](#)

Example

Fig. 49 shows some important block parameters that might be worth displaying.

6.6 Font and Font Sizes

All elements within a model except for *text annotations* should be displayed in the same font and size. Common fonts like for instance *Arial* should be used.

Purpose

Establishing a transparent, standardized layout.

Remark

The guideline helps to simplify reviews and maintenance.

References

- [Controller Style Guidelines for Production Intent Using MATLAB, Simulink and Stateflow V1.0.0 \[8\], 4.3.1.1](#)

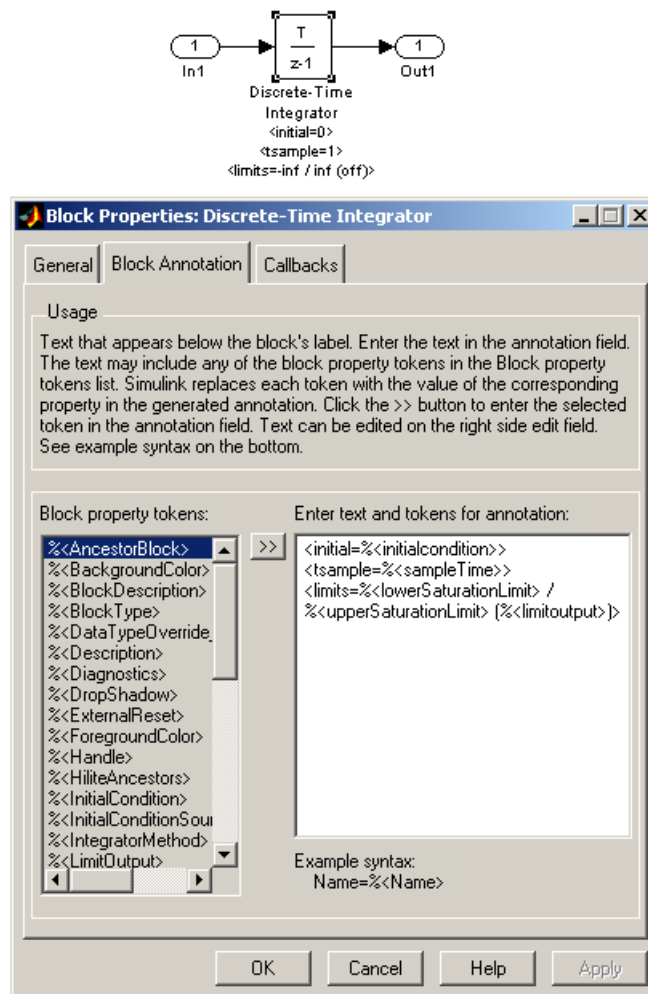


Fig. 49:

7 Simulation Parameters

A correct simulation in Simulink requires suitable simulation parameter settings for the individual model (*Configuration Parameters* in the *Simulation* menu). Although the default settings may be adequate in some cases, model-specific adjustments might be necessary to produce sufficient simulation results. The proposed simulation parameter settings described in this chapter should be regarded as recommendations only, and deviations might be appropriate for specific models.

7.1 Solver Settings

The *Solver options* should be set to *Type: Fixed-step* and *Mode: SingleTasking*. The integration algorithms *ode1 (Euler)*, *ode2 (Heun)*, *ode3 (Bogacki-Shampine)*, *ode4 (Runge-Kutta)*, *ode5 (Dormand-Prince)* and the *Fixed step size* should be set in accordance with the required accuracy.

Purpose

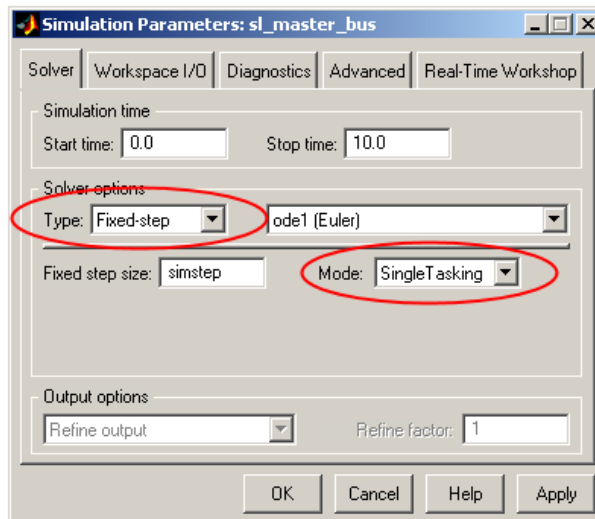
Proper model simulation.

Remark

The *Type: Fixed-step* option causes the simulation to be performed with the fundamental *Fixed step size*. The *Mode: SingleTasking* option ensures proper simulation of multirate models without the introduction of *Rate Transition* blocks. If plant models require simulation with adaptive step sizes, the solver option *Type: Variable-step* has to be used.

Example

Setting of the Simulation Parameters/Solver:



7.2 Diagnostics Settings

The *simulation options* for *Consistency checking* and *Bounds checking* should be set to *none*. In addition, the following settings should be chosen for the *Configuration options*:

- Algebraic loop: Error
- Block priority violation: Error
- SingleTask rate transition: Warning

- Check for singular matrix: Warning
- Underspecified data types: Warning
- Unneeded type conversion: Warning
- Vector/Matrix conversions: Warning
- Signal label mismatch: Warning
- Unconnected block input: Error
- Unconnected block output: Error
- Unconnected line: Error

Purpose

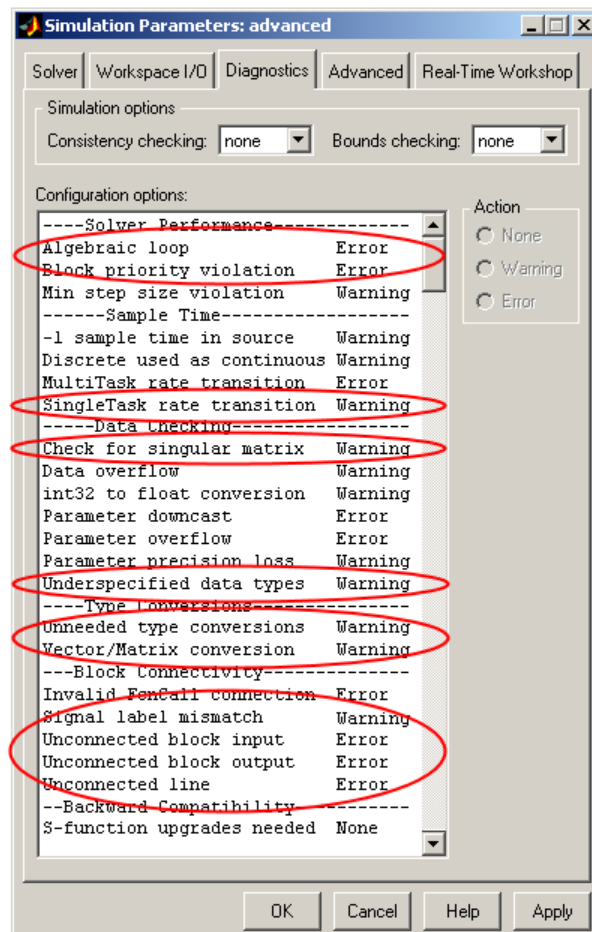
Proper diagnostics functionality.

Remark

The *Consistency checking* and *Bounds checking* options are used to validate user-written *S-functions* and might significantly increase the simulation time. The selected *Configuration options* serve to signal warnings and errors in order to prevent functional modeling errors.

Example

Settings for Simulation Parameters/Diagnostics. Deviations from the Simulink default settings are marked.



7.3 Advanced Settings

For the *Model parameter configuration*, the *Inline parameters* option should be deactivated. On the *Optimizations* tab, the following settings should be chosen:

- Block reduction: Off
- Boolean logic signals: On
- Conditional input branch: On
- Parameter pooling: On
- Signal storage reuse: Off
- Zero-crossing detection: On

Purpose

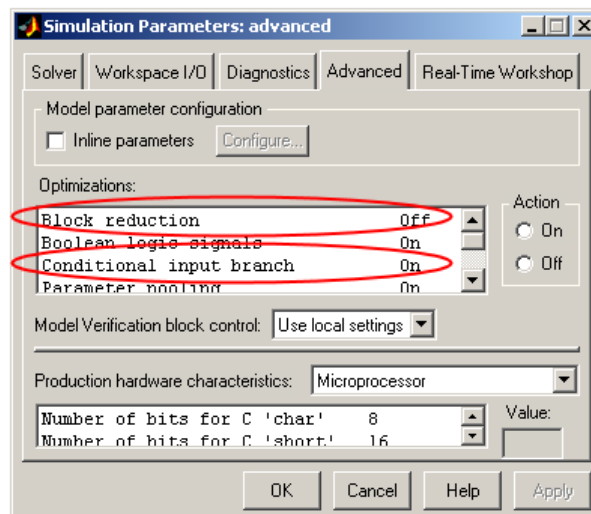
Proper model simulation.

Remark

The deactivation of the *Inline parameters* option enables run-time parameter modification during the rapid prototyping phase. The activation of *Conditional input branch: On* reduces the simulation time since calculations are not performed for deactivated paths for *Switch* and *Multiport Switch* blocks.

Example

Settings for Simulation Parameters/Advanced. The deviations from the Simulink default settings are marked.



8 Data Types

With regard to the use of data types, it is important to distinguish between two different phases:

- The simulation on the block diagram level in Simulink when the actual control algorithm is designed and tested using model-in-the-loop (MIL) simulation.
- The code level, where implementation specific details like fixed-point scalings are added and taken into consideration. The software-in-the-loop (SIL) simulation is the proper place to test the specifics of the implementation of the algorithm.

The actual development of the algorithm and the MIL-simulation should be devoid of implementation specific characteristics. This has important implications with regard to the use of data types in a MIL-simulation because saturation and overflows in integer operations are highly implementation specific rather than part of the control algorithm.

As a rule of thumb, the following principles should be upheld for the model-in-the-loop simulation: All signals which are covering a continuous-valued scale by nature have to be modeled in the form of floating point values with maximum accuracy, i.e. with the *double* data type. The fact that those signals are quantized for digital signal processing in practice and implemented in the form of fixed-point signals in the generated code shall have no impact for the control algorithm development phase and the MIL simulation. Implementation specific information shall only be added during the code generation phase, while the MIL simulation shall be performed with maximum accuracy.

Signals which are discrete by nature having only a definite number of potential values can be modeled using integer data types also (see below for restrictions with regard to the blockset stand-alone). Thereby, it has to be made sure that the limited data range does not lead to problems with saturation, potential overflows etc. following the principle that implementation specific data shall have an impact on the generated code only. Saturation and overflows are extremely data type-dependend and their exact semantics in Simulink are not properly defined.

Fixed-point data types shall be used on the code level exclusively, not in a model-in-the-loop simulation.

Note: The use of integer data types is significantly restricted if the TargetLink stand-alone blockset is used because the *Cast output to TargetLink type* flag is mostly ignored in this mode.

With regard to the use of non-double data types in general, one has to be very careful not to use a modeling style with poorly defined semantics in Simulink like saturation in integer operations, overflows, rounding in integer operations etc. The figures 50 and 51 demonstrate unexpected results which occur if integer arithmetic is used in conjunction with integer saturation in Simulink. This semantic is not emulated by TargetLink which performs saturation on the final output only, thereby producing the "expected results". The figures 52 and 53 demonstrate how block-specific and intransparent the semantics in Simulink are, if the *round integer calculation towards* option is used. Such a modeling style should not be used and the behaviour is not emulated by TargetLink.

8.1 General Use of Data Types

All signals which are continuous by nature should be modeled using the *double* data type on the block level. Information with regard to a later fixed-point implementation should be added during the code generation phase later on, not on the Simulink block level.

Signals which are discrete by nature can be modeled using integer data types like *int8*, *int16* etc. Here, it has to be taken care that the range is sufficient to make sure that the integer arithmetic is devoid of any overflows, integer saturations etc. Those are implementation specific characteristics which shall be handled during the target implementation phase.

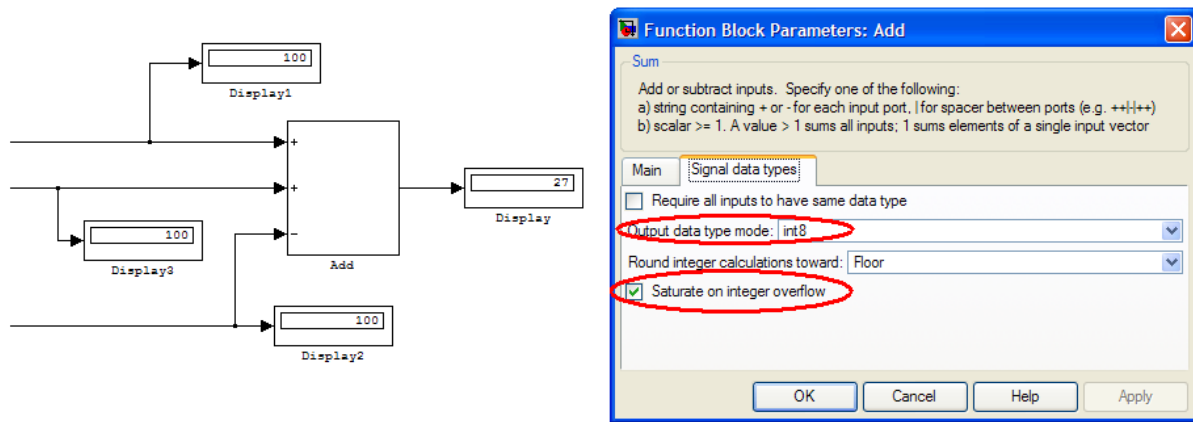


Fig. 50: Saturated addition using integer arithmetic: The output of the sum block is specified as *int8* and the *saturate on integer overflow* option is set. Note that not the assumed result $100 + 100 - 100 = 100$, which perfectly fits into an *int8* variable, is calculated but 27. This is a consequence of the saturation of intermediate results in Simulink which leads to a somehow unexpected result. Such a modeling style must not be used.

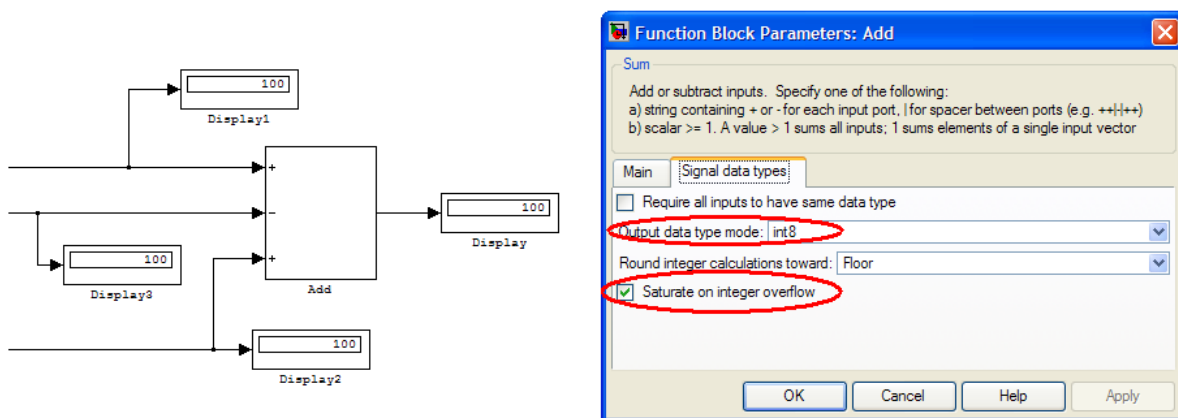


Fig. 51: Saturated addition with swapped input signals compared to figure 50. Now the expected result 100 is calculated because no saturation of intermediate results occurs. Hence, the addition is far from being commutative and such a modeling style must not be used.

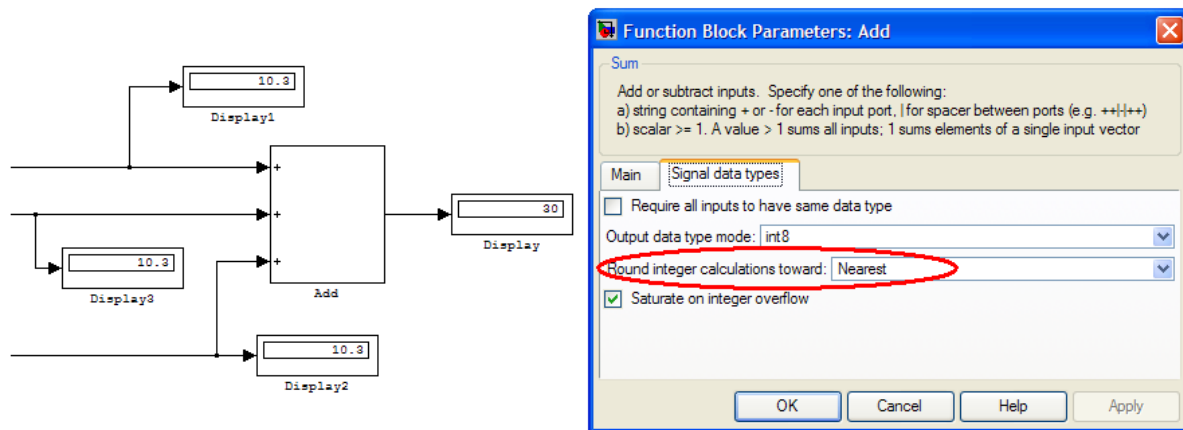


Fig. 52: Integer addition with rounding: Since an integer data type is specified for the output and input signals are non-integer, a rounding operation must be performed. Although the *round integer calculation towards* option is set to *Nearest*, Simulink calculates 30 as result. This is due to the fact that the input signals of the *Sum* block are directly cast to the output type applying the specified rounding functionality. This behaviour is highly block-specific.

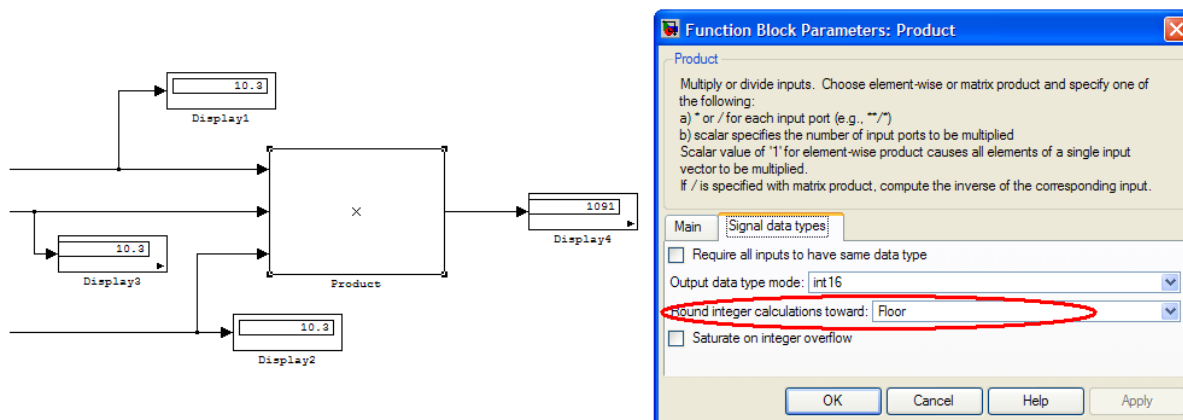


Fig. 53: Integer Multiplication with rounding. As opposed to the *Sum* block in figure 52, the rounding operation is only performed on intermediate results, not on the input signals directly. As a consequence Simulink's calculated result does not equal 1000. Because this behaviour is highly block-specific and intransparent, a modeling style requiring such operations should be avoided.

Fixed-point data types must not be used for simulations in Simulink on the block diagram level (model-in-the-loop simulation) at all. They should be used for software-in-the-loop simulations (code level) exclusively.

Logical signals can be modeled using the data type *boolean*.

Note: The use of integer data types is significantly restricted if the TargetLink stand-alone blockset is used because the *Cast output to TargetLink type* flag is mostly ignored in this mode.

Purpose

Proper modeling style and a distinction between control algorithm development and its final implementation, equal results for Simulink and TargetLink simulation modes.

Remark

Data types during the function development phase should be free from implementation specific details. The use of fixed-point data types is highly implementation specific and furthermore it is intransparent, how exactly the fixed-point operations are carried out with regard to intermediate results. These details should be addressed during the implementation/coding phase.

8.2 Data Types for Relational and Logical Operator

For the *Relational Operator* and *Logical Operator* blocks, the *boolean*, *logical* (see *Configuration Parameters:Optimization*) or *double* data types must be used. Furthermore, the settings should be consistent for the blocks, meaning, they should be either set to *boolean*, *double* or *logical* for all *Relational Operator* and *Logical Operator* blocks.

Purpose

Identical data types for TargetLink and Simulink.

Remark

The output data types of the TargetLink *Relational Operator* and *Logical Operator* blocks are either *double* or *boolean* dependent on the global Simulink option *Implement logical signals as boolean (vs. double)* in the *Simulation/Configuration Parameters/Optimization* dialog. The behaviour is therefore identical to the *logical* option in Simulink. In order to ensure equal data types in Simulink and TargetLink, the options above have to be specified consistently for all blocks. If necessary, the global Simulink option *Implement logical signals as boolean (vs. double)* has to be modified also. If succeeding blocks shall be driven by a different data type, then the output signals of *Relational Operator* and *Logical Operator* blocks have to be cast to the proper type using a *Data Type Conversion* block.

Example

The figures 54 and 55 show the inadmissible and proper use of data types.

8.3 Data Types for Signal Busses at the TargetLink-Simulink Interface

Bus signals crossing the border from a *TargetLink Subsystem* to Simulink are always of the *double* data type during a software-in-the-loop simulation. Hence, if succeeding blocks require other data types, *Data Type Conversion* blocks have to be inserted to perform the required typecast.

Purpose

Proper data types at the border of TargetLink subsystems.

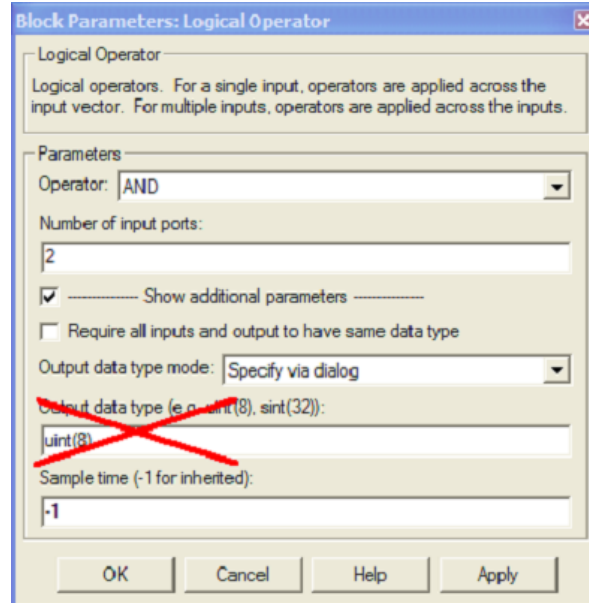


Fig. 54: Data types like `uint(8)` must not be used for the output of a *Logical Operator* or *Relational Operator* block.

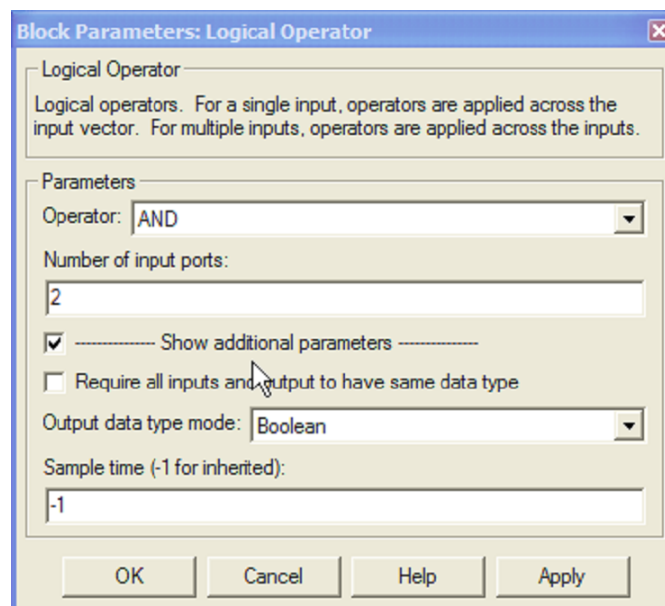


Fig. 55: The output data type is set to *Boolean*. In order to produce the same output data type for the corresponding TargetLink block (full-featured mode or stand-alone mode), the global Simulink option *Implement logical signals as boolean (vs. double)* has to be activated.

Remark

Busses leading out of a *TargetLink subsystem* are supported since TargetLink 2.1. However, the TargetLink generated S-function for a software-in-the-loop simulation outputs *double* signals within buses exclusively.

Example

Figure 56 shows the use of a *Data Type Conversion* block at the TargetLink-Simulink interface.

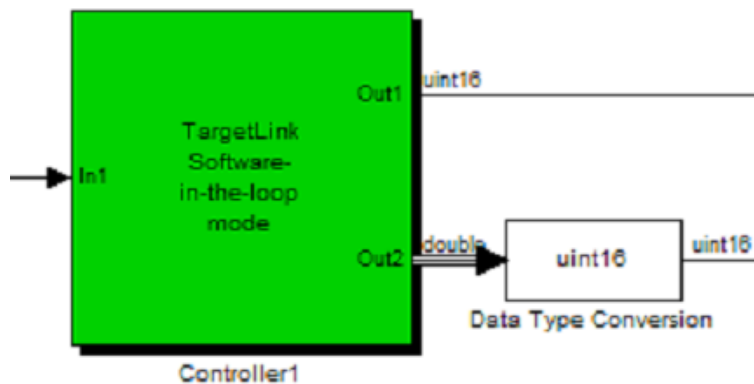


Fig. 56: If data types other than *double* are required for bus signals coming out of a *TargetLink Subsystem*, *Data Type Conversion* blocks have to be inserted. This is only relevant for the software-in-the-loop simulation.

8.4 Avoidance of Overflows

For all integer operations, the specified value range for the output data type as well as those of intermediate variables have to be sufficient to avoid overflows. In particular, overflows must not be used deliberately.

Purpose

Proper modeling style, equal results for Simulink and TargetLink simulation modes.

Remark

Basically, overflows shall not be used as a stylistic element because they are highly implementation specific and constitute a highly intransparent modeling style. Over- (and underflows) are dependent on size and kind of the used data types whose use is intransparent and block-dependent in Simulink, particularly for intermediate results. Moreover, TargetLink does not emulate the over/underflow behaviour in Simulink. In the TargetLink blockset stand-alone, over- and underflows are avoided in principal by saturation.

Example

Figure 57 shows an example of integer arithmetic in which overflows cannot occur.

8.5 Avoidance of Saturation in Integer Arithmetic

Saturation in integer arithmetic via the *Saturate on Integer overflow* option should be avoided through the use of data types with sufficient bit lengths for outputs as well as intermediate results. Excepti-

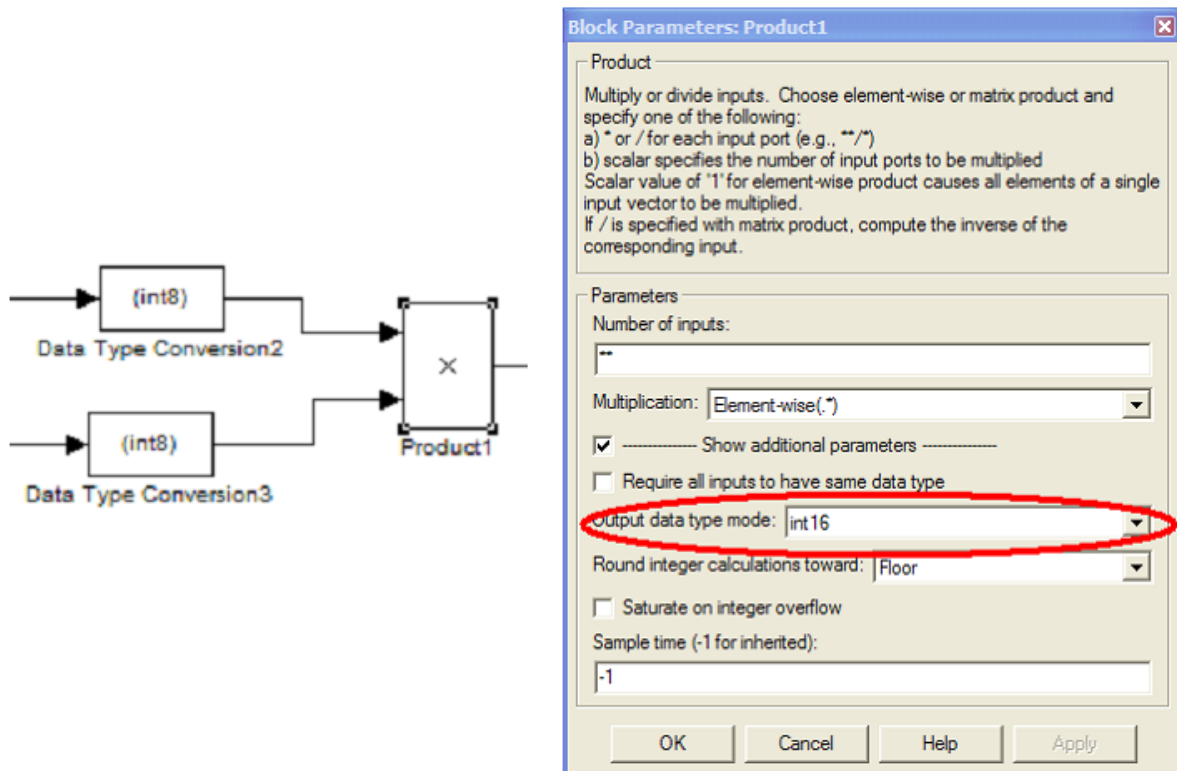


Fig. 57: Example of a *Product* block with integer arithmetic. The *Int16* output data type ensures that no overflow occurs.

ons can only be made for blocks, where it can be assured that the saturation is not carried out on intermediate results but only on the final output signal, like in the following blocks:

- Gain
- Product block with less than three inputs
- Multipoint Switch

If the saturation is part of the control algorithm itself rather than the specific integer arithmetic, the *Saturation* block should be used.

Note: In order to avoid saturation, it is irrelevant whether the option *Saturate on Integer overflow* is set or not but that the data type is sufficient to make sure that no saturation occurs.

Purpose

Proper modeling style, equal results for Simulink and TargetLink simulation modes.

Remark

In general, implementation specific aspects like saturation in integer arithmetic operations have to be handled on the code level. Moreover, the semantics on how saturation is performed in integer arithmetic in Simulink is quite intransparent and blockspecific. The inputs of a *sum* block for instance, are directly cast to the output type and potentially saturated just like any other intermediate result. This may lead to unexpected behaviour and is not emulated by TargetLink where only block outputs are saturated.

Example

Figure 58 shows an example for integer arithmetic.

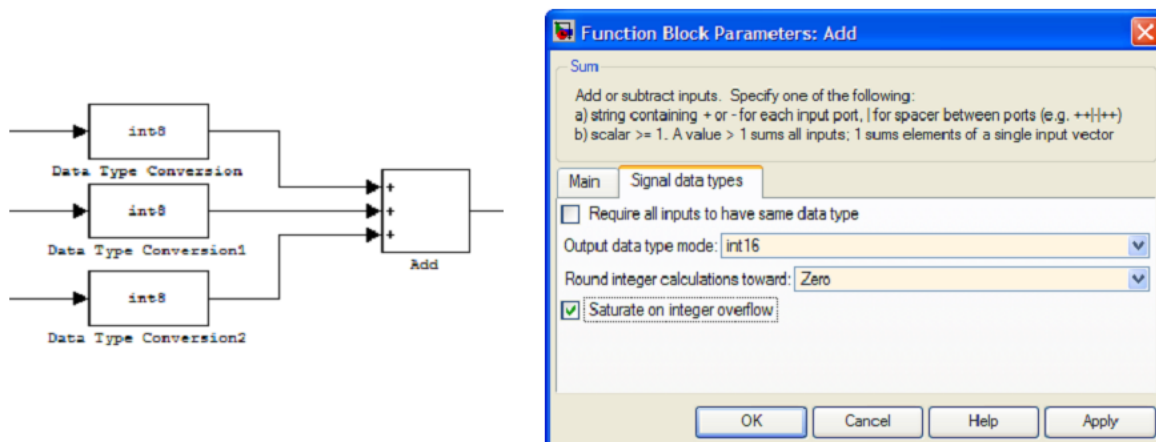


Fig. 58: Example of a *Sum* block with three input signals. If the *Saturate on Integer overflow* option has been activated, then the intermediate results are saturated, which is partly intransparent and might also lead to (incorrect) unexpected results. Here, the data type *Int16* at the output ensures that saturation never occurs, just as a potential over/underflow.

8.6 Avoidance of Rounding Operations

Rounding operations within integer calculations via the property *Round Integer Calculation towards ...* should be avoided. If a rounding function is required then the *Rounding Function* has to be used for

that.

Note: Rounding operations appear if integer output data types are specified for a block in which floating-point values are used also, or if the results of integer operations are not displayable as integer variables (for instance divisions).

Purpose

Proper modeling style, equal results for Simulink and TargetLink simulations.

Remark

The semantics on how the rounding operation is performed is partially intransparent and block-specific in Simulink. TargetLink does not emulate the different rounding methods in integer calculations.

Example

Figure 59 shows an example where integer rounding operations might be required.

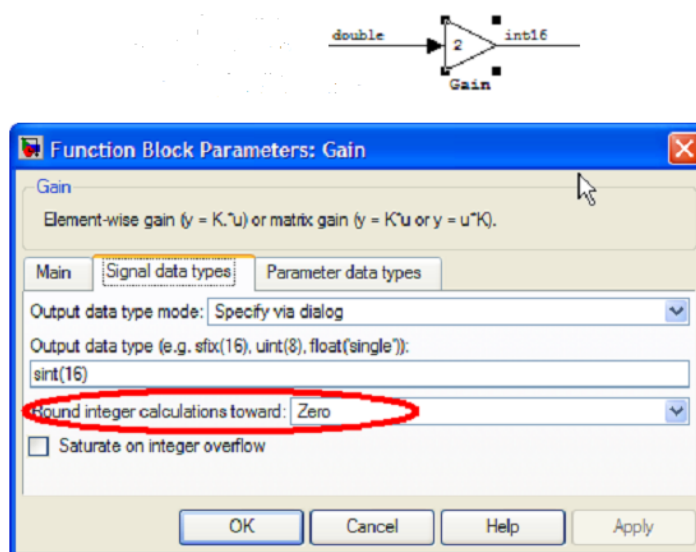


Fig. 59: Example of a *Gain* block which is driven by the floating-point data type *double* and specified to have the integer data type *Int16*, which is the reason why a rounding operation might be required.

8.7 Use of Data Types for Discrete Transfer Fcn, Discrete State-Space and Discrete Filter Blocks

The data types used for input and output variables of *Discrete Transfer Fcn*, *Discrete State-Space* and *Discrete Filter* blocks must be either consistently *floating-point* or consistently *fixed-point* but not a mixture of the two.

Purpose

Code generation with TargetLink.

Remark

Mixing floating-point and fixed-point code in these blocks is currently not supported by TargetLink for reasons of code efficiency.

References

- [TargetLink Production Code Generation Guide \[1\]](#), TargetLink Limitations, Block-Specific Limitations, Discrete Transfer Fcn/Discrete State-Space

8.8 Use of Data Types for Trigonometric Blocks

The data types used for the hyperbolic functions *sinh*, *cosh*, *tanh*, *asinh*, *acosh*, *atanh* within a *Trigonometric Function* block must all be floating-point. For the *atan2* function, the use of 32-bit *integer* input signals and output signals of *unsigned integer* type is not permitted.

Purpose

Code generation with TargetLink.

Remark

The combinations of the above functions and data types are not supported by TargetLink.

References

- [TargetLink Production Code Generation Guide \[1\]](#), TargetLink Limitations, Block-Specific Limitations, Trigonometric Function with atan2

8.9 Use of Data Types for Math Blocks

For TargetLink versions prior to TargetLink 2.1, fixed-point code settings for the *mod* and *rem* functions of the *Math* block have to be specified in the form of *power of two-scaling* without offset for the input signal and *power of two-scaling* for the output signal. The *Arbitrary Scaling* option is not permitted for the above functions for TargetLink versions prior to 2.1. Since TargetLink 2.1, arbitrary scaling is supported but the offsets of input and output must be set to zero.

Purpose

Code generation with TargetLink.

Remark

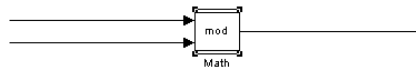
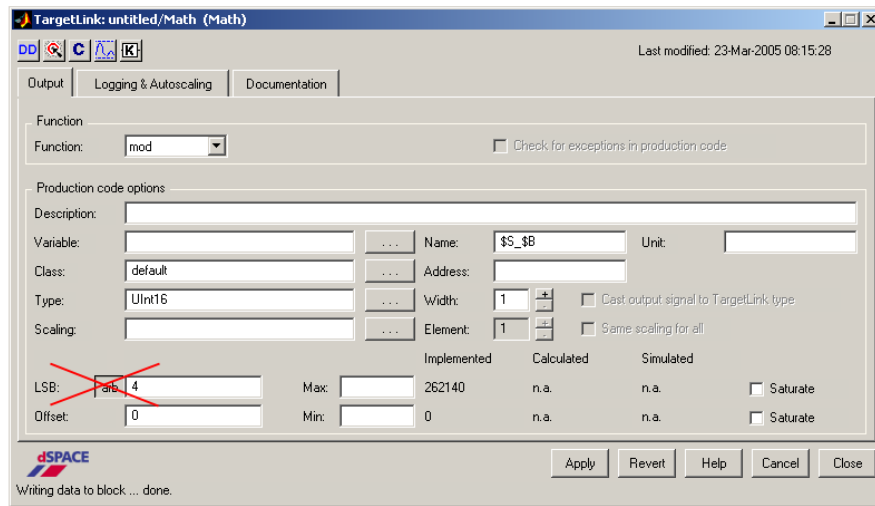
The combinations of the above functions and data types are not supported by TargetLink.

References

- [TargetLink Production Code Generation Guide \[1\]](#), TargetLink Limitations, Block-Specific Limitations, Math with Mod/Rem operator

Example

Inadmissible arbitrary scaling of the output signal of the *Math* block using the *mod* function.



8.10 Use of Data Types for Selector Blocks

If the *external index signal* option is specified for the *Selector* block, all input signals must have identical data type and scaling information.

Purpose

Code generation with TargetLink.

Remark

TargetLink supports this option under the above mentioned limitations for reasons of code efficiency.

References

- [TargetLink Production Code Generation Guide \[1\]](#), TargetLink Limitations, Block-Specific Limitations, Selector

8.11 Use of Data Types for Look-Up Table und Look-Up Table (2D) Blocks

Floating-point data types for the *Look-Up Table* and *Look-Up Table (2D)* blocks must not be used in combination with the *Equidistant with* option.

Purpose

Code generation with TargetLink.

Remark

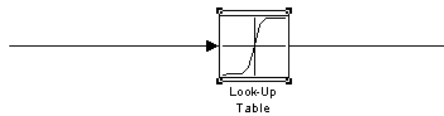
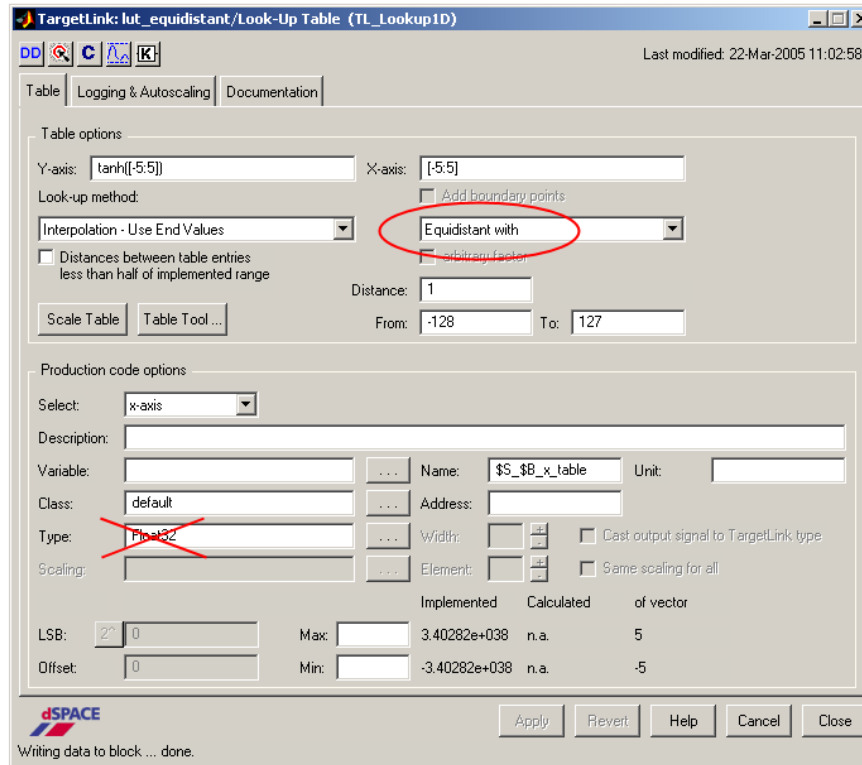
TargetLink currently does not support *floating-point* data types in combination with equidistant break-point data.

References

- [TargetLink Production Code Generation Guide \[1\]](#), TargetLink Limitations, Block-Specific Limitations, Look-Up Table(1D and 2D)

Example

Inadmissible option in combination with *floating-point* data for the *Look-Up-Table* block.



8.12 Use of Data Types for Multiport Switch Blocks

The first input (*control port*) of the *Multiport Switch* block should be driven by a signal with an *unsigned integer* data type.

Purpose

Ensuring a transparent modeling style.

Remark

The input signal of the control port has to be non-negative and is cast to an *integer* number by Simulink during the simulation.

9 Handling of Look-up Tables

This section contains rules relating to the handling of *Look-up Tables* for the purpose of efficient code generation.

9.1 Common Settings for Look-up Table Blocks

The settings for look-up table blocks within one project should be kept as consistent as possible with regard to data types, search algorithms etc.

Purpose

Generation of efficient code.

Remark

TargetLink generates special C functions for look-up tables. If common settings are specified for multiple look-up table blocks, the generated table functions can be reused. This leads to a significant reduction of code size.

References

- [TargetLink Advanced Practices Guide \[2\]](#), Working with Special TargetLink Blocks > Look-Up Table Blocks > Table Functions > Table Function Reuse

Example

The name and the functionality of the function generated for a look-up table block depends on the specified data types, search functions, the look-up table method and a couple of other options. The name of the table function to be generated because of the settings is indicated on the *documentation* page of the block dialog, see fig. 60. The fewer separate look-up table functions are generated, the smaller the resulting code size.

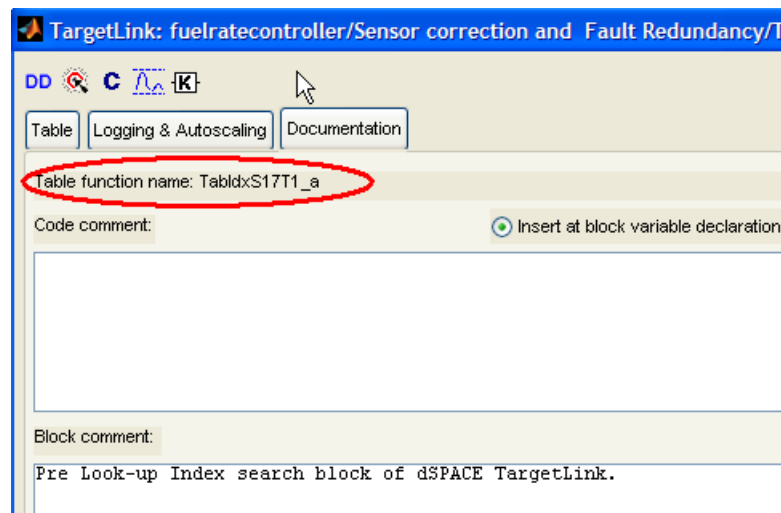


Fig. 60: Name of the generated look-up table function for a look-up table block. If the code generation options are common for multiple blocks, the generated function is automatically reused for each of them.

9.2 Merging of Table Vectors

If the same table(vector) is to be used in multiple Look-up Table blocks, it must be ensured that only one variable is generated in the code. This is accomplished by specifying a fixed variable name and a variable class with the *MERGEABLE* optimization attribute.

Purpose

Generation of efficient code.

Remark

As a result, the table is allocated only once, resulting in less ROM consumption.

Example

Fig. 61 shows the selection of a variable class with the *MERGEABLE* optimization attribute set and a fixed, block-independent variable name.

Fig. 61: Required specification in the block dialog to enable variable merging.

9.3 Common Search Function for Multiple Look-up Table Blocks

If multiple Look-up Table blocks use the same input table vector with identical input signal, *PreLook-Up Index Search* blocks should be used in conjunction with *Interpolation (n-D) using PreLook-up* or *Direct Look-up Table (n-D)* blocks. Hence, the generated search function and the calculated results can be reused.

Purpose

Generation of efficient code.

Remark

Reusing of search functions and the calculated results saves execution time.

References

- [TargetLink Advanced Practices Guide \[2\]](#), Working with Special TargetLink Blocks > Look-Up Table Blocks > Implementing Look-Up Tables > How to Optimize Table Searches and Reuse Search Results

Example

Figure 62 shows an excerpt from the TargetLink demo model *FuelSys*, in which the search results are reused via *PreLook-up Index Search* blocks.

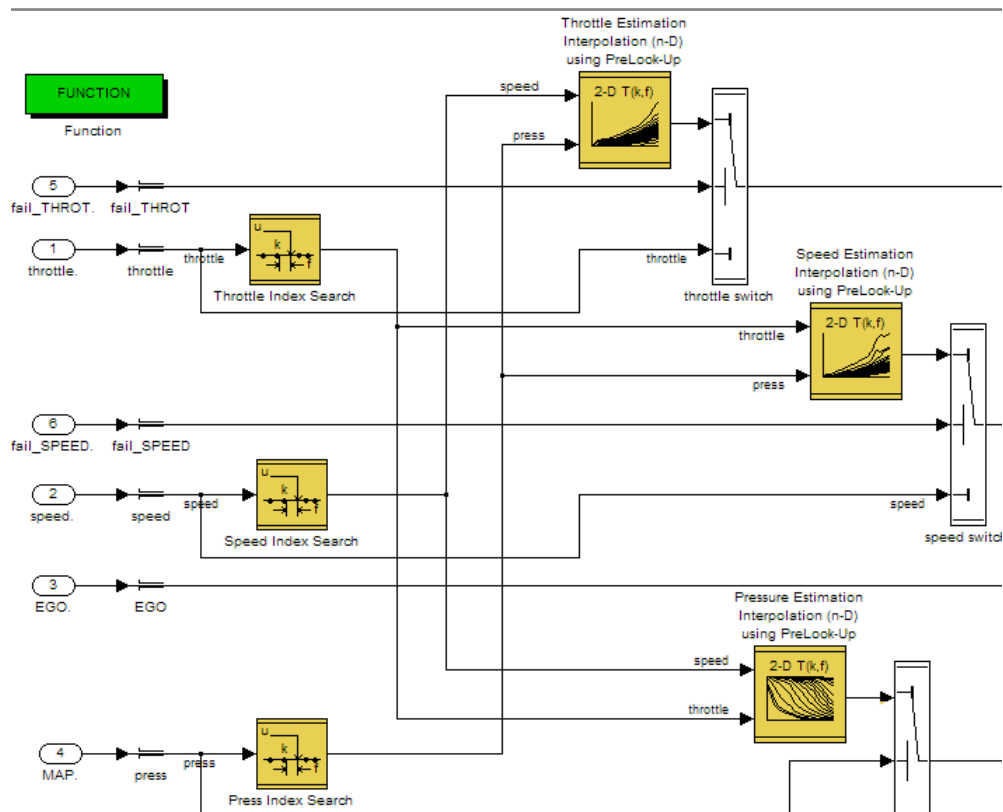


Fig. 62: Mutual search function for multiple blocks.

10 Function Handling

This chapter contains rules relating to the handling of functions for the purpose of efficient code generation.

10.1 Inlining of Functions

The *CompilerInline* and *Optimization* properties for function classes in the Data Dictionary determine when function calls should be replaced by the expansion of the function code (function inlining).

Purpose

Generation of efficient code.

Remark

Since the function call is eliminated when function inlining is performed, the execution time is reduced at the expense of increased code size.

References

- [TargetLink Advanced Practices Guide \[2\]](#), Mapping of Subsystems and Functions > TargetLink Subsystems and Function Inlining

Example

When the *CompilerInline* property is set for a function class in the Data Dictionary, TargetLink creates an *inline* keyword as a prefix for the function definition, see fig. 63, which is a recommendation to the compiler to inline the function.

For functions which reside in the *TargetLink Subsystem* and are called from there, it is also possible to let TargetLink do the function inlining. This behavior is specified via the *Optimization* property and an inlining threshold, see fig. 64. The following options are available:

- CG_AUTO_INLINE: The function is inlined according to an internally calculated cost function and the global inlining threshold.
- CG_FORCE_INLINE: The function is inlined regardless of the value of the internal cost function.
- CG_NO_INLINE: The function is not inlined.

Functions which are generated implicitly by TargetLink rather than being specified explicitly by the user are inlined if the cost function does not exceed the global inlining threshold.

10.2 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.

Purpose

Generation of efficient code.

Remark

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.

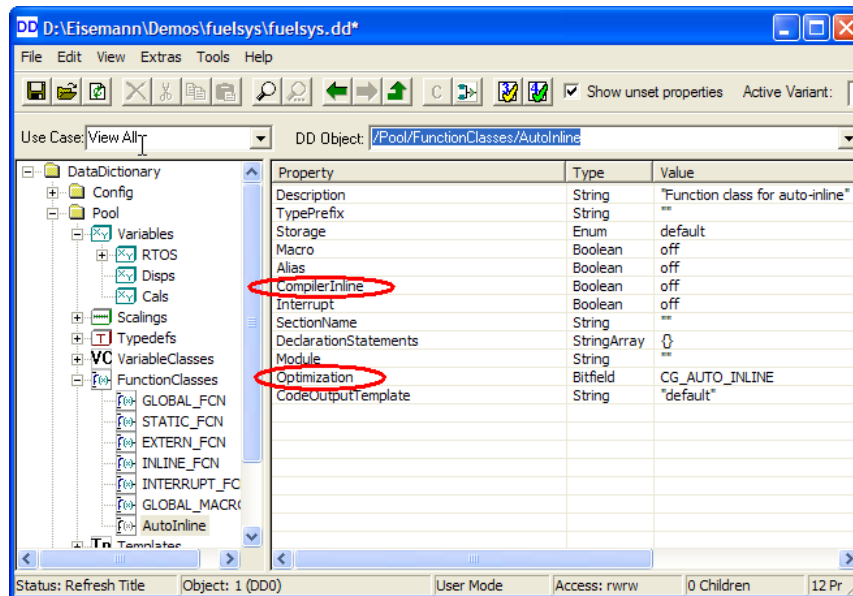


Fig. 63: Function class with settings for the *CompilerInline* and *Optimization* properties.

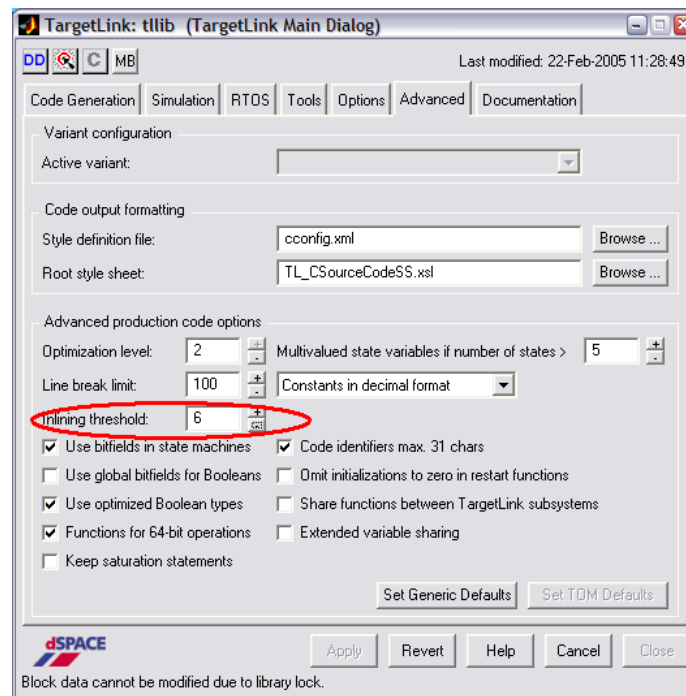


Fig. 64: Global setting for the *Inlining Threshold* in the TargetLink Main Dialog.

References

- [TargetLink Advanced Practices Guide \[2\]](#), Mapping of Subsystems and Functions > Defining the Function Interface > Function Interfaces

Example

In the example in fig. 65, the input variables are passed as function arguments on one hand and as global variables on the other. This can be done for instance, by specifying the predefined *FCN_ARG* and *FCN_REF_ARG* variable classes on one hand and *MERGEABLE_GLOBAL* on the other for the TargetLink *Inport* and *Outport* blocks (in general: depending on the *Scope* property of the variable class). If no TargetLink *Inports* and *Outports* are present, the default behavior of the code generator is applied, i.e. global variables are used.

As a result, TargetLink generates the following code for both subsystems:

```
[...]
/* call of function: Schnittstellen/GlobalVars */
Sa3_GlobalVars();

/* call of function: Schnittstellen/FcnArg */
Sa2_FcnArg(In, &(Out1));
[...]
```

10.3 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*). If the subsystems are structurally identical but differ in the parameter set used, *function reuse* should be applied as well. To make TargetLink generate reused functions, the prototype must be placed in a library and special settings are required, see the example below.

Purpose

Generation of efficient code.

Remark

Function reuse reduces code size, i.e. ROM consumption.

References

- [TargetLink Advanced Practices Guide \[2\]](#), Mapping of Subsystems and Functions > Function Reuse

Example

Fig. 66 shows a model with multiple instances of a filter block. Function reuse makes sure that code for the subsystem is generated only once and called for each instance.

To enable function reuse for the filter blocks, they must be taken from a library into which the block prototype was previously placed. In the case of a Simulink subsystem, a *Function* block and TargetLink *Inports* and *Outports* have to be placed within the subsystem. In addition, the *Make function reusable* option also has to be specified, see fig. 67. Moreover, each instance must be assigned a common name for the function to be generated and a proper function interface. For state charts, the corresponding settings can be specified using the property manager.

For the model in fig. 66, TargetLink generates the following code, applying function reuse:

```
[...]
```

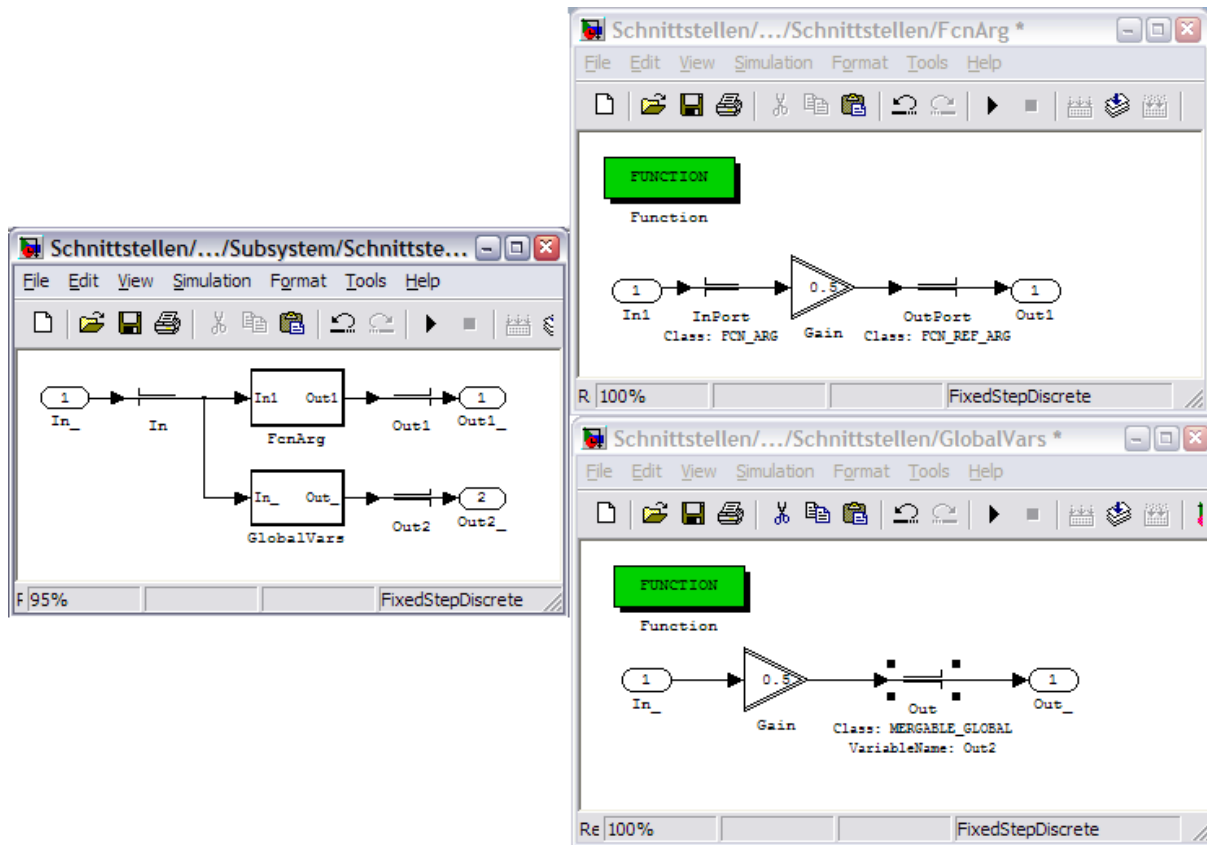


Fig. 65: Passing input parameters as global variables or function arguments.

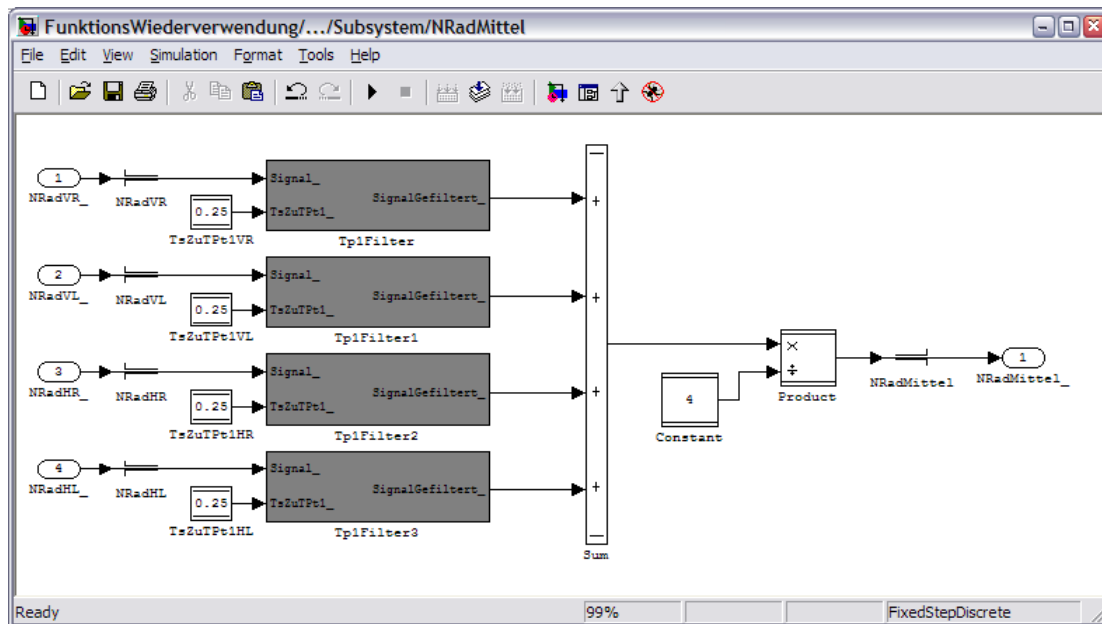


Fig. 66: Function reuse of a subsystem: There are multiple instances *Tp1Filter*, *Tp1Filter1*, *Tp1Filter2*, *Tp1Filter3* for which only one function should be generated.

```
/* call of function: NRadMittel/Tp1Filter */
Tp1Filter(Sa1_NRadVR, 32 /* 0.25 */, &(SFTP11_SignalGefiltert),
&(tagISV_SFTP11_Tp1Filter));

/* call of function: NRadMittel/Tp1Filter1 */
Tp1Filter(Sa1_NRadVL, 32 /* 0.25 */, &(SFTP11_SignalGefiltert_a),
&(tagISV_SFTP11_Tp1Filter1));

/* call of function: NRadMittel/Tp1Filter2 */
Tp1Filter(Sa1_NRadHR, 32 /* 0.25 */, &(SFTP11_SignalGefiltert_b),
&(tagISV_SFTP11_Tp1Filter2));

/* call of function: NRadMittel/Tp1Filter3 */
Tp1Filter(Sa1_NRadHL, 32 /* 0.25 */, &(SFTP11_SignalGefiltert_c),
&(tagISV_SFTP11_Tp1Filter3));

[...]
```

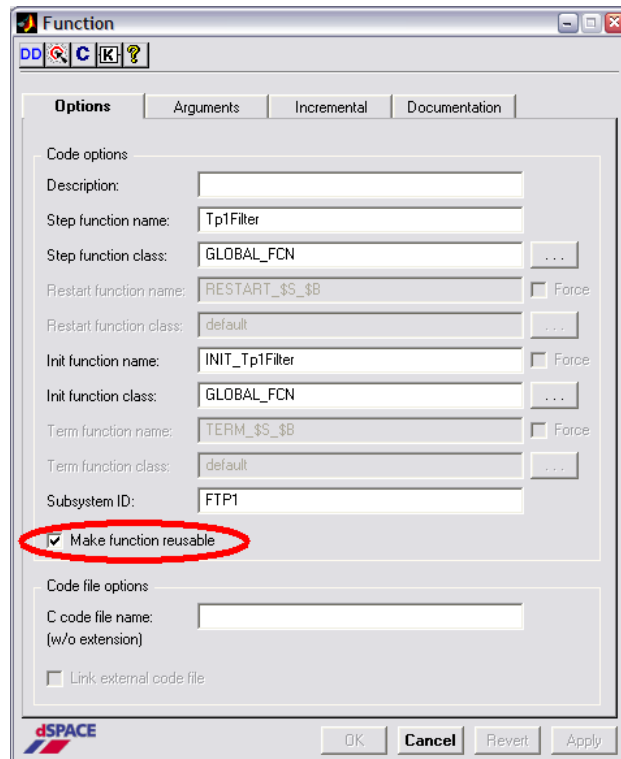


Fig. 67: Settings in the dialog of the *Function* block to enable function reuse.

11 Variable Handling

This chapter contains rules relating to the handling of variables for the purpose of efficient code generation.

11.1 Use of Bitfields

Bitfields allow memory-efficient storage of boolean variables and Stateflow state variables. Two separate options *Use global Bitfields for booleans* and *Use bitfields in state machines*, are available in the TargetLink Main Dialog.

Purpose

Generation of efficient code.

Remark

Using bitfields can reduce memory consumption at the cost of increased execution time for accessing variables stored in bitfields.

Example

Fig. 68 shows the two options for bitfields on the Advanced tab of the TargetLink Main Dialog.

Please note that boolean variables are only stored in bitfields if they are scalar, not logged, and if the selected variable class is *default*. A *Boolean* data type also has to be selected.

If the option *Use bitfields in state machines* is selected, the storage of the current state in Stateflow is coded in bitfields in which the variables are allocated with the proper length.

11.2 The Impact of Variable Classes on Optimization

A variable class other than *default* should only be specified if this is really necessary, for instance to make a variable calibratable, measurable etc. If user-defined variable classes are used, a maximum number of optimization attributes should be set whenever possible.

Purpose

Generation of efficient code.

Remark

Specifying a variable class other than *default* affects the optimization performed by the code generator. Variables with *default* variable class can be merged, moved into conditional branches or even erased entirely. If a user-specified variable class is selected, the optimizations performed by the code generator depend on the optimization attributes *ERASABLE*, *MOVABLE*, *MERGEABLE* and *SCOPE_REDUCIBLE*.

Example

Fig. 69 shows the specification of the *default* variable class which grants the code generator the maximum freedom with regard to optimizations to be performed.

If a variable class other than *default* is specified, the attributes for the *Optimization* property of the variable class determine the optimization mechanisms of the code generator, see fig. 70.

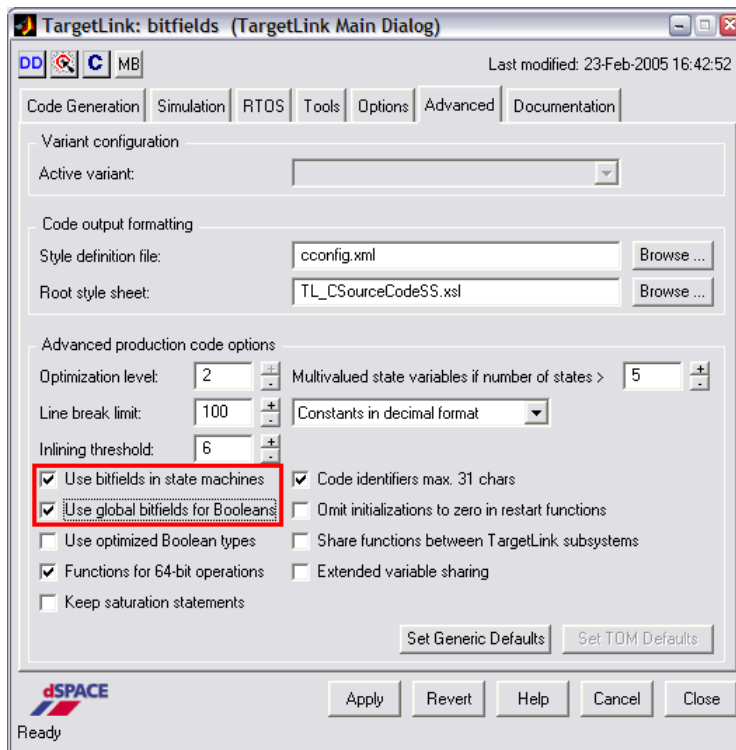


Fig. 68: Advanced page of the TargetLink Main Dialog with the available bitfield options.

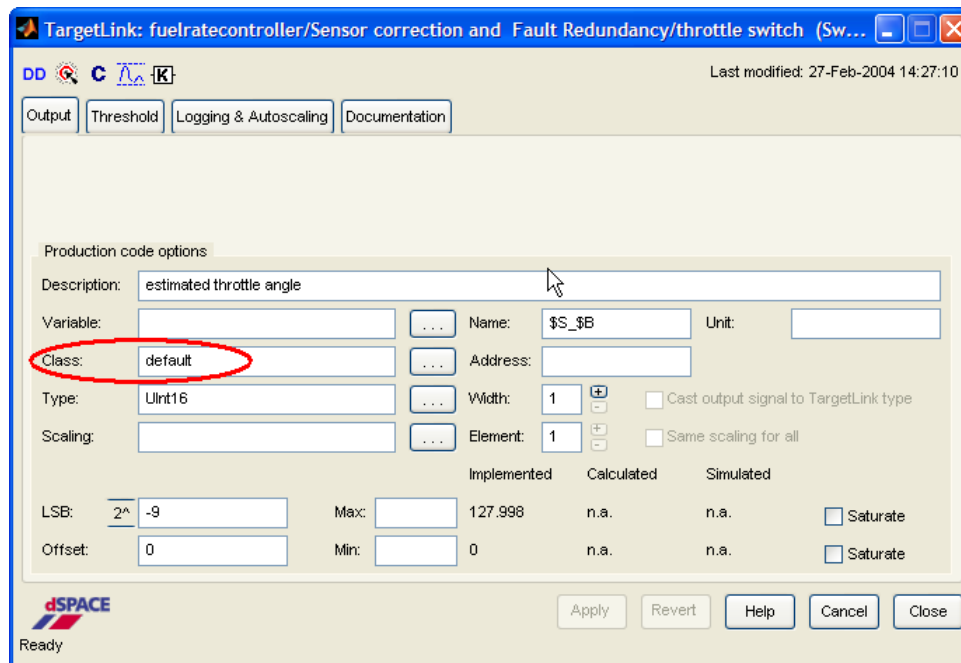


Fig. 69: Selection of the *default* variable class in the block dialog.

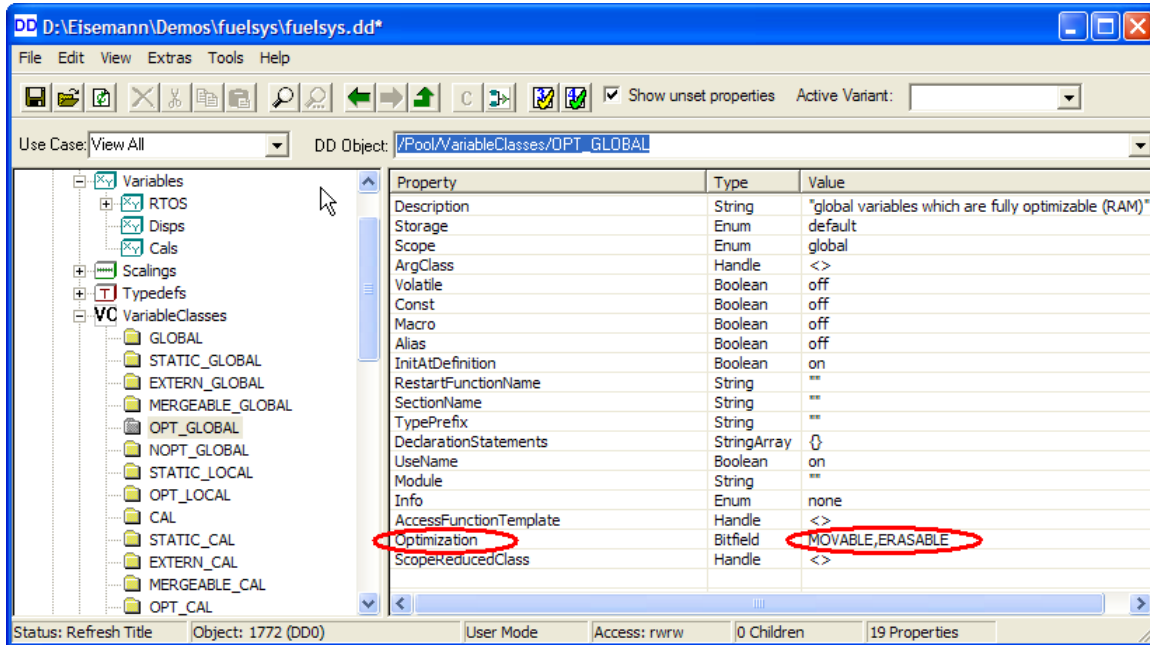


Fig. 70: Optimization property of a variable class.

11.3 Erasing of Variables

If the *ERASABLE* optimization attribute is set for a variable class, all the variables of that class can be erased by the code generator if their values are not needed.

Purpose

Generation of efficient code.

Remark

The erasing of superfluous variables and code reduces code size and execution time.

Example

Fig. 71 shows an example with two variables which are irrelevant to the rest of the model. The code generator uses the *ERASABLE* optimization attribute to decide whether the variables must be deleted or not. The attribute is set for the *DISP_ERASABLE* variable class and cleared for the other variable class, see fig. 72.

As a result, TargetLink generates the following code:

```
Void Subsystem(Void)
{
    /* Gain: Subsystem/Class DISP_NOT_ERASABLE */
    Sal_Class_DISP_NOT_ERASABLE = Sal_In * 5;

    /* Outport: Subsystem/Out. */
    Sal_Out = Sal_In;
}
```

The calculation of the variable *Sal_Class_DISP_NOT_ERASABLE* is carried out, although the result is not necessary for the rest of the model. The calculation of the variable *Sal_Class_DISP_ERASABLE*

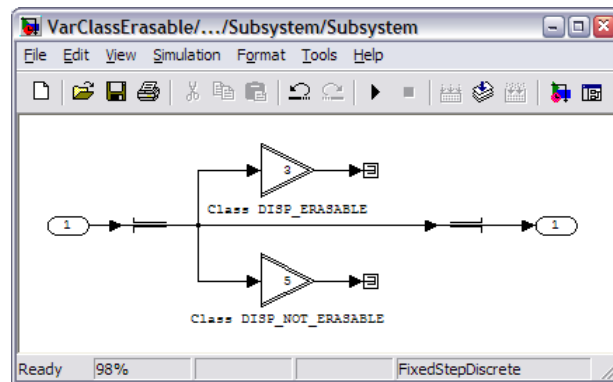


Fig. 71: Model with two irrelevant variables which could be erased.

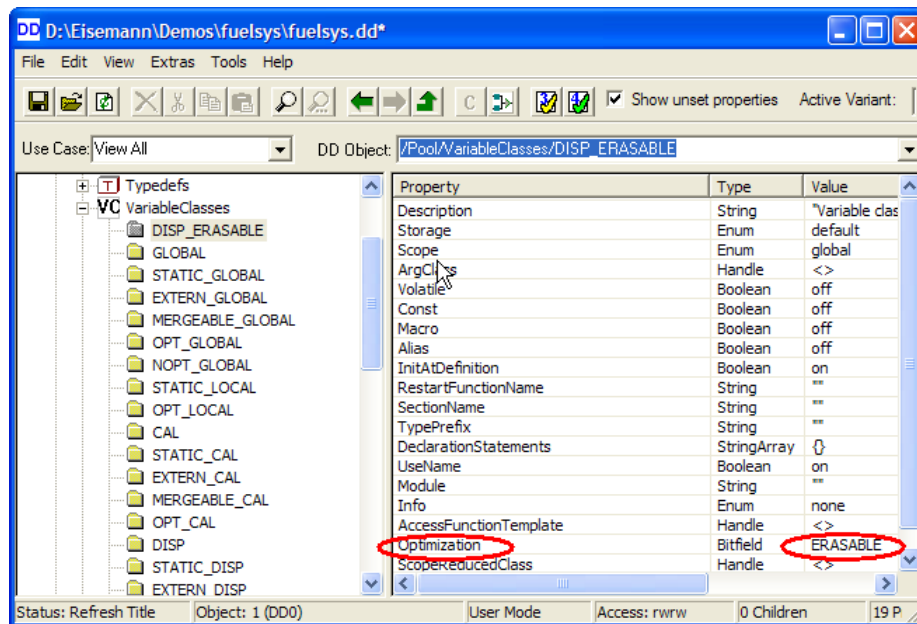


Fig. 72: The *ERASABLE* attribute is set, enabling the code generator to delete the variable if possible.

is avoided due to the specified optimization attribute. In general, intermediate variables are only erased if the *ERASABLE* attribute is set.

11.4 Merging of Variables

The *MERGEABLE* optimization attribute of a variable class enables the generation of only one variable in the C code for multiple variables in the model with identical names. Without the *MERGEABLE* attribute, the code generator would issue an error message.

Purpose

Generation of efficient code.

Remark

The merging of variables in one single instance in the generated code saves RAM and simplifies calibration, since only one variable needs to be adapted.

References

- [TargetLink Advanced Practices Guide \[2\]](#), Configuring TargetLink and Adapting Code to Company Coding Styles > Variable Classes > Example of Merging Multiple Declarations of Variables

Example

Fig. 73 shows a variable class with the *MERGEABLE* attribute set. Variables with identical names are only merged in one variable if the *MERGEABLE* attribute is set for the respective variable classes.

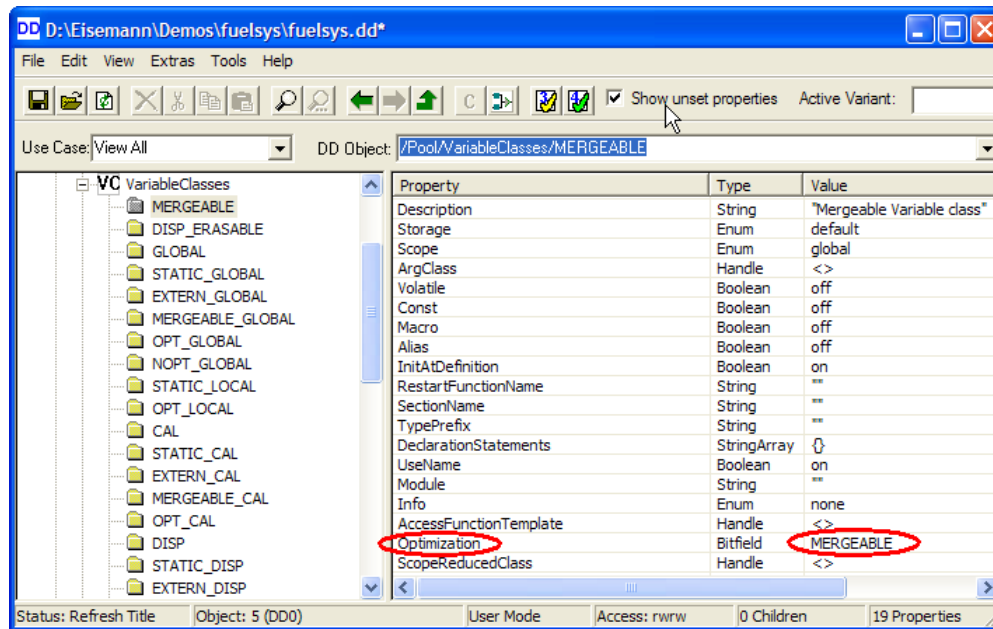


Fig. 73: Setting the *MERGEABLE* attribute for a variable class.

11.5 Reducing the Scopes of Variables

The scope of variables can be reduced via code generator optimizations if the *SCOPE_REDUCIBLE* attribute is set for the respective variable class.

Purpose

Generation of efficient code.

Remark

From a software engineering point of view, variables should always be generated with minimum scope. The code is then better structured and name collisions are avoided. If reduction down to the function-local level is possible, RAM size can be reduced at the cost of increased stack consumption.

References

- [TargetLink Advanced Practices Guide \[2\]](#), Optimizing the Production Code > Optimizing an Entire TargetLink System > Optimizing the Code by Scope Reduction

Example

Fig. 74 shows the *Optimization* and *Scope Reduced Class* properties of a variable class, which are relevant to the scope reduction of variables. To enable scope reduction, the *SCOPE_REDUCIBLE* attribute has to be set for the corresponding variable class. The *Scope Reduced Class* property determines how far the scope should be reduced. The following four different levels exist:

- Global
- Module global (static global)
- Static function local (static local)
- Function local

If *default* has been selected for the variable class, TargetLink tries to reduce the scope in the above order down to the lowest level possible.

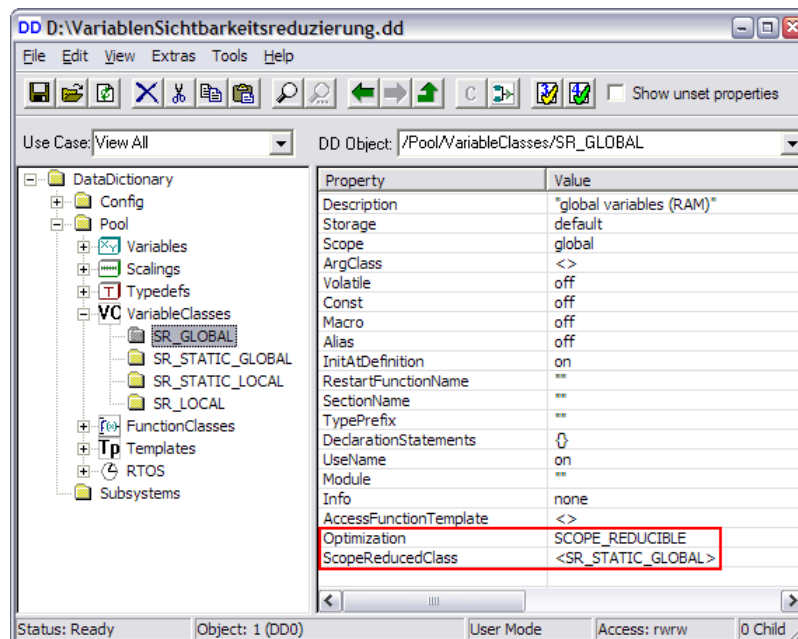


Fig. 74: Variable class with enabled scope reduction.

11.6 Moving of Variables

The *MOVABLE* optimization attribute for variable classes signals to the code generator that the code for variables of that particular class can be moved into dependent branches whenever possible.

Purpose

Generation of efficient code.

Remark

When code is moved into dependent branches, intermediate results are calculated only if they are really required. This reduces execution time.

References

- [TargetLink Advanced Practices Guide \[2\]](#), Optimizing the Production Code > Optimizing an Entire TargetLink System > Optimizing Logging > The Variable Class Attribute 'Movable'

Example

Fig. 75 shows a subsystem which contains branches that do not need to be calculated. The *MOVABLE* optimization attribute is set for the *DISP* variable class, see fig 76.

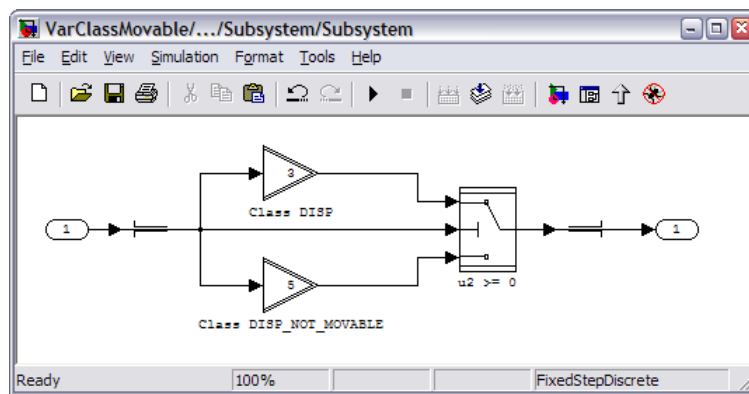


Fig. 75: Model with two *Gain* blocks, only one of which has a movable output variable that can be moved into a dependent branch. As a consequence, the variable is only calculated if the result is required for the subsequent *Switch* block.

TargetLink generates the following code:

```
Void Subsystem(Void)
{
    /* Gain: Subsystem/Class DISP_NOT_MOVABLE */
    Sal_Class_DISP_NOT_MOVABLE = Sal_In * 5;

    /* SwitchSubsystem/u2 >= 0 */
    if (Sal_In >= 0) {

        /* Gain: Subsystem/Class DISP */
        Sal_Class_DISP = Sal_In * 3;

        /* # combined # Outport: Subsystem/Out. */
        Sal_Out = Sal_Class_DISP;
    }
}
```

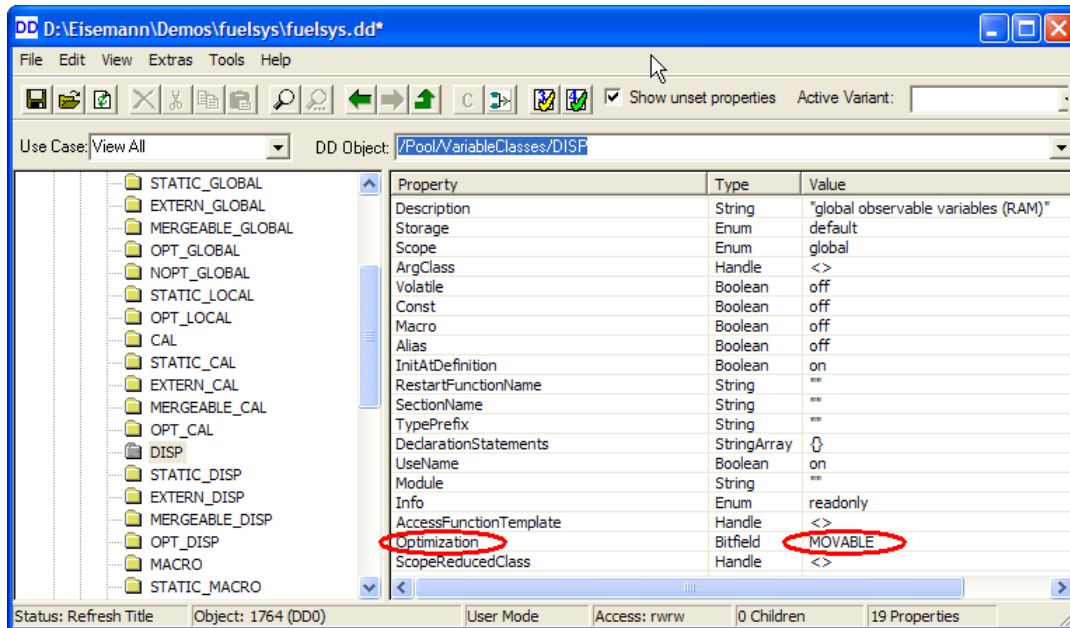


Fig. 76: The *MOVABLE* optimization attribute is set for the *DISP* variable class. As a consequence, variables of that class can be moved into dependent branches.

```

else {

    /* # combined # Outport: Subsystem/Out. */
    Sal_Out = Sal_Class_DISP_NOT_MOVBABLE;
}

```

As a consequence, the `Sal_Class_DISP` variable is calculated only if it is really required, whereas the `DISP_NOT_MOVBABLE` variable, which stems from a non-movable class, is calculated in each time step whether it is required or not.

12 Stateflow Code

This chapter contains rules for the generation of efficient production code for Stateflow.

12.1 Use of Transitions in Control Flow

In a control flow graph, each junction with outgoing transitions should have one transition with neither an event nor a condition specified. The only exception to this rule are situations in which only one transition exists.

Purpose

Generation of well-structured and efficient code.

Remark

Adherence to this rule enables the control flow to be translated directly into structured C code constructs.

Example

Figures 77 and 78 show examples of proper and improper transition modeling.

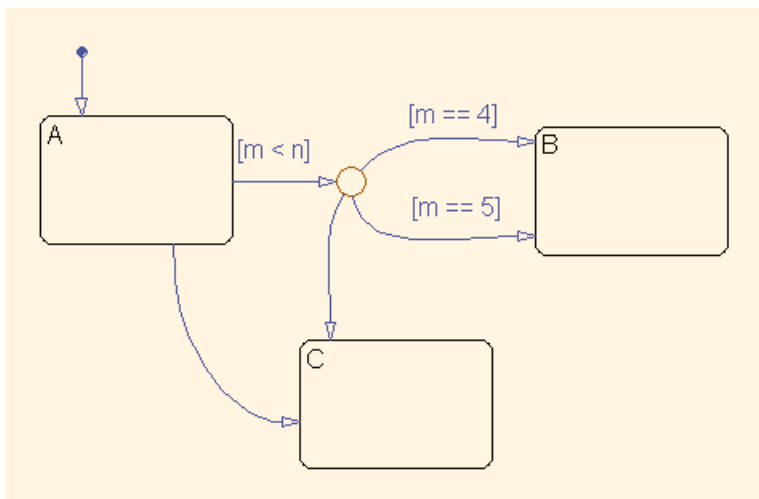


Fig. 77: Proper modeling. A transition devoid of events and conditions is present.

12.2 Specification of Range Information for Variables

The properties *sf.min* and *sf.max* should be specified for each Stateflow variable. In addition, the range information should be as tight as possible, meaning a high minimum and low maximum.

Purpose

Generation of optimal code.

Remark

The tighter the ranges for Stateflow variables, the smaller the data types of intermediate result variables that can be chosen by TargetLink's internal stateflow scaling algorithm. This leads to operations with smaller bit sizes.

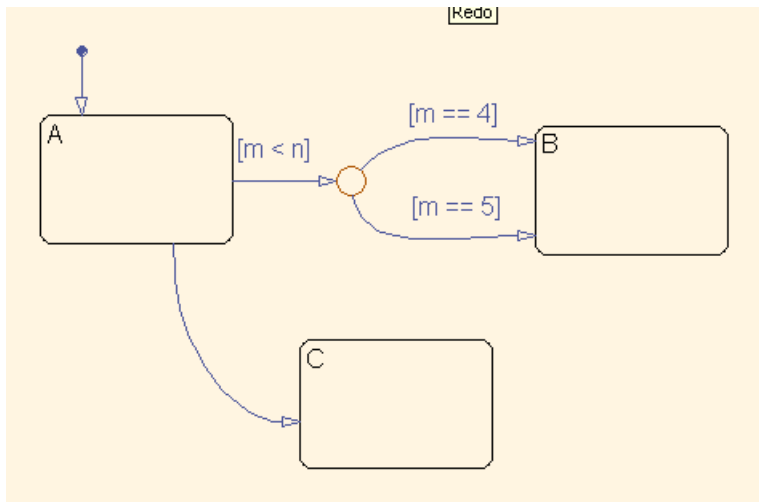


Fig. 78: Improper modeling. The outgoing transitions of the junction all have a condition or an event specified.

12.3 Avoidance of Complex Fixed-Point Expressions

Using complex expressions for fixed-point operations within the Stateflow action language shall be avoided. If a condition or an assignment contains multiple fixed-point operands, intermediate variables with *Temporary* scope should be introduced for each operation to which the intermediate result is assigned.

Purpose

Generation of optimal code.

Remark

Without intermediate variables, the explicit specification for ranges and data types for intermediate results cannot be performed. TargetLink therefore has to assume a maximum range for intermediate results, which often requires 32-bit arithmetic. Introducing intermediate variables allows the ranges to be restricted so that smaller bit lengths can be used.

References

- [TargetLink Production Code Generation Guide \[1\]](#), TargetLink Basics, Integer Calculation Basics, Implementation of Integer Calculation Operations

Example

Fig. 79 show improper use of a complex fixed-point expressions in Stateflow. This leads to the following code fragment, which contains lots of 32-bit data types:

```

Sb1_Out1_ = (Int16) (((((Int16) (UInt16) (((UInt16) (((UInt16) Cb1_a) *
  ((UInt16) (Int16) (Sb1_In2_ << 8))) >> 8)) +
  ((UInt16) (((UInt32) (((UInt32) (((UInt16)
  ((UInt16) (-(UInt16) Cb1_a))) + 256)) *
  ((UInt16) (Int16) (Sb1_In3_ << 8))) << 15)) >> 23)))) * 100)
  >> 8);
  
```

Proper modeling in Stateflow requires the splitting of complex expressions into operations with two operands each and the introduction of auxiliary variables for intermediate results, see fig. 80. The

following code fragment is now generated, requiring no 32-bit arithmetic:

```
Sb1_Out1_ = (Int16) (((Int16) (((Int16) (((UInt16) Cb1_a) * ((UInt16)
(Int16) (Sb1_In2_ << 8))) >> 8)) + ((Int16) (((UInt16)
(((UInt16) (-((UInt16) Cb1_a))) + 256)) * ((UInt16) (Int16)
(Sb1_In3_ << 8))) >> 8))) * 100) >> 8);
```

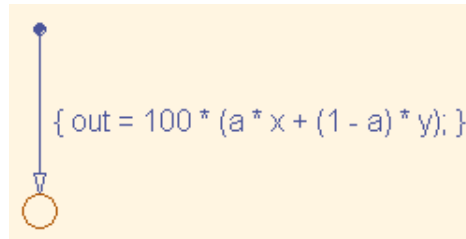


Fig. 79: Complex expression involving multiple 16-bit fixed-point operands which leads to the generation of 32-bit arithmetic.

12.4 Local or Output Variables as State Variables

If the current state of a Stateflow block should be made available through a variable or a given variable has to implement the state using predefined values, a variable with *Output* or *Local* scope has to be specified. The corresponding state value must be assigned to the variable in the first instruction of the entry action.

Purpose

Generation of optimal code.

Remark

The code generator has to store the current state in a variable anyway. Adherence to the rule might save some memory if the state has to be made available.

References

- [TargetLink Advanced Practices Guide \[2\]](#), Working with Stateflow, Optimizing the Code for a State Machine, Using State Variables as State Flags

12.5 Avoidance of TargetLink Specifications for Stateflow Input Variables

The explicit specification of TargetLink properties for input variables to Stateflow charts shall be avoided.

Purpose

Generation of optimal code.

Remark

TargetLink uses the output variables of preceding blocks as input variables for Stateflow. If TargetLink properties are specified for Stateflow input variables, these variables are generated and used for calculations instead of the output variables of the preceding blocks. The Stateflow input variables need function parameter classes assigned to them only if functions with a well-defined interface have to be generated for the Stateflow chart.

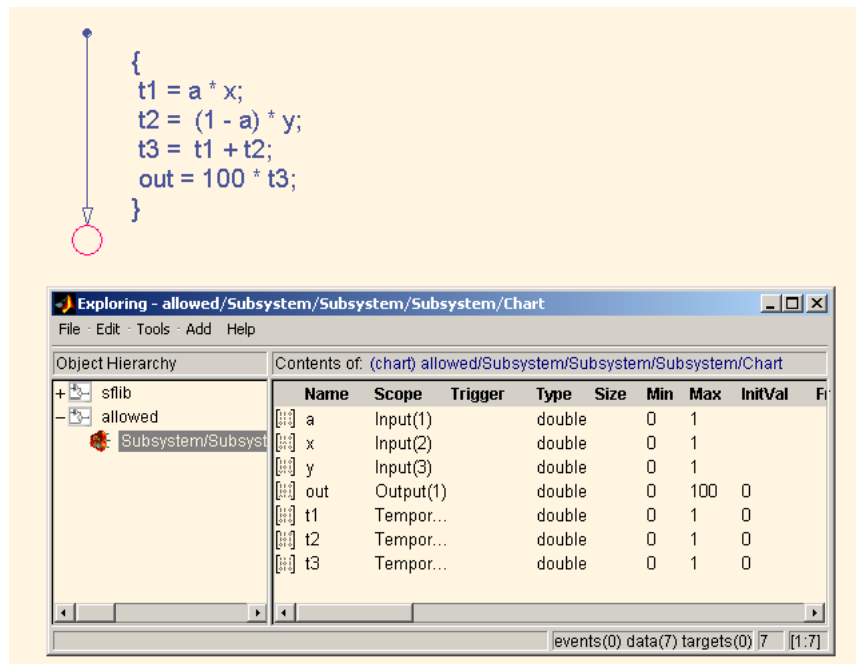


Fig. 80: 32-bit arithmetic can be avoided by introducing auxiliary variables.

Example

The model in fig. 81 shows a Stateflow input variable *a* with *double* type.

If code is generated and no TargetLink properties are specified, see fig. 82, the Stateflow input variable is eliminated and replaced by the output variable of the preceding *Gain* block.

However, if TargetLink properties like the name in the generated code or the variable class are specified for *a*, see fig. 83, the variable automatically appears in the generated code.

12.6 Temporary Variables

If the value of a Stateflow variable is not required for subsequent time steps, this variable shall be specified with *Temporary* scope.

Purpose

Generation of optimal code.

Remark

Temporary variables can be allocated on the stack by TargetLink, while local variables are usually located in RAM.

12.7 Usage of States

States should only be used for modeling where real state machine behavior is required. Model fragments which are basically intended for implementing pure control flow should be modeled using transitions and junctions exclusively, not states.

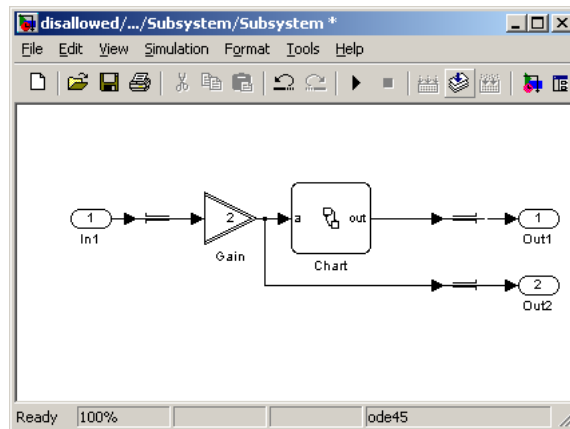


Fig. 81: Model with a Stateflow input variable *a* which is driven by a *Gain* block.

Name	sf_name	sf_stepfunctionclass	sf_class	sf_sftype
out	\$C_0		default	int16
Chart	\$C_0	GLOBAL_FCN		
event	\$C_0			
a	\$C_0		default	double
TargetLink M...				
Gain				
In1				
Out1				
Out1_1				

Fig. 82: No specified TargetLink properties for the input variable *a*.

Name	sf_name	sf_stepfunctionclass	sf_class	sf_sftype
out	\$C_0		default	int16
Chart	\$C_0	GLOBAL_FCN		
event	\$C_0			
a	myname		GLOBAL	double
TargetLink M...				
Gain				
In1				
Out1				
Out1_1				

Fig. 83: Specification of TargetLink properties for Stateflow input variable *a* leads to inefficient code.

Purpose

Generation of optimal code.

Remark

The usage of states leads to the generation of state variables and code to implement the respective behavior. States should therefore only be used if their usage is required in principal to avoid the generation of superfluous code fragments.

Example

Examples of improper usage of states for control flow modeling are shown in fig. 84, while the proper way is demonstrated in in fig. 85.

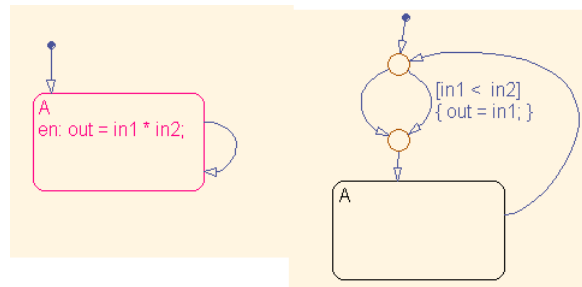


Fig. 84: Control flow modeling using superfluous states.

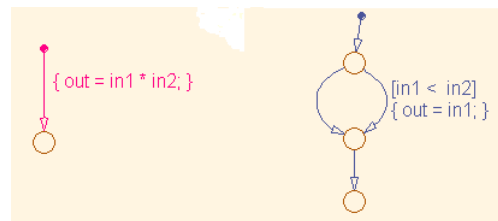


Fig. 85: Elimination of the state in fig. 84 representing pure control flow.

12.8 Usage of Design Patterns for Control Flow

If language constructs like loops, *For* statements, *While* statements etc. have to be modeled in Stateflow, special design patterns should be used.

Purpose

Generation of optimal code.

Remark

The usage of design patterns makes sure that TargetLink can translate individual segments of the control flow into efficient C language constructs.

Example

The following design patterns for control flow should be used whenever applicable:

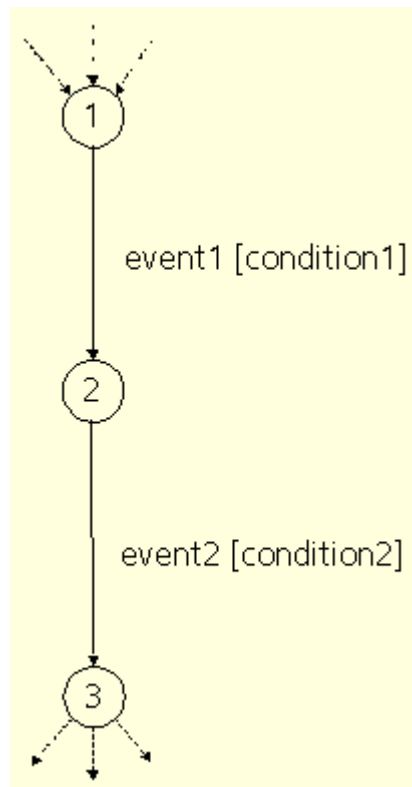


Fig. 86: Design pattern for a logical *and* condition.

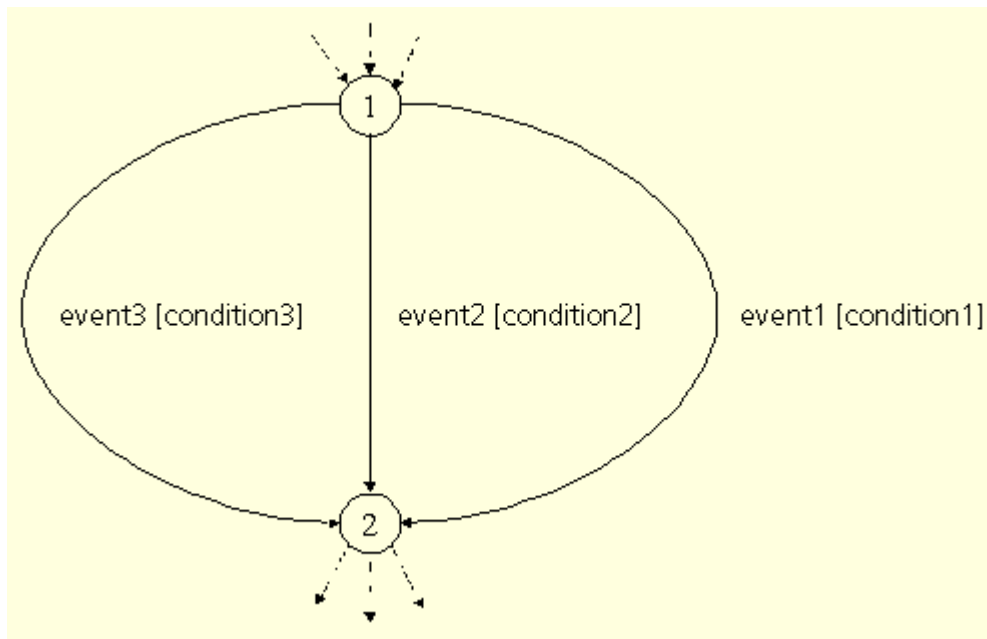


Fig. 87: Design pattern for a logical *or* condition.

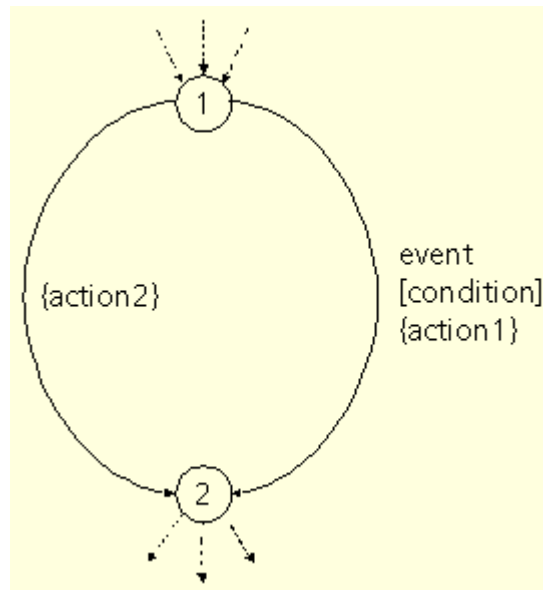


Fig. 88: Design pattern for an *If-Then* conditional statement.

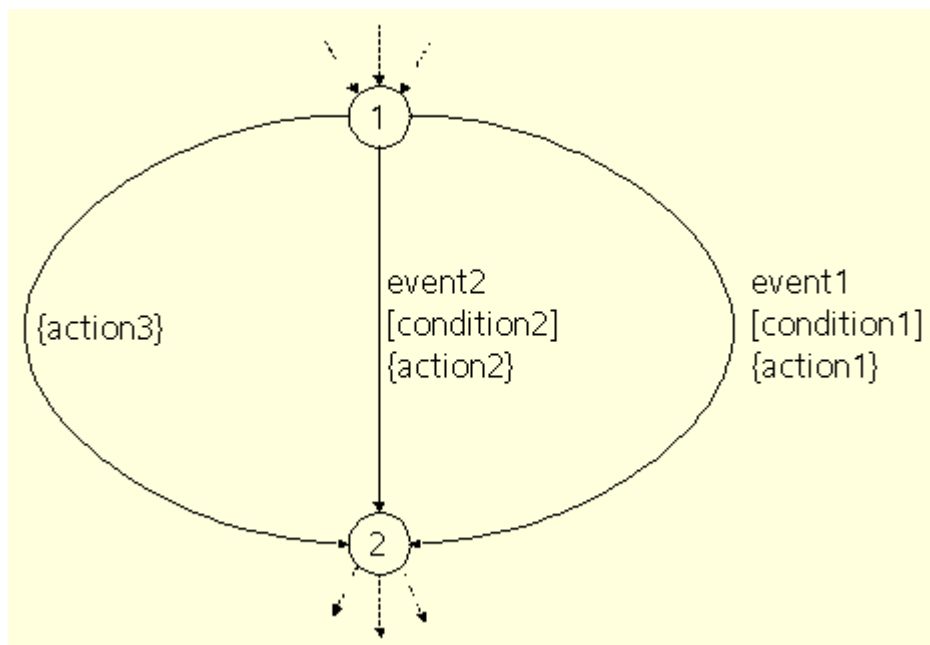


Fig. 89: Design pattern for an *If-Then-Else-If* conditional statement.

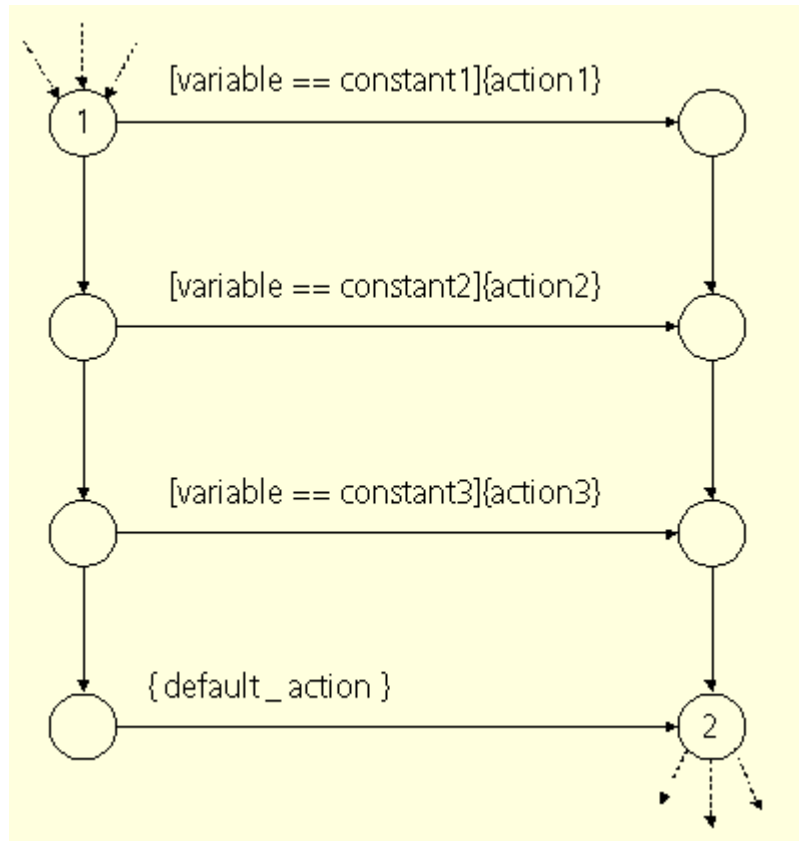


Fig. 90: Design pattern for a *Switch* statement with default branch.

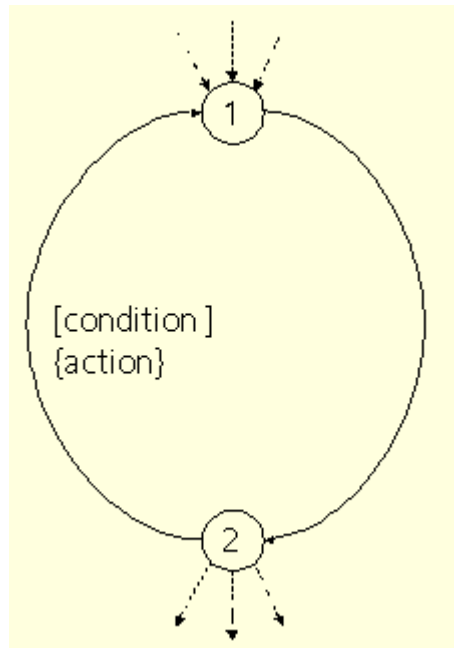


Fig. 91: Design pattern for a *While* loop statement.

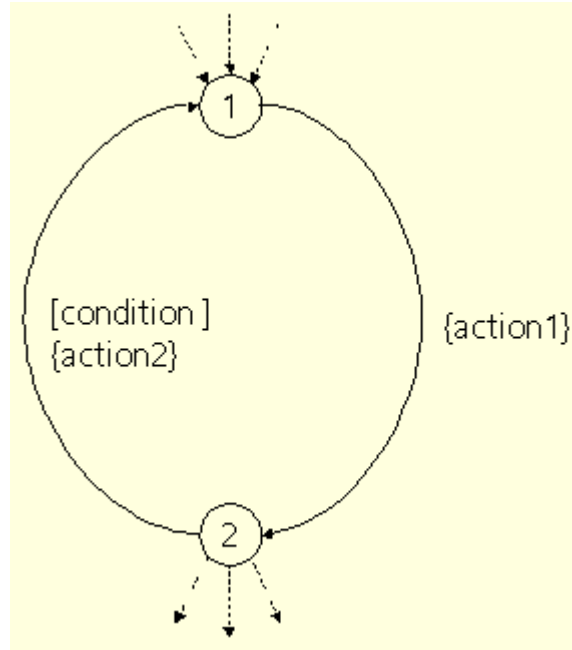


Fig. 92: Design pattern for a *While* loop statement.

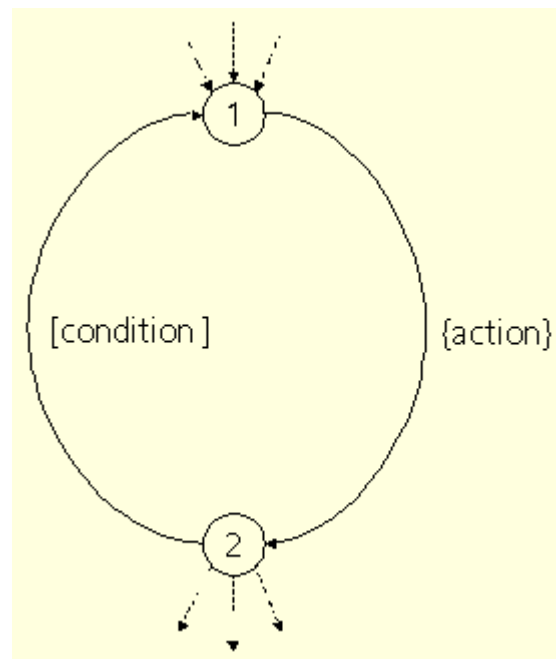


Fig. 93: Design pattern for a *Do-While* loop statement.

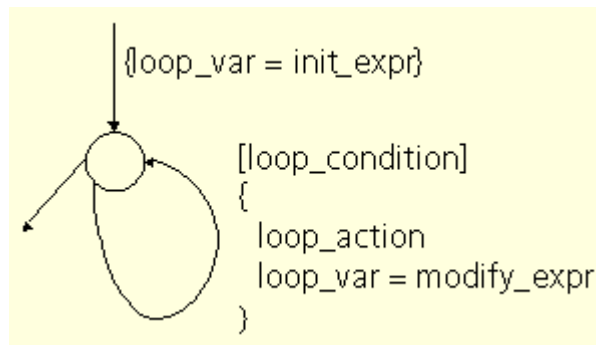


Fig. 94: Design pattern for a *For* loop statement.

13 Fixed-Point Code Generation

This chapter deals with TargetLink settings that are relevant for the generation of highly efficient fixed-point code.

For the rules in this chapter, it is helpful to consider TargetLink's scaling method. TargetLink uses a scaling factor S_x and an offset O_x to represent a real world value x using an integer representation x' in the following form:

$$x = S_x \cdot x' + O_x$$

The proper setting of O_x and S_x is the main objective of the rules in this chapter.

13.1 Power of Two-Scaling

Power of Two-scaling factors are preferable to arbitrary scaling factors.

Purpose

Generation of efficient code.

Remark

Scaling factors which are specified as powers of two lead to very efficient implementations via bitshifts, particularly if divisions and multiplications are involved. On most processors, bitshifts require less execution time than multiplications and divisions.

References

- [TargetLink Production Code Generation Guide \[1\]](#), TargetLink Basics, Integer Calculation Basics, Implementation of Integer Calculation Operations

Example

Scaling factors are specified as power of two in the block dialog by selecting $2^{\wedge}$ for the *LSB* control element, see [fig 95](#).

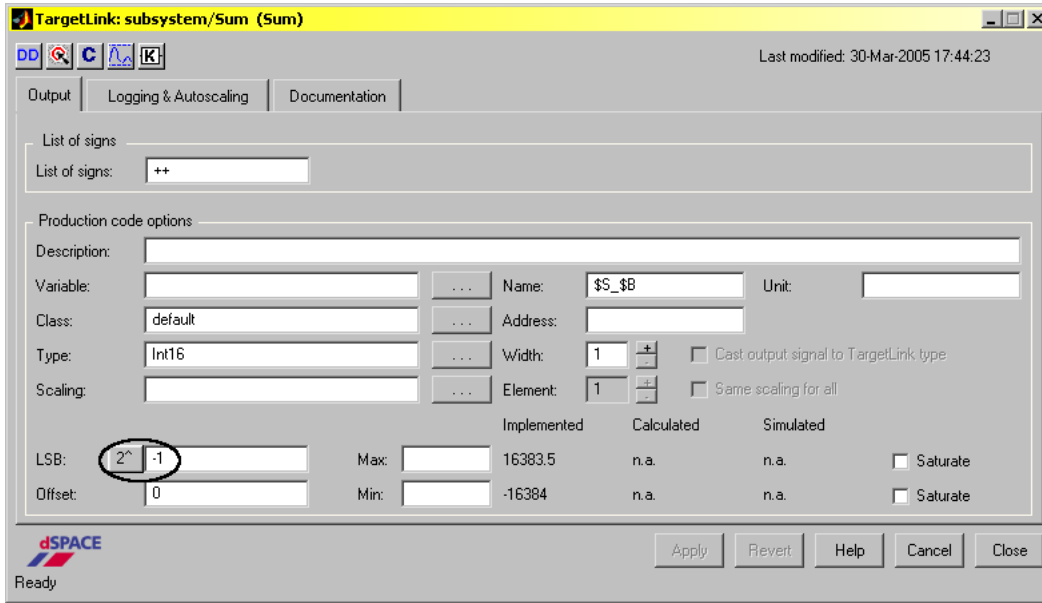


Fig. 95: Selection of power of two-scaling.

For the example model in fig. 96, the scalings of the inputs and output are combined and implemented as bitshifts, leading to the following code:

```
Sal_OutPort = (Int16) (((Int16) (Sal_InPort1 >> 3)) +
                        ((Int16) (Sal_InPort2 >> 2)));
```

13.2 Offsets at Block Inputs and Outputs

Offsets should be preferably set to zero if the accuracy is sufficient or at least chosen in a coordinated manner for block inputs and outputs.

Purpose

Generation of efficient code.

Remark

Non-zero offsets for block inputs, outputs, states and parameters lead to the generation of additional code and hence to increased code size and execution time.

Example

Taking scaling parameters (scaling factors S_i and offsets O_i) into consideration, an addition for instance can be written as

$$y' = \frac{S_{x1}}{S_y} x'_1 + \frac{S_{x2}}{S_y} x'_2 + \frac{O_{x1} + O_{x2} - O_y}{S_y}$$

where x'_1 , x'_2 and y' are the fixed-point representations of the summands and the result.

A multiplication with scaling parameters can be expressed as

$$y' = \frac{S_{x1} \cdot S_{x2}}{S_y} x'_1 \cdot x'_2 + \frac{S_{x1} \cdot O_{x2}}{S_y} x'_1 + \frac{S_{x2} \cdot O_{x1}}{S_y} x'_2 + \frac{O_{x1} \cdot O_{x2} - O_y}{S_y}$$

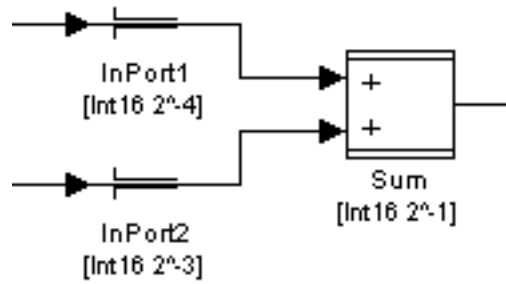


Fig. 96: Model with power of two-scaling. Data types and scaling factors are indicated in brackets.

Setting offsets to zero or choosing them in an appropriate, coordinated manner eliminates some of the terms and operations above. If setting offsets to zero is not an option, then the following coordinated settings for the offsets lead to optimal code:

$f(x_1, x_2)$	Offsets
Addition	$O_y = O_{x1} + O_{x2}$
Subtraction	$O_y = O_{x1} - O_{x2}$
Multiplication	$O_y = O_{x1} \cdot O_{x2}$
Division	each $O_i = 0$ helps

13.3 Product Block with Arbitrary Scaling

If arbitrary scaling factors are chosen for the input and output variables of a *Product* block, they should be chosen such that the resulting scaling factor in the multiplication/division can be implemented easily (via bit-shifts).

Purpose

Generation of efficient code.

Remark

Unsuitable arbitrary scaling factors lead to additional operations and increase the required bit length, especially for a multiplication.

A division without offsets can be expressed as follows (scaling factors S_i):

$$y' = \frac{S_{x1}}{S_{x2} \cdot S_y} \cdot \frac{x'_1}{x'_2} = \frac{N \cdot x'_1}{D \cdot x'_2}$$

The resulting rescaling operation is expressed as a division of two integer numbers N/D . To save an additional multiplication, it is preferable to choose the scaling factors so that $D = 1$ holds true, resulting in multiplication by N only.

A multiplication without offset can be expressed as

$$y = \frac{S_{x1} \cdot S_{x2}}{S_y} \cdot x'_1 \cdot x'_2 = \frac{N}{D} \cdot (x'_1 \cdot x'_2)$$

To limit the bit length required for the numerator (3 factors), the scaling factors should be preferably set such that $N = 1$ holds true, resulting in division by D only.

References

- [TargetLink Production Code Generation Guide \[1\]](#), TargetLink Basics, Integer Calculation Basics, Implementation of Integer Calculation Operations

Example

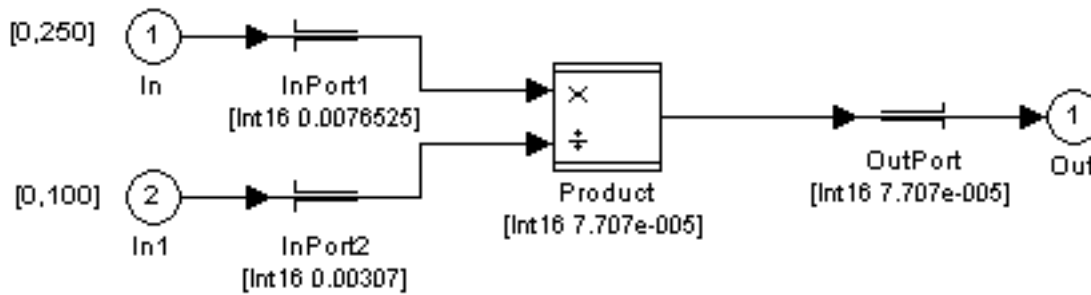


Fig. 97: Division example.

The scaling factors in fig. 97 are chosen such that the rescaling operation results in a single multiplication with a constant. Hence, multiplication in the denominator with wider bit length is avoided, leading to the following code:

```
Sal_OutPort = (Int16) (((Int32) Sal_InPort1) * 32343) / Sal_InPort2);
```

Example

The scaling factors in fig. 98 are chosen such that the scaling operation results in a single division through a constant. Hence, an additional multiplication with 64-bit data type is avoided, leading to the following code:

```
Sal_OutPort = (Int16) (((Int32) Sal_InPort1) *  
                        ((Int32) Sal_InPort2)) / 30236);
```

13.4 Constants and Multiplications with Constants

The efficiency of the code can be enhanced if gain or constant parameters of *Gain / Constant* blocks are either chosen appropriately or specified with a permitted tolerance.

Purpose

Generation of efficient code.

Remark

Real constants are implemented as divisions of two integer numbers if fixed-point code is generated. This is particularly efficient if, numerator or denominator can be expressed as a power of two, which leads to bitshift operations. If a certain tolerance is specified for the constant/gain parameter or the values are chosen appropriately, the numerator or denominator can be chosen as power of two-values by the code generator.

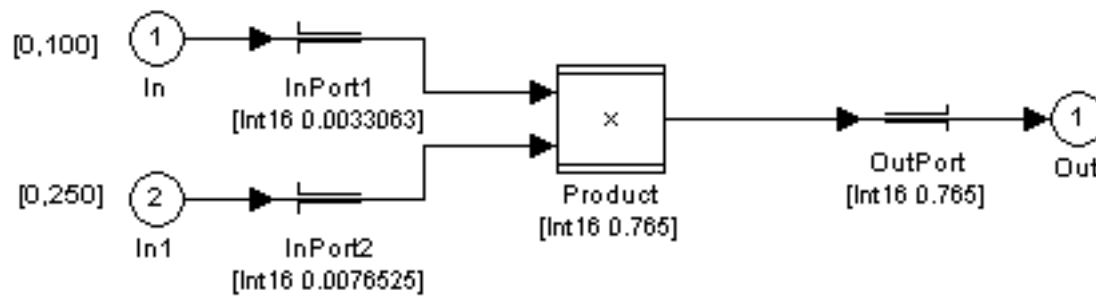


Fig. 98: Multiplication example.

References

- [TargetLink Production Code Generation Guide \[1\]](#), TargetLink Basics, Integer Calculation Basics, Implementation of Integer Calculation Operations

Example

If no permitted tolerance is specified for the gain parameter with the constant value 23.74, see fig. 99, the following code is generated:

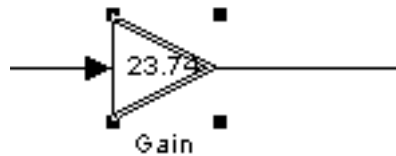


Fig. 99: A constant multiplication in a *Gain* block. Minor modifications of the gain parameter lead to more efficient code.

```
Sal_Gain = (Int16) (((Int32) Sal_InPort1) * 1187) / 50);
```

The division can be replaced by a more efficient bitshift operation if the gain parameter 23.74 can be modified to 23.75, which leads to the following code:

```
Sal_Gain = (Int16) (((Int32) Sal_InPort1) * 95) >> 2);
```

As an alternative, an acceptable deviation (tolerance level) can be specified in the block dialog, see fig. 100, which results in the following efficient code:

```
Sal_Gain = (Int16) (((Int32) Sal_InPort1) * 24309) >> 10);
```

13.5 Use of Constraint Limits

Value ranges of block outputs that are known in advance shall be specified as constraint limits in the block dialog.

Purpose

Generation of efficient code.

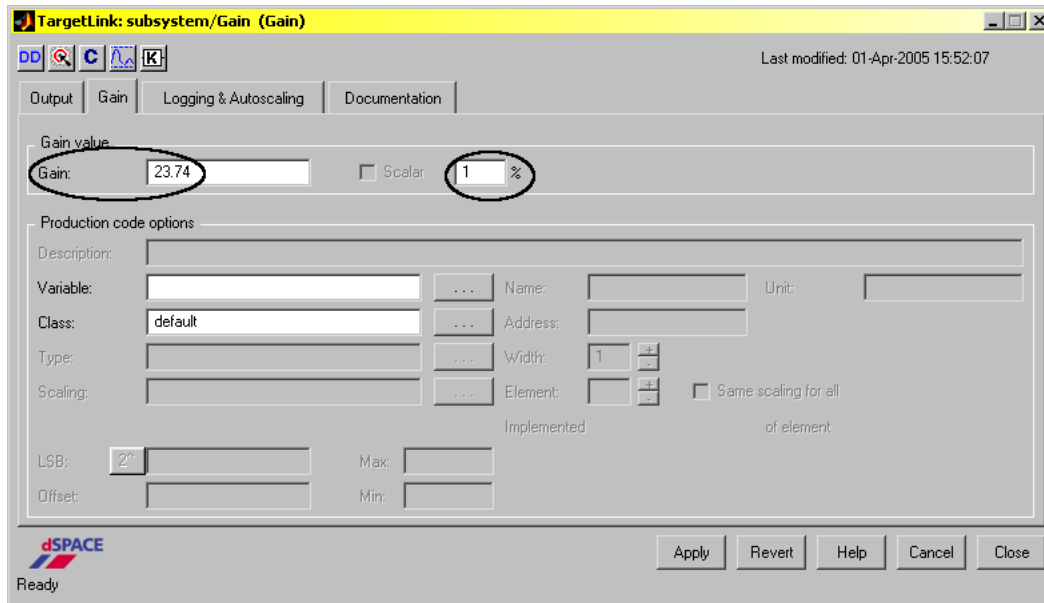


Fig. 100: Specification of an acceptable tolerance level. A deviation of 1% is the default setting for *Gain* blocks.

Remark

The code generator evaluates range information and propagates them as far as possible through the model. Giving the code generator information these ranges might prevent the necessity to use wider data types, particularly 64 bit, to avoid over/underflows.

References

- [TargetLink Production Code Generation Guide \[1\]](#), TargetLink Basics, Integer Calculation Basics, Implementation of Integer Calculation Operations, Range Determination Basics

Example

Fig. 101 shows a model with specified scaling that requires the generation of a 64-bit multiplication. It leads to the generation of the following code:

```
Int32 Aux_S32 /* LSB: 2^2 OFF: 0 MIN/MAX: -8589934592 .. 8589934588 */;
UInt32 Aux_U32 /* LSB: 2^2 OFF: 0 MIN/MAX: 0 .. 17179869180 */;

/* Product: subsystem/Product */
F__I64MULI32I32(Sal_InPort1, (Int32) Sal_InPort2, &(Aux_S32), &(Aux_U32));

/* Output: subsystem/Out */
Sal_OutPort = C__I32SHRI64C6_LT32(Aux_S32, Aux_U32, 2, 30);
```

The generation of 64-bit code contained in the called functions can be avoided by specifying constraint limits for the input variables if these are known in advance, see fig. 102. The code generator now detects that the result of the multiplication can be stored in an *Int32* variable under all circumstances, thereby avoiding the use of 64-bit arithmetic:

```
Sal_OutPort = (Int32) ((Sal_InPort1 * ((Int32) Sal_InPort2)) >> 2);
```

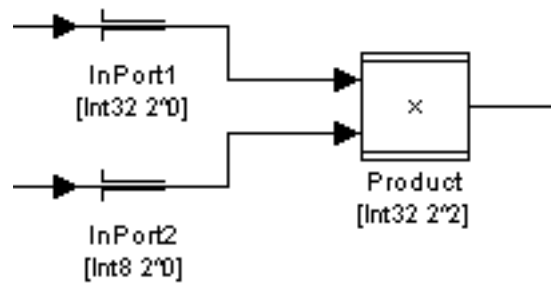
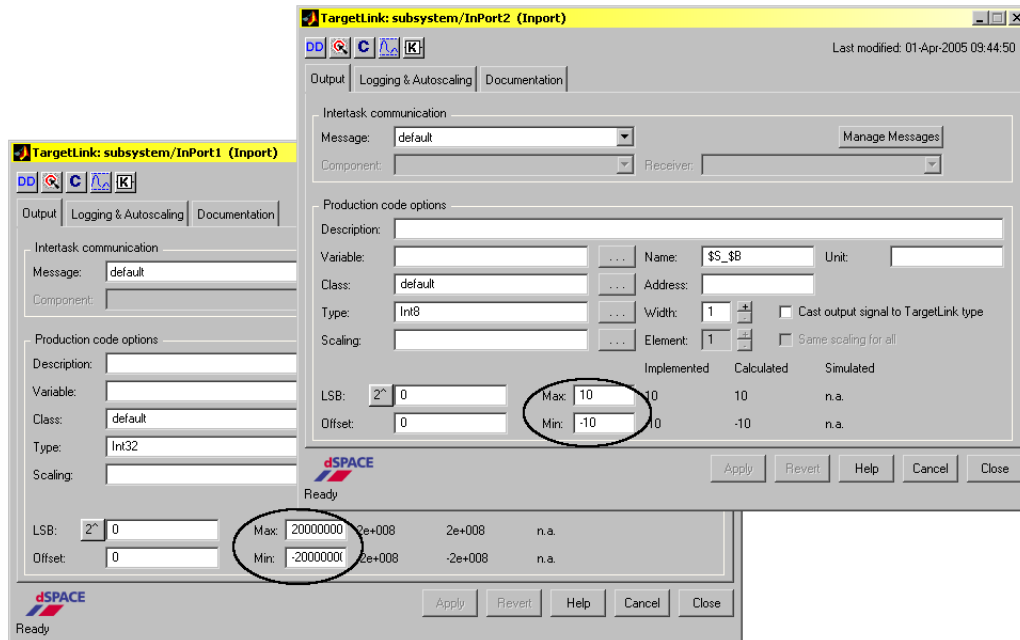


Fig. 101: Model with specified data types and scalings.

Fig. 102: Specification of constraint limits for the *Inports*.

13.6 64-Bit Operations

64-bit operations should be avoided, particularly if only ANSI C code and no target-specific code is generated.

Purpose

Generation of efficient code.

Remark

Only few compilers for microcontrollers provide support for 64-bit data types. Although TargetLink is shipped with optimized libraries for 64-bit operations, their usage leads to an increase in execution time and code size. It should be noted though that the use of 64-bit operations cannot always be avoided if wide data types are specified.

References

- [TargetLink Production Code Generation Guide \[1\]](#), TargetLink Basics, Integer Calculation Basics, Implementation of Integer Calculation Operations

Example

64 bit operations are generated by TargetLink if 32-bit data types are used as block inputs and due to enabled saturation or scalings, an intermediate 64-bit variable for the result must be introduced to prevent an undesired over- or underflow. See fig. 103 for an example.

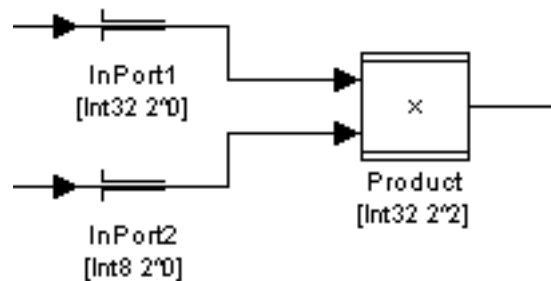


Fig. 103: Specified data types leading to the generation of an intermediate 64-bit variable. The intermediate result for the multiplication cannot be stored in a data type ≤ 32 bit.

13.7 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.

Purpose

Generation of efficient code.

Remark

The combination of variables with different scaling parameters requires additional rescaling operations, leading to additional code. Type conversions also lead to additional assembly statements by some compilers.

Example

Fig. 104 shows an example of improper scaling due to inconsistencies.

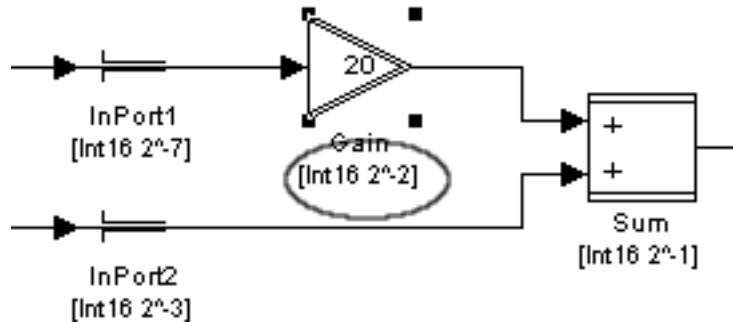


Fig. 104: Model fragment with unnecessarily inconsistent scaling that could be avoided.

Although the scaling factor of 2^{-2} at the *Gain* block is sufficient to store the result of *InPort1* multiplied by 20, it is more useful to use the scaling of the succeeding *Sum* block 2^{-1} to avoid a subsequent shift operation for rescaling.

13.8 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.

Purpose

Generation of efficient code.

Remark

The saturation of block output variables is implemented via range checking using conditional statements. This leads to increased code size and execution time.

References

- [TargetLink Production Code Generation Guide \[1\]](#), TargetLink Basics, Integer Calculation Basics, Implementation of Integer Calculation Operations

Example

If no saturation is specified for the output variable of the *Sum* block in fig. 105, the following code is generated:

```

Sal_Sum = (Int8) (((UInt8) (Int8) (Sal_InPort1 << 2)) +
                ((UInt8) (Int8) (Sal_InPort2 << 1)));
  
```

If (upper and lower) saturation is specified, the following less efficient code is generated:

```

Int16 Aux_S16;
Aux_S16 = (Int16) (((Int16) (((Int16) Sal_InPort1) << 2)) +
                ((Int16) (((Int16) Sal_InPort2) << 1)));
Sal_OutPort = ((Aux_S16 > 127) ? 127:
                ((Aux_S16 < -127) ? -127: (Int8) Aux_S16));
  
```

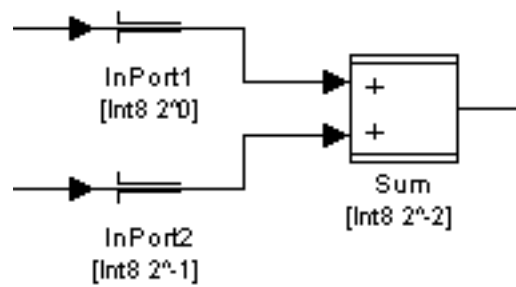


Fig. 105: *Sum* block with or without specified saturation of the output variable.

14 MISRA Compliance

This chapter contains rules relating to the compliance of TargetLink-generated code with the MISRA C 1998 standard. Since compliance with individual MISRA rules occasionally depends on code generation settings as well as the modeling style in Simulink, adherence to the rules in this chapter makes TargetLink compliant with a maximum number of MISRA rules. It should be noted, however, that the MISRA rules were developed with the primary focus on hand-programming rather than automatic code generation, and that code performance can be significantly reduced by following the guidelines in this chapter. In addition, compliance with individual MISRA rules might impose significant restrictions on the modeling style, which might be inappropriate in some cases. It should be noted also that the MISRA standard explicitly allows deviations from the standard if they are carefully thought through and documented. In some instances, TargetLink does indeed deviate from the standard for the reason of code efficiency.

Detailed MISRA compliance analysis can be found in the documents [MISRA Compliance Documentation For TargetLink 2.0 \[3\]](#) and [MISRA C:2004 Compliance Documentation For TargetLink 2.1 \[4\]](#)

14.1 Generic Code without Pragmas

For C code generation according to ISO 9899, the *Assembler and C extensions* and *Enable sections/pragmas/inline/ISR/user attributes* options should not be selected.

Purpose

MISRA C compliance.

Remark

This rule is necessary for compliance with MISRA rule 1. If one of the options is selected, TargetLink may generate code which does not conform with ISO 9899 in general.

Example

The options for generic ANSI C code generation are available in the TargetLink Main Dialog, see fig. [106](#).

14.2 Assembly Code

In order to avoid target-specific assembly code generation, the *Assembler and C extensions* checkbox in the TargetLink Main Dialog has to be cleared.

Purpose

MISRA C compliance.

Remark

Adherence to this rule makes sure that the generated code complies with MISRA rules 2 and 3. MISRA rule 3 requires that assembly code shall be used only if it is encapsulated in a C function, which TargetLink does not adhere to in the case of some macros. Clearing the *Assembler and C extensions* checkbox also makes sure that no language other than ANSI C will be generated, ensuring compliance with MISRA rule 2.

Example

The settings regarding the generation of non-ANSI C code can be found in the TargetLink Main Dialog, see fig. [107](#).

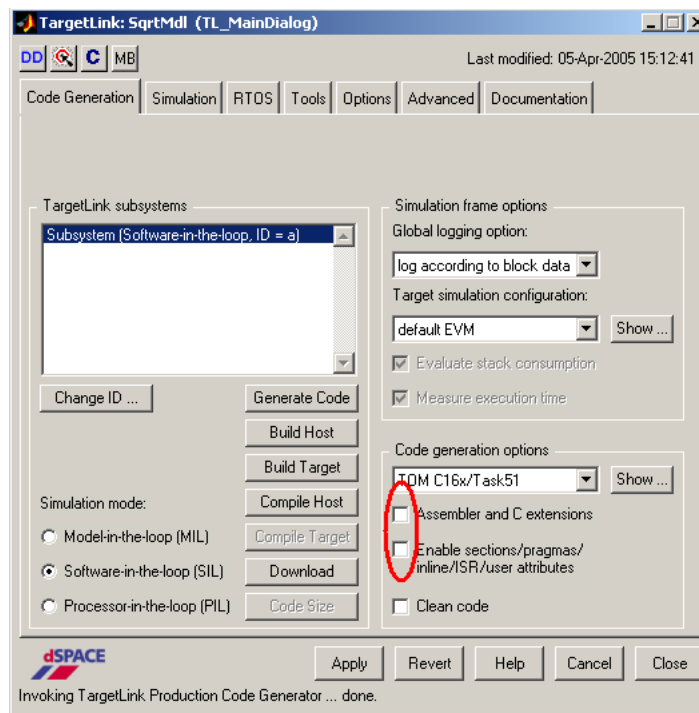


Fig. 106: Generic code without pragmas.

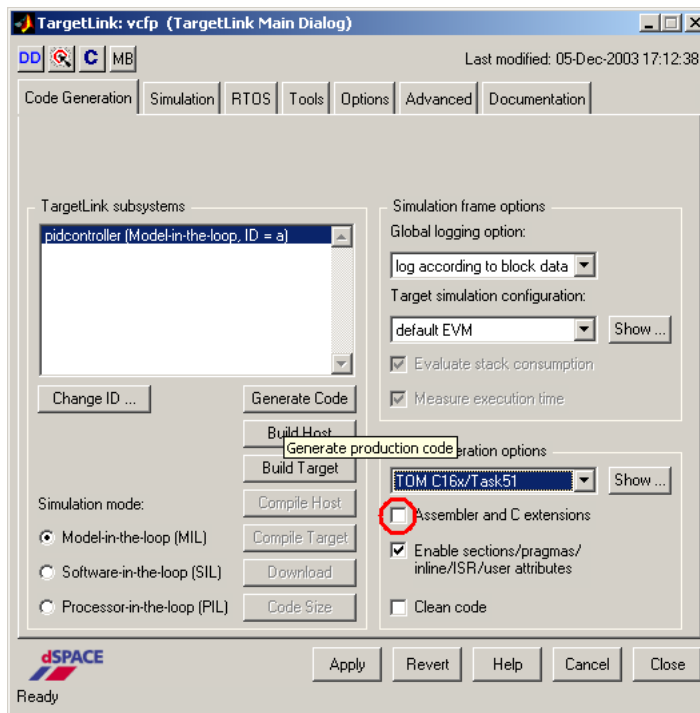


Fig. 107: The *Assembler and C extensions* option shall not be selected if the generated code has to be compliant with MISRA rules 2 and 3.

14.3 Avoidance of 64-Bit Arithmetic

64-bit arithmetic should be avoided if a high degree of compliance with individual MISRA rules has to be achieved.

Purpose

MISRA C compliance.

Remark

This rule serves to make TargetLink generated code compliant with MISRA rules 5, 45, 59, 66, 67, 90, 96, 109 and 110. Pointer casts (treated in rule 45) for instance are used in the implementations of some 64-bit multiplication macros. Rules 109/110 are not adhered to in some 64-bit divisions and multiplications because unions are used to access certain memory areas. 64-bit divisions are also not compliant with rules 66 and 67, and some 64-bit macros do not adhere to rules 90 and 96.

Note: The macros and library functions used for 64-bit arithmetic have been tested extremely thoroughly by the vendor. It is therefore extremely unlikely that noncompliance with the above rules using 64-bit arithmetic results in unsafe code.

Regardless of the MISRA compliance aspect, 64-bit operations should be avoided whenever possible to ensure efficient code.

Example

The model in fig. 108 contains a *Gain* block, whose predecessor is an *InPort* with *Int32* output, see fig. 109. The Gain parameter is set to 3/5 and the output is saturated, see fig. 110.

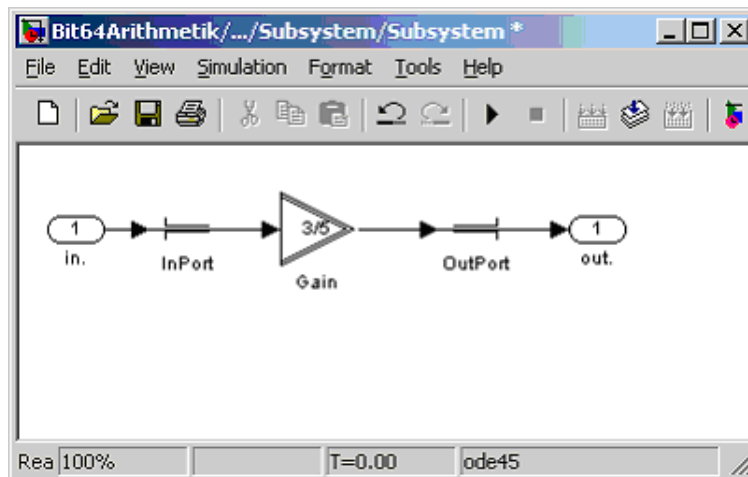


Fig. 108: Example model.

This results in the following code, containing functions for 64-bit arithmetic (F__I64MULI32U32, C__I16DIVI64C32_SAT):

```
Void Subsystem(Void)
{
    /* SLLocal: Default storage class for local variables | Width: 32 */
    Int32 Aux_S32;
    UInt32 Aux_U32;

    /* Gain: Subsystem/Gain */
    F__I64MULI32U32(Sa1_InPort, (UInt32) 3, &(Aux_S32), &(Aux_U32));
```

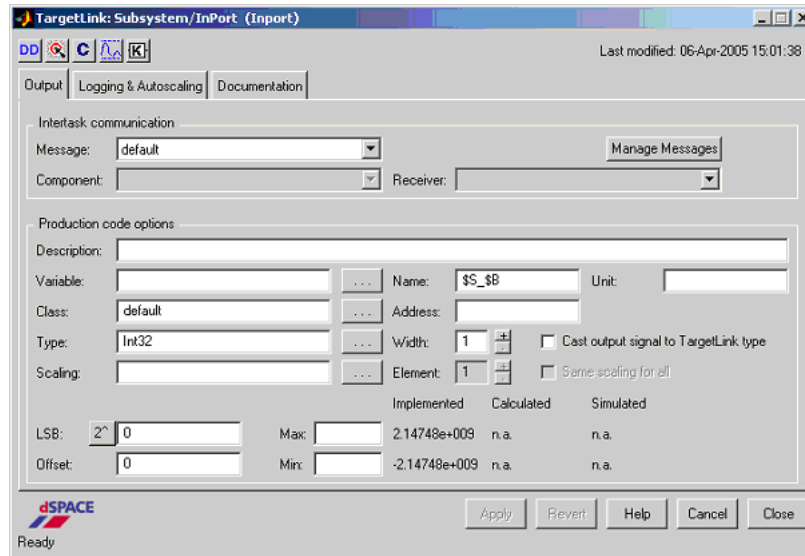


Fig. 109: Settings of the *InPort*.

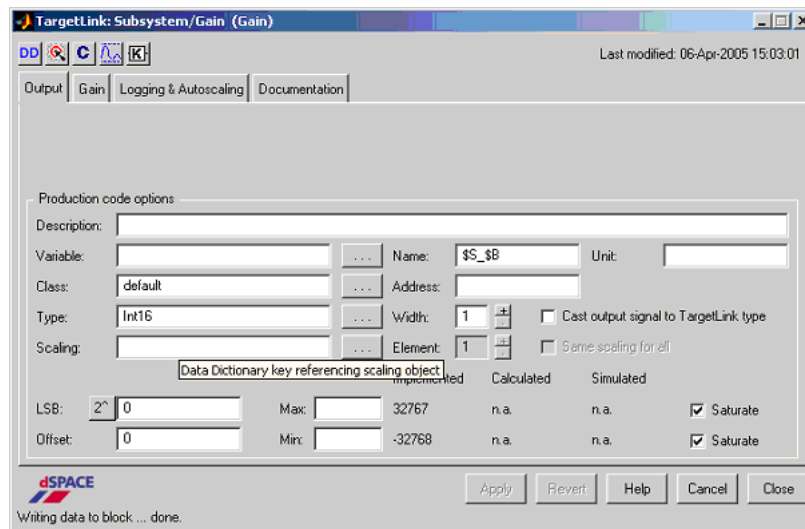


Fig. 110: Saturated output variable of the *Gain* block.

```
C__I16DIVI64C32_SAT(Aux_S32, Aux_U32, 5, 0, 163835, 32767, Sal_OutPort);
}
```

In order to avoid overflows under all circumstances, 64-bit calculation cannot be avoided entirely. However, if a constraint range is specified at the *InPort*, the calculations can be performed with a smaller bit length. In this case, even the saturation can be avoided and 64-bit operations are no longer necessary. This results in the following code without 64-bit arithmetic:

```
Void Subsystem(Void)
{
    /* Outport: Subsystem/out.
       # combined # Gain: Subsystem/Gain
       Sal_OutPort = (Int16) (((Int16) Sal_InPort) * 3) / 5);
}
```

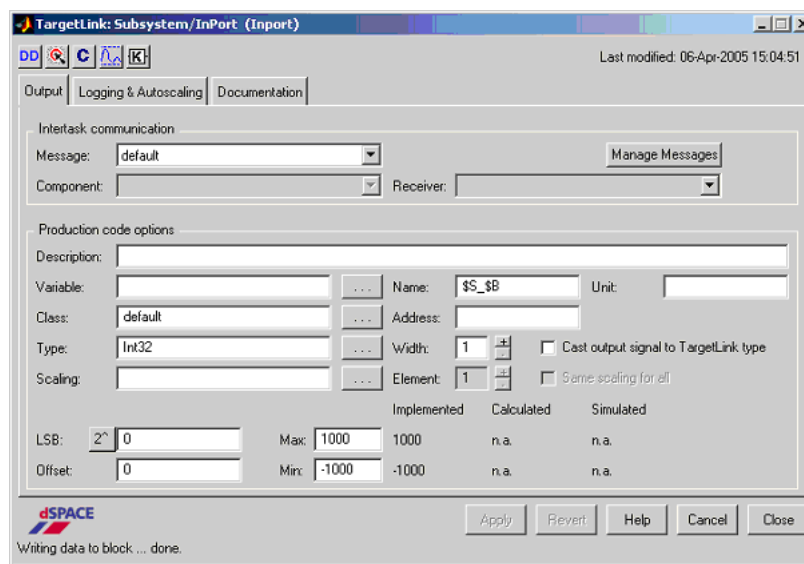


Fig. 111: Specification of *Constraint Ranges* for the *InPort* block.

14.4 Function Calls in Logical Expressions

Situations should be avoided in which optimized (erased) block output variables lead to the appearance of function calls as second operands in logical expressions.

Purpose

MISRA C compliance.

Remark

Adherence to this rule makes sure that the generated code complies with MISRA rules 33 and 46. MISRA rule 33 requires that the second operand in a logical expression is side-effect free. However, if a block output variable is erased through optimization, leading to a function call appearing as second operand in a logical expression, MISRA rule 33 is violated. This also violates MISRA rule 46, requiring that the execution order of an expression must be irrelevant.

Example

For the model in fig. 112, TargetLink generates a look-up table call, whose result is compared to the constant 1. The result is used in a logical operation.

Using a default variable class for the output of the *Look-up Table* block, see fig. 113, the following code is generated:

```
Sal_OutPort = Sal_InPort && (((Int16) Tab1DS0I2T1563_a(
&(Sal_Look_Up_Table_map), (Int8) Sal_InPort1)) > 1 /* 1. */);
```

Obviously, the function call `Tab1DS0I2T1563_a`, which is generated for the look-up table, appears as a second operand in the logical operation.

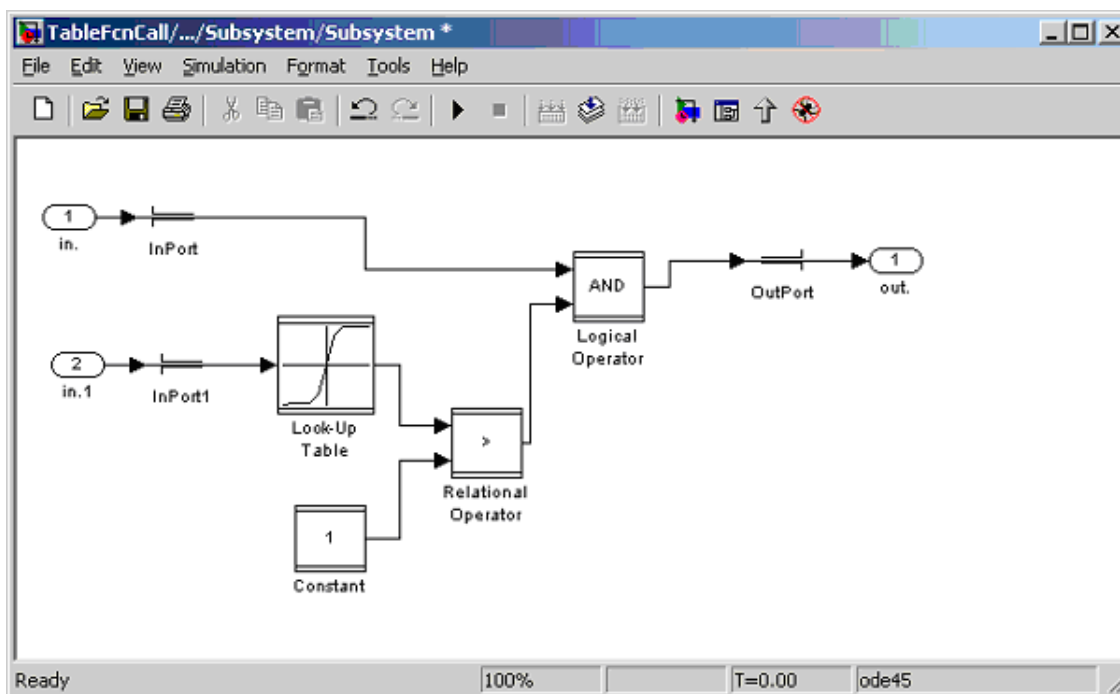


Fig. 112: Example model with a *Look-up Table* block, whose output is evaluated in a subsequent relational and logical expression.

However, if a non-optimizable variable class like *NOPT_LOCAL* is used, see fig. 114, the following code is generated:

```
Sal_Look_Up_Table = (Int16) Tab1DS0I2T1563_a(&(Sal_Look_Up_Table_map),
(Int8) Sal_InPort1);
Sal_OutPort = Sal_InPort && (Sal_Look_Up_Table > 1 /* 1. */);
```

This code fragment violates neither MISRA rule 33 nor MISRA rule 46.

14.5 Fixed-Point Sqrt

The square root function of the *Math* block should not be used in combination with fixed-point code if compliance with a maximum number of MISRA rules is desired.

Purpose

MISRA C compliance.

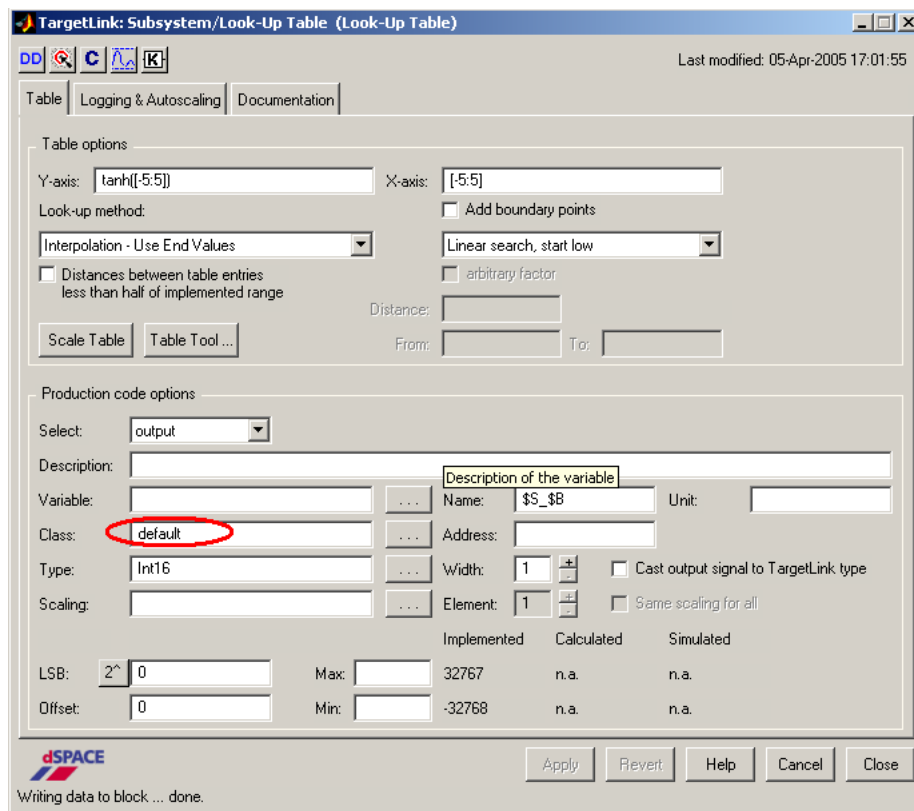


Fig. 113: A default variable class is used at the output of the Look-Up Table block. As a consequence the variable is erasable during code optimization.

Remark

Adherence to this rule serves to comply with MISRA rules 5, 59, 90 and 96, which are violated by TargetLink's fixed-point implementation of the *sqrt* function.

Note: The macros and library functions used for the *sqrt* function have been tested extremely thoroughly by the vendor. It is therefore extremely unlikely that non-compliance with the above rules results in unsafer code.

Example

The model in fig. 115 results in two generated calls to the *sqrt* functions. The upper path of the model is specified entirely for fixed-point code generation including a call a fixed-point implementation of the *sqrt* function, here `C__I16SQRTI32`. This results in the following code fragment:

```
C__I16SQRTI32((Int32) (((Int32) Sal_InPort) << 10), Sal_OutPort);
```

The lower path is partly specified for fixed-point and partly for floating-point code generation, which results in a call to the *sqrt* standard function. This ensures compliance with the MISRA rules.

```
if (Sal_InPort1 >= 0) {  
    Sal_FlpSqrt = (Float32) sqrt((Float64) Sal_InPort1);  
}  
else {  
    Sal_FlpSqrt = (Float32) (-sqrt(-(Float64) Sal_InPort1));  
}
```

14.6 FIR Filter Block

The *FIR Filter* block should not be used in combination with fixed-point code if compliance with a maximum number of MISRA rules is desired.

Purpose

MISRA C compliance.

Remark

Adherence to this rule serves to comply with MISRA rules 5, 30, 47, 59 and 74. The code for the *FIR Filter* block is implemented with macros and library functions which do not comply with the MISRA rules above.

Note: The macros and library functions used for the *FIR Filter* block have been tested extremely thoroughly by the vendor. It is therefore extremely unlikely that non-compliance with the above rules results in unsafer code.

14.7 Avoidance of Fixed-Point Code for Trigonometric Functions

Trigonometric functions should not be used in combination with fixed-point code if compliance with a maximum number of MISRA rules is desired.

Purpose

MISRA C compliance.

Remark

Adherence to this rule serves to comply with the MISRA rules 5, 45 and 74, which are partially violated by TargetLink's fixed-point implementation of trigonometric functions. The implementation of the 32-bit *Sin* and *Cos* functions contain for instance calls to 64-bit macros which violate MISRA rule 45.

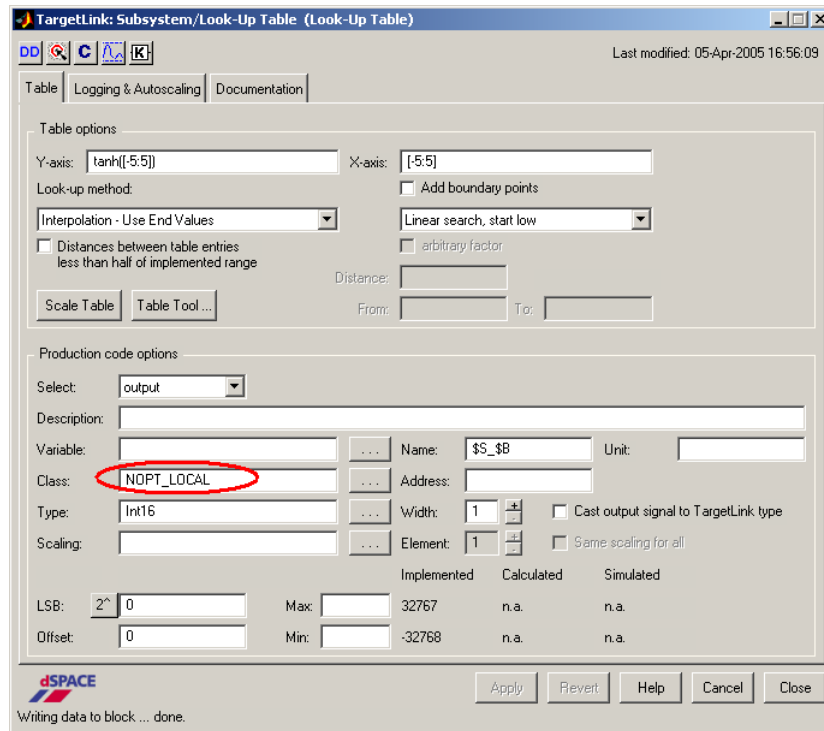


Fig. 114: The specification of a non-erasable variable class for the output of the *Look-Up Table* block.

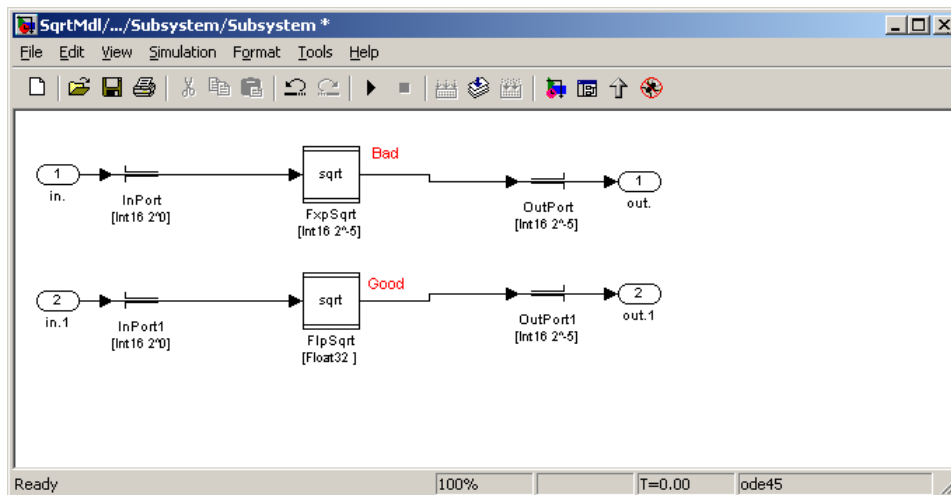


Fig. 115: Fixed-point and floating point sqrt.

MISRA rule 74 is violated by a 16-bit *Sin* function, in whose declaration and definition the parameter names differ. Since the comments of the function headers of all trigonometric functions contain an @ symbol, MISRA rule 5 is also violated.

Note: The fixed-point trigonometric functions are implemented as macros or library functions that have been tested extremely thoroughly by the vendor. It is therefore extremely unlikely that non-compliance to the above rules results in unsafer code.

14.8 Avoidance of Identical Identifier Names

Identical names for identifiers (variables, types etc.) should be avoided even if the names do not overlap.

Purpose

MISRA C compliance.

Remark

Adherence to this rule serves to comply with MISRA rules 12 and 21. TargetLink does not check whether identifiers with identical names in completely independent namespaces exist, which might result in a potential violation of MISRA rule 12. Under certain circumstances TargetLink generated code might also not comply with MISRA rule 21, which requires that identifiers with identical names shall not overlap.

Example

The model in fig. 116 contains two independent subsystems. Both contain a *Gain* block, which is named *Gain*, see fig. 117. The output variables of the *Gain* blocks are both local, each using the name macro \$B, see fig. 118. If code is generated for this model, two functions are created, each with a local variable called *Gain*. Although this is perfectly correct code, it violates MISRA rule 12.

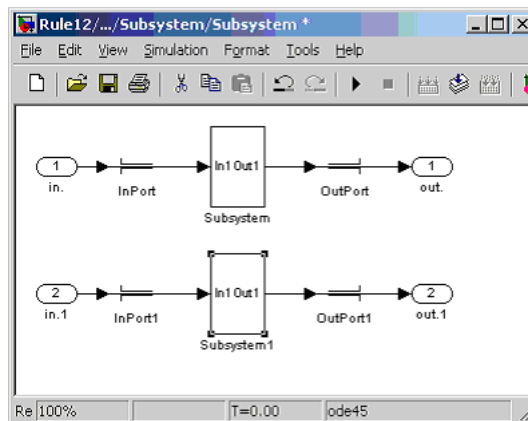


Fig. 116: Example model with two independent subsystems.

14.9 Local Macros

Local macros should not be used.

Purpose

MISRA C compliance.

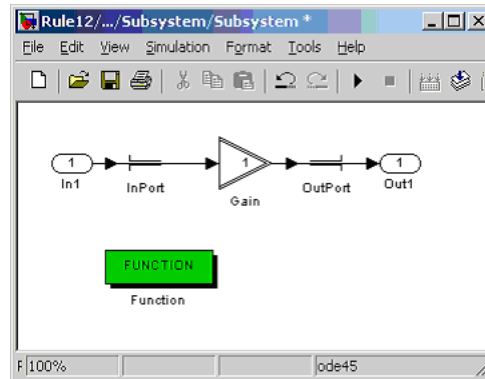


Fig. 117: A Gain block is contained in each of the two subsystems.

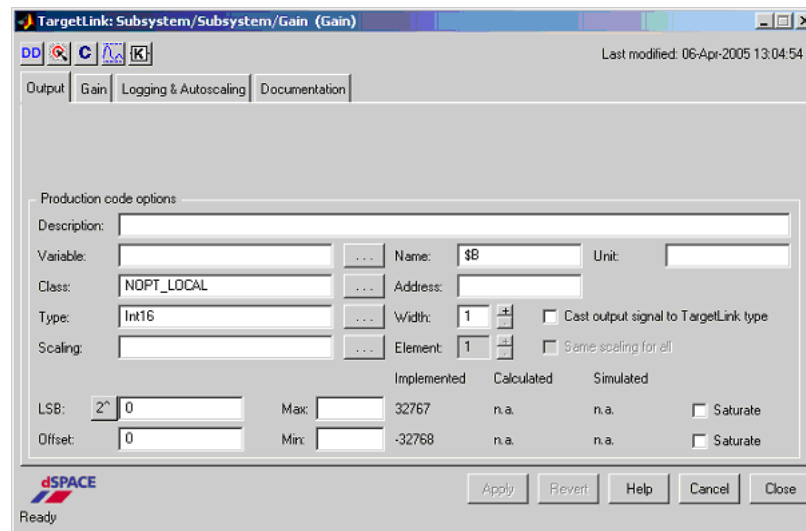


Fig. 118: Use of the name macro \$B for the output variable of the Gain block. As a consequence, the variable inherits its name Gain from the block.

Remark

Adherence to this rule serves to comply with MISRA rules 91 and 92. TargetLink generates `#define` and `#undef` preprocessor directives for local macros, which violates MISRA rules 91 and 92.

Example

The model in fig. 119 contains a Gain block, whose gain parameter's variable class is specified to be *MACRO*, see fig. 120. Because of these settings, TargetLink generates the following code, which violates the MISRA rules 91 and 92:

```
Void Rule_91_92 (Void)
{
    #define Sal_Gain_gain (Int16)1

    /* Outport: Rule_91_92/out.
       # combined # Gain: Rule_91_92/Gain */
    Sal_OutPort = Sal_InPort * Sal_Gain_gain;
    #undef Sal_Gain_gain
}
```

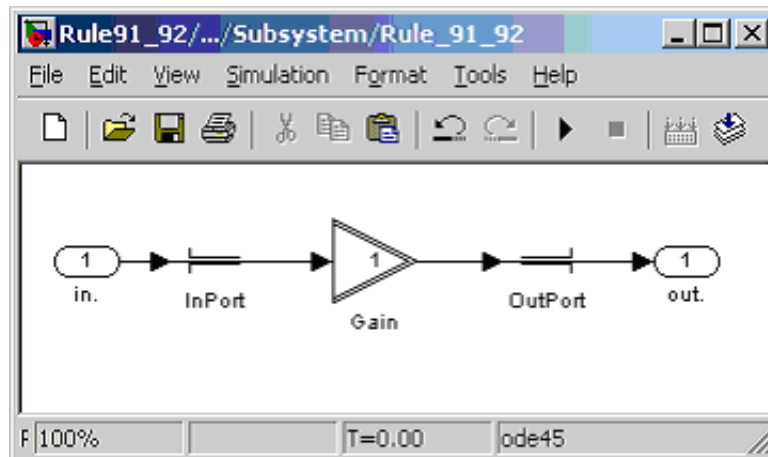


Fig. 119: Model with *Gain* block.

14.10 Use of Custom Look-up-Table Functions

A custom Look-up Table implementation should be used if compliance with a maximum number of MISRA rules is desired.

Purpose

MISRA C compliance.

Remark

This rule is necessary for compliance with MISRA rules 45, 81 and 82. Rule 45 requires that no casts to pointer types shall be used, which TargetLink's *Look-up Table* implementations do. They appear occasionally in the initialization of the so-called *map* structure or within assignments to auxiliary variables within the look-up functions. In addition, TargetLink generated Look-up Table code might violate the MISRA rules 81 (function parameters should be *const* whenever possible) and MISRA rule 82 (single exit point).

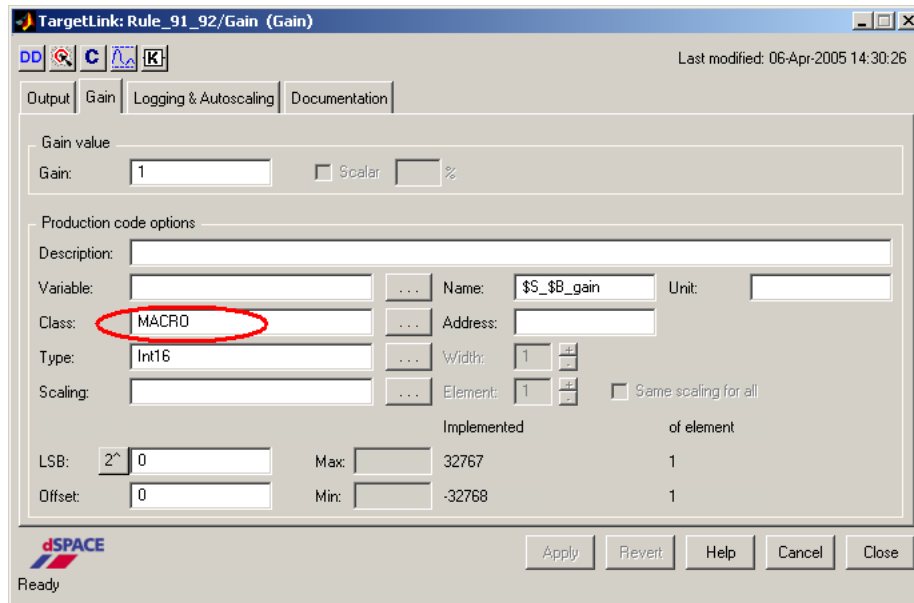


Fig. 120: The *MACRO* variable class is specified for the gain parameter. As a consequence, the gain parameter is implemented in the form of local macros using preprocessor directives.

Example

A violation of MISRA rule 45 can be found in the code generated for the TargetLink demo *Table*, model *1D Table*. The auxiliary variable `x_table` is initialized via pointer cast in a Look-up Table function, see the last line.

```
typedef struct UserRecord_tag {
    const volatile UInt16 z_table[8];
    const volatile UInt16 x_table[8];
    UInt16 Nx;
} UserRecord;

UInt16 Tab1DS0I0T521_a(const UserRecord * map, UInt16 x)
{
    ...
    const UInt16 * x_table /* Scaling may differ through function reuse. */;
    ...
    x_table = (const UInt16 *) map->x_table;
    ...
}
```

Code for the TargetLink demo *Fuelsys* contains an example of a pointer cast for the initialization of the *map* structure, which violates MISRA rule 45 (see the last lines):

```
#define MERGEABLE_CAL const volatile

typedef struct MAP_Tab2DS17I2T4169_a_tag {
    UInt8 Nx;
    UInt8 Ny;
    const UInt16 * x_table;
    const UInt16 * y_table;
```

```
    const UInt16 * z_table;
} MAP_Tab2DS17I2T4169_a;

MERGEABLE_CAL UInt16 axis_SpeedVect[18] =
{
    /*[0..9]*/ 200, 300, 400, 500, 600, 700, 800, 1000, 1200, 1400,
    /*[10..17]*/ 1600, 1800, 2000, 2400, 2800, 3200, 3600, 4000
    /* 50., 75., 100., 125., 150., 175., 200., 250., 300., 350.,
    400., 450., 500., 600., 700., 800., 900., 1000. */
}

Void IntakeAirflowEstimation(Void)
{
    ...
    /* SLStaticLocalConst: Default storage class for static local const
    variables| Width: N.A. */
    static const MAP_Tab2DS17I2T4169_a Sa2_Pumping_Constant_map = {
        18 /* Nx: */,
        19 /* Ny: */,
        (const UInt16 *)&(axis_SpeedVect[0]) /* x_table: vector with x-values */,
        (const UInt16 *)&(axis_press[0]) /* y_table: vector with y-values */,
        (const UInt16 *)&(pumpCon_z_table[0][0]) /* z_table: matrix with z-values */
    };
    ...
}
```

14.11 Multibyte Characters

Comments, for instance those entered in the *Code comment* field of block dialogs, should be free of any multibyte characters and wide-string literals.

Purpose

MISRA C compliance.

Remark

Adherence to this rule serves to comply with the MISRA rule 8. TargetLink does not check for the use of multibyte characters or wide-string literals.

14.12 Identifier Length

The length of identifiers should be restricted to 31 characters via the *Code identifiers max. 31 chars* option in the TargetLink Main Dialog.

Purpose

MISRA C compliance.

Remark

Adherence to this rule serves to comply with MISRA rule 11. MISRA rule 11 requires that identifiers like variable, function or type names are distinguishable within the first 31 characters. The option *Code identifiers max. 31 chars* makes sure that all identifiers are limited to 31 characters.

Example

The option is available in the TargetLink Main Dialog, see fig. 121.

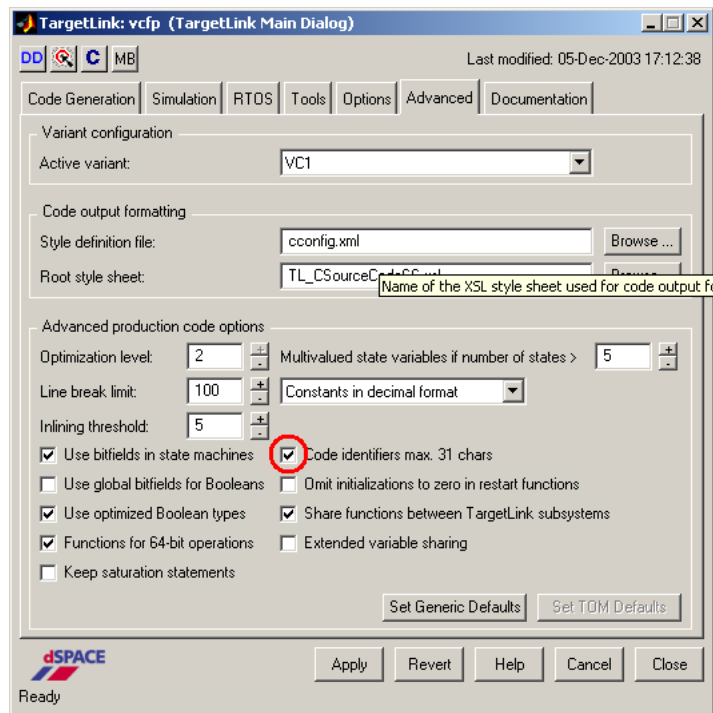


Fig. 121: Option *Code identifiers max. 31 chars*.

14.13 Use of Bitfields

The option *Use bitfields in state machines* should not be selected if compliance to MISRA rule 13 is desired.

Purpose

MISRA C compliance.

Remark

Adherence to this rule is required if the generated code must comply with MISRA rule 13. MISRA rule 13 demands that no basic types like *long*, *int*, *short* etc. shall be used. Instead, platform-independent types shall be applied, which take the bit sizes into account, like *Int8*, *Uint8*, *Int16* etc. TargetLink adheres to this rule with the exception of bitfields, which are usually *unsigned int* variables. This behavior is usually uncritical. It may be the case though, that bitfields are assigned to other non-bitfield variables which are then cast to *unsigned int*, which violates MISRA rule 13. An example would be the assignment of a bitfield for states to a *NextState* variable.

Note: For bitfield assignments in Stateflow, it can be guaranteed that no overflows occur because the value ranges for the variables are chosen properly.

Example

The option is available in the TargetLink Main Dialog, see fig. 122.

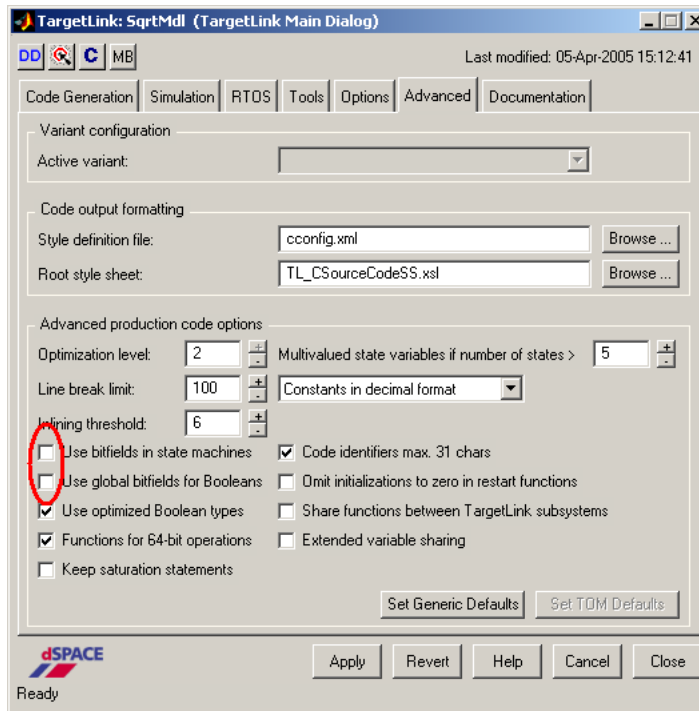


Fig. 122: Options for the use of bitfields.

14.14 Variable Scopes

Variables should be defined on the function level whenever possible. Local or *default* variable classes should be used for that purpose. If that is impossible, the selected variable class should possess the *SCOPE_REDUCIBLE* attribute for the *Optimization* property. A value for the *ScopeReducedClass* property can also be specified. If the latter is not the case, TargetLink derives a class with reduced scope on its own.

Purpose

MISRA C compliance.

Remark

Adherence to this rule serves to comply with MISRA rule 22. TargetLink's scope reduction mechanism makes sure that variables have a minimum scope (exceptions are look-up table maps and reuse structures). In order to make this mechanism work, each variable class must be scope reducible.

Example

Whether a variable is scope reducible or not depends on the settings of its variable class, see fig. 123.

14.15 External Variables

External variables should either be specified in the Data Dictionary directly or have their origin specified via the module entry.

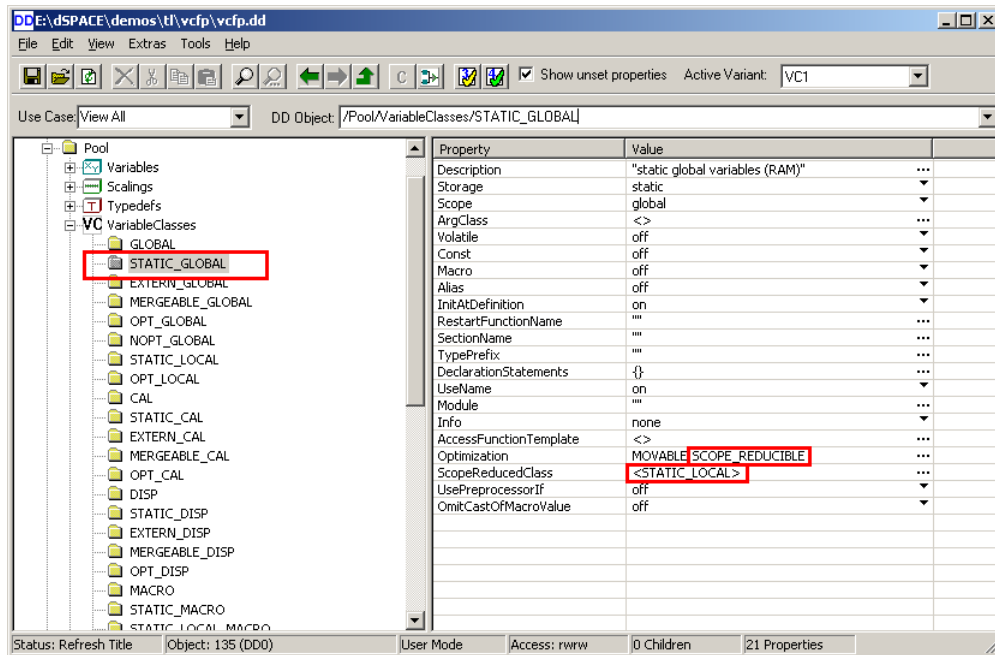


Fig. 123: Specifying the scope reduction mechanism for the `STATIC_GLOBAL` variable class for which the `SCOPE_REDUCIBLE` optimization attribute is set.

Purpose

MISRA C compliance.

Remark

Adherence to this rule serves to comply with MISRA rule 27, which requires that variables shall be declared *extern* only once. If a variable is declared as *extern* and TargetLink is unaware of its origin, a declaration for this variable might be generated on multiple occasions, not only once.

Example

The model in fig. 124 contains a gain variable with external scope, see fig. 125. TargetLink generates an extern variable declaration for the gain value into the code, which is undesired if an additional code file with an extern variable declaration is included:

```

/*****\
  EXTERN_GLOBAL: external global variables (RAM) | Width: 16
\*****\
EXTERN_GLOBAL Int16 MyGain;

```

However, if the variable for the gain value is specified in the Data Dictionary, see fig. 126, the generated code contains only an include statement for the file with the extern variable declaration for the gain value.

14.16 Uninitialized Variables and the Switch Block

The *MultiPort Switch* block should not be used if compliance to MISRA rule 30 is desired.

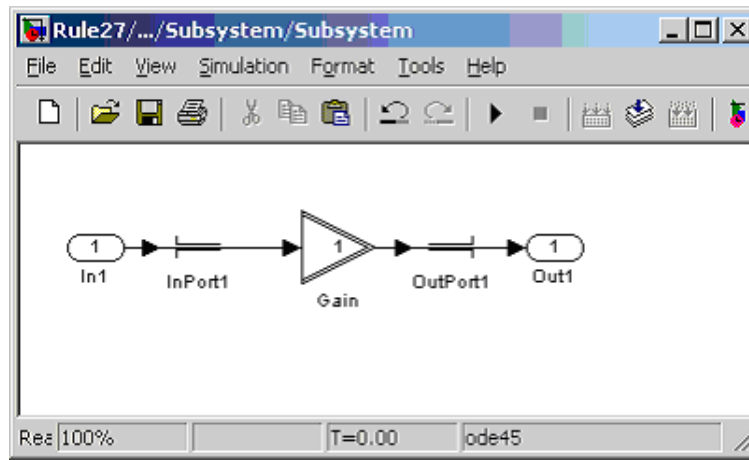


Fig. 124: Example model with an extern gain variable.

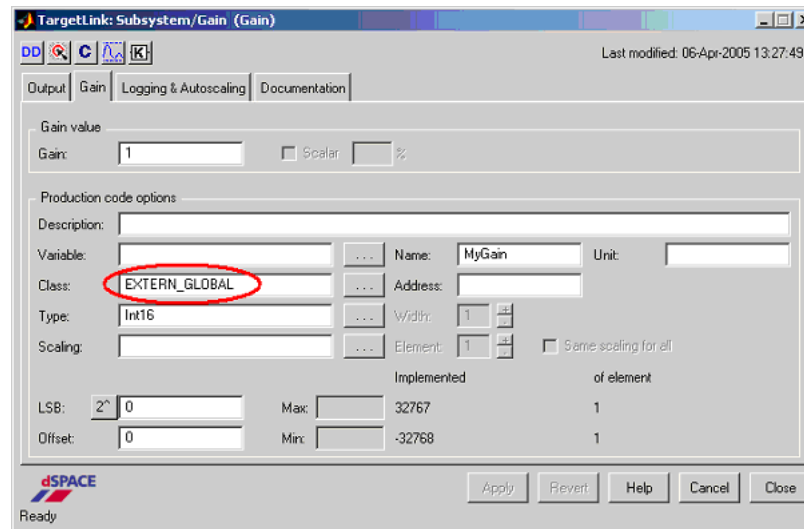


Fig. 125: An extern class is selected for the gain value. However, the variable is not explicitly specified in the Data Dictionary and TargetLink is therefore unaware of its origin.

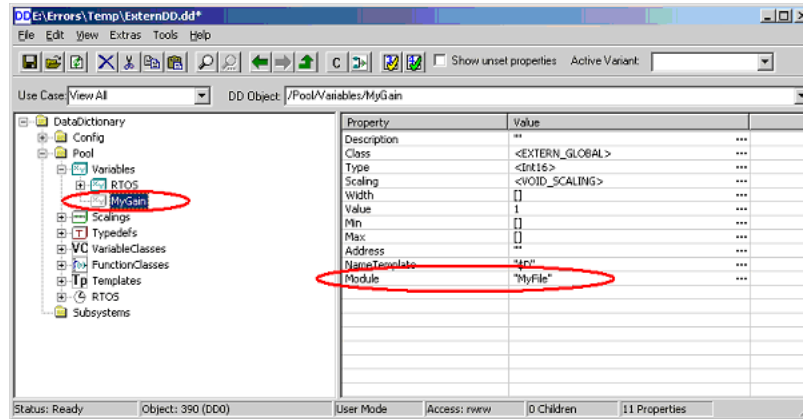


Fig. 126: Specification of the extern gain value and its origin via the *Module* property in the Data Dictionary.

Purpose

MISRA C compliance.

Remark

Adherence to this rule serves to comply with MISRA rule 30, which requires that all variables must be assigned a value before they are used. Code generated for the *Multiport Switch* block violates this rule, because there is no place to specify a default branch in the Simulink *Multiport Switch* block and TargetLink therefore simply reflects the behavior of the Simulink block.

Note: Proper modeling of the control input of the *MultiPort Switch* block ensures that its output is always defined properly. Simple MISRA checkers will nevertheless signal a violation of rule 30 though.

Example

For the example model in fig. 127 and no *constraint limits* specified for the control signal (the first input) of the *Multiport Switch* block, TargetLink generates the following code:

```
/* Multiport switch: Subsystem/Multiport Switch */
switch (Sal_InPort) {
  case 1: {
    Sal_Multiport_Switch = Sal_InPort1;
    break;
  }
  case 2: {
    Sal_Multiport_Switch = Sal_InPort2;
    break;
  }
  case 3: {
    Sal_Multiport_Switch = Sal_InPort3;
    break;
  }
  default: {
    /* Default case could be executed! In case, the output value of
       the block 'Multiport switch: Subsystem/Multiport Switch'
       will be undefined */
  }
}
```

```

}

/* Outport: Subsystem/out. */
Sal_OutPort = Sal_Multiport_Switch;

```

Should the control signal have a value unequal 1, 2 or 3, then the variable `Sal_Multiport_Switch` is not initialized before its first use in an assignment in the last line.

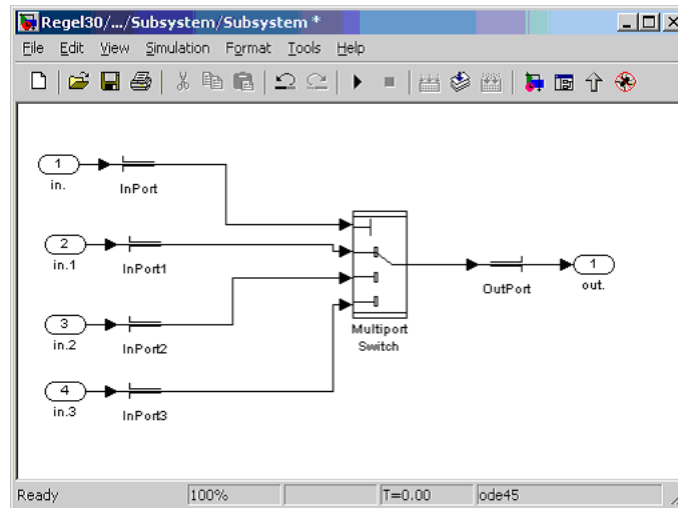


Fig. 127: Multiport Switch Block

Since the *Multiport Switch* block in Simulink provides no way to specify the behavior for a default case, it has to be ensured that the control signal is clipped to one of the permissible values using additional input logic on the modeling level.

The generation of a default branch without content in the code can be suppressed by specifying *constraint limits* for the control signal, see fig. 128, but the generated code still violates MISRA rule 30:

```

/* Multiport switch: Subsystem/Multiport Switch */
switch (Sal_InPort) {
    case 1: {
        Sal_Multiport_Switch = Sal_InPort1;
        break;
    }
    case 2: {
        Sal_Multiport_Switch = Sal_InPort2;
        break;
    }
    case 3: {
        Sal_Multiport_Switch = Sal_InPort3;
        break;
    }
}

/* Outport: Subsystem/out. */
Sal_OutPort = Sal_Multiport_Switch;

```

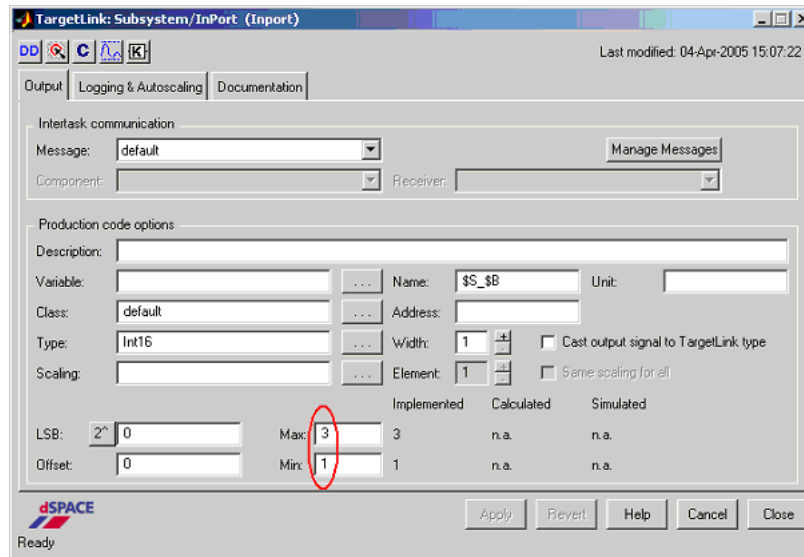


Fig. 128: A specified constraint range for the control signal indicates the set of real values to the code generator.

14.17 Avoidance of Recursions in Stateflow

To avoid recursive function calls in the code, the modeling style in Stateflow has to avoid recursive function calls in graphical functions and loops with multiple inputs and outputs. In addition, event broadcasting shall be used only in the form of *directed broadcasts* for local events whose recipients are neither the currently active state nor one of its predecessors.

Purpose

Restricted language subset for safety-critical systems, MISRA C compliance.

Remark

Adherence to this rule serves to comply with MISRA rule 70, which forbids the use of recursive functions. In general, recursive functions have to be avoided when code has to be generated for safety-critical systems.

Example

First of all, recursions can be modeled directly using *graphical functions*, see fig. 129

Secondly, if a local event is cast during the entry, execution or exiting of a state, and triggers the execution of the state itself, one of its predecessors in the hierarchy or the entire chart, a recursive function call is generated, see fig. 130. Local events should therefore be used only in the form of *directed broadcasts* whose recipients should be neither the state itself nor one of its predecessors.

Thirdly, a control flow structure for a loop leads to recursive function calls if it cannot be transformed into a *While*, *For* or *Do While* loop immediately. Examples of such situations are loops with multiple entry or exit points, see fig. 131. To avoid these recursive function calls, the design patterns for efficient control flow in this document should be used.

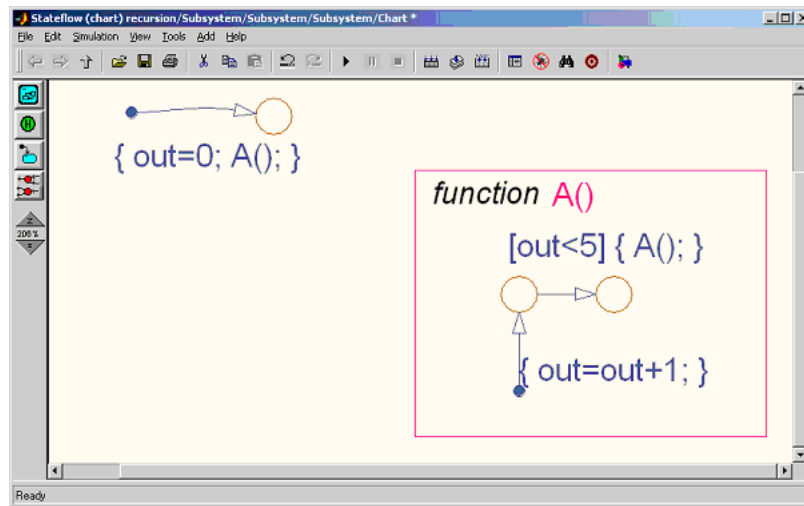


Fig. 129: Recursive function call as a result of a recursion in a graphical function.

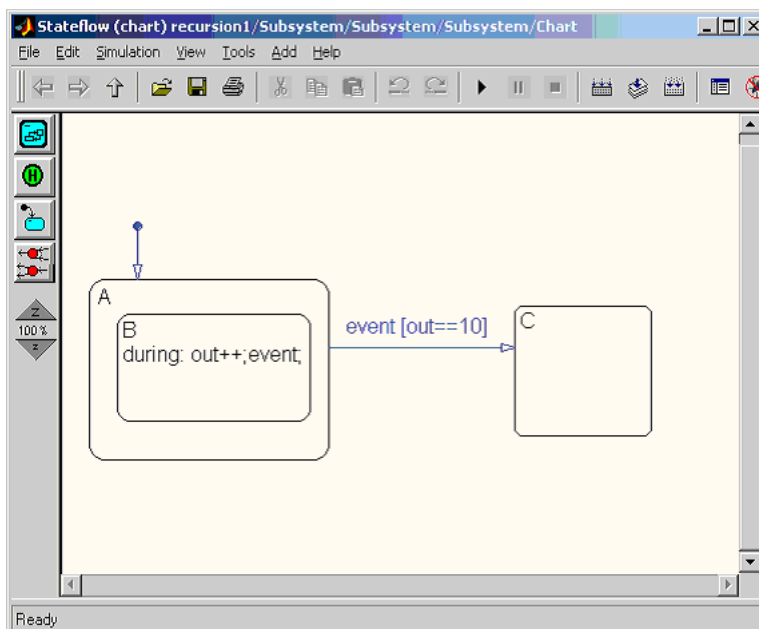


Fig. 130: Recursive function call as a result of event broadcasting.

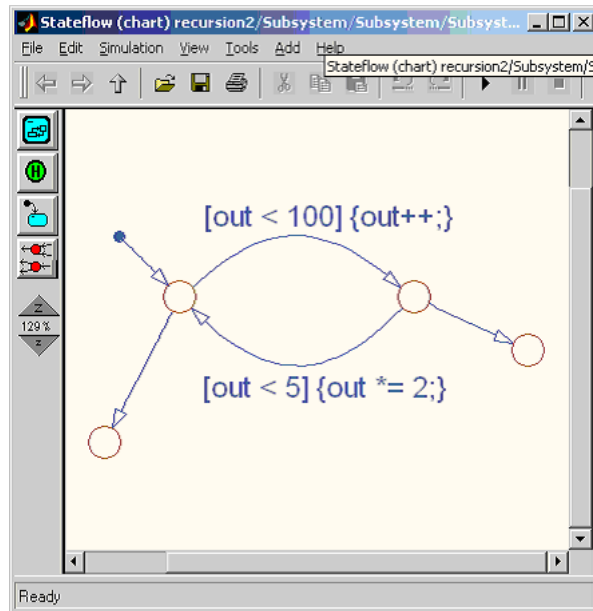


Fig. 131: Recursive function call as a result of special control flow structures.

14.18 Include Statements and Custom Code

Preprocessor include directives in *Custom Code* blocks should not be used, but replaced by *AddFile* blocks.

Purpose

MISRA C compliance.

Remark

Adherence to this rule serves to comply with MISRA rule 87. Include directives in the *Custom Code* block section are appended to the usual variable declaration section. This behavior violates MISRA rule 87, which says that only preprocessor directives may be placed before include directives.

Example

The file *MyFile.h* is included in the declaration section of a *Custom Code* block.

```
...
#####
Enter declaration code for production code simulation below
#####
/* fxp_decl_begin(top,common) */
# define DBUFLNGTH 255
# include "MyFile.h"
/* fxp_decl_end */
```

The code generated for the *Custom Code* block entails the include directive for *MyFile.h* below the variable declarations, which violates MISRA rule 87:

```
static UInt8 Sal_transport_delay_nDelays = 40 /*
    Description: delay time as multiple of sample time (0 .... 255) */;
```

```
/* Custom code: custom_demo/transport delay << declaration code (common, top) >> */
#define DBUFLNGTH 255
#include "MyFile.h"
```

However, if an *AddFile* block is used to include *MyFile.h*, see fig. 132, the include directive is generated into the proper section:

```
/*-----*\
INCLUDES
\*-----*/
#include "MyFile.h"
```

Now the generated code is compliant with MISRA rule 87.

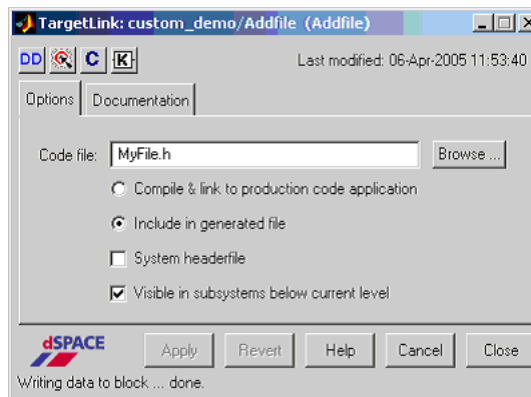


Fig. 132: Use of the *AddFile* block to include *MyFile.h*.

14.19 Avoidance of Library Function Names

No function shall have the same name as one of the library functions used.

Purpose

MISRA C compliance.

Remark

Adherence to this rules serves to comply with MISRA rule 115. TargetLink does not check whether identifiers possess identical names as library functions.

Example

Fig. 134 shows an example in which a subsystem called *sin* contains a *Function* block. If the name macro \$B is used for the TargetLink step function, this leads to the generation of a function called *sin*, which is also the name of a standard library function.

If the name of the TargetLink subsystem is **sin*, the generated function is called *_sin* because special characters are replaced by underscores. If there is a library function *_sin()*, MISRA rule 115 is violated and the code might not compile.

14.20 Volatile Variables in Logical Expressions

Situations should be avoided in which *volatile* variables appear as second operands in logical expressions if compliance to a maximum number of MISRA rules is desired.

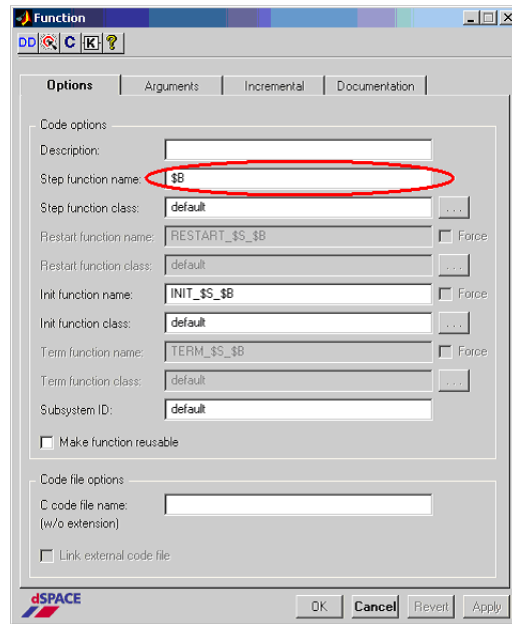


Fig. 133: Function name specified by the name macro

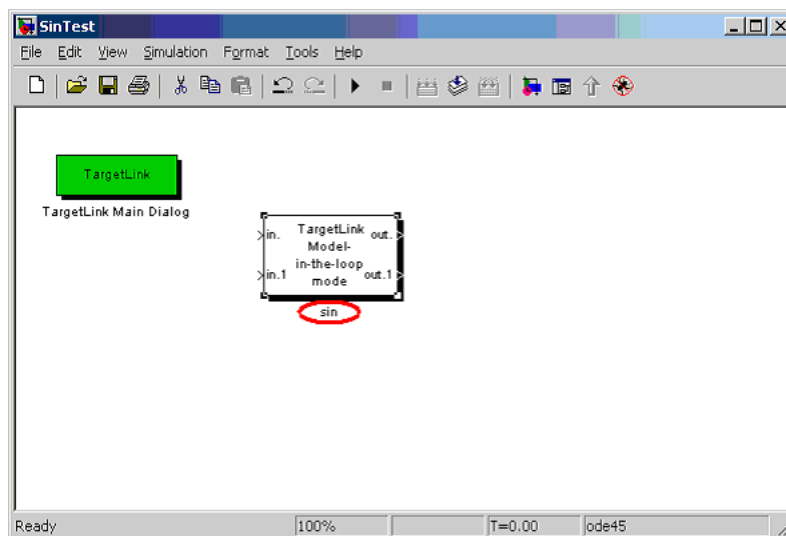


Fig. 134: TargetLink subsystem with identical name as a library function.

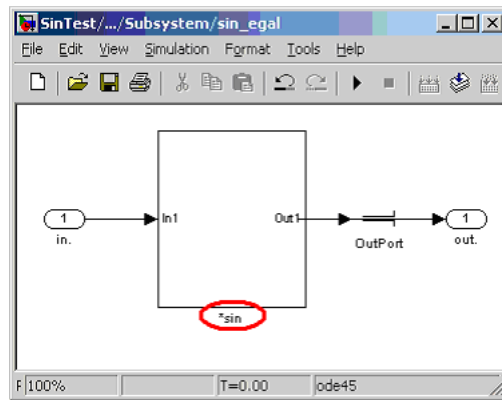


Fig. 135: Subsystem called **sin*.

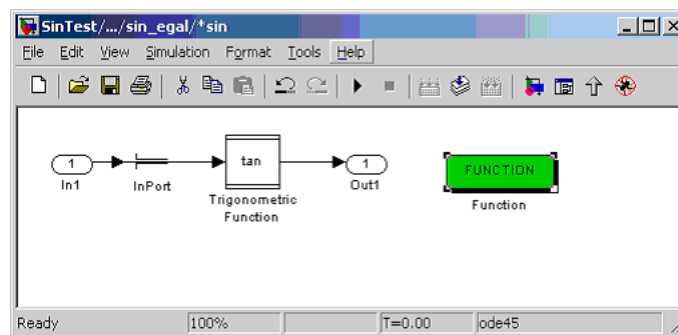


Fig. 136: Subsystem called **sin* with a *Function* block.

Purpose

MISRA C compliance.

Remark

Adherence to this rule serves to comply with MISRA rules 33 and 46. MISRA rule 33 requires that the second operand of logical expressions is side-effect free. A *volatile* variable violates this rule. In addition, MISRA rule 46 is affected which demands that the execution order must be irrelevant.

Example

In the model in fig. 137, a volatile variable (the output of the *Constant* block) is used in a comparison and a subsequent logical expression. The output of the *Constant* block is volatile, because the *STATIC_CAL* variable class has been specified for it, see fig. 138. Since the *default* variable class is chosen for the additional variables, TargetLink generates the following code:

```
Sal_OutPort = Sal_InPort && (Sal_InPort1 > Sal_Constant);
```

The volatile variable `Sal_Constant` appears as a second operand in a logical expression, thereby violating the MISRA rules 33 and 46.

If compliance to those rules is intended, a non-erasable variable should be specified for the output of the *Relational Operator* block, see fig. 139. Hence, the output variable cannot be eliminated and the volatile output of the *Constant* block does not appear in the logical expression:

```
Sal_Relational_Operator = Sal_InPort1 > Sal_Constant;
Sal_OutPort = Sal_InPort && Sal_Relational_Operator;
```

Now the generated code is compliant with MISRA rules 33 and 46.

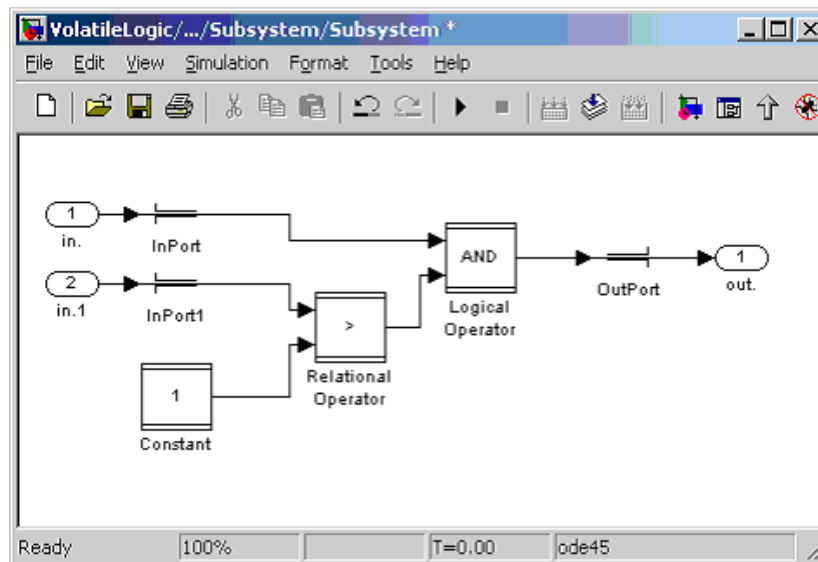


Fig. 137: Volatile constant used in a logical expression.

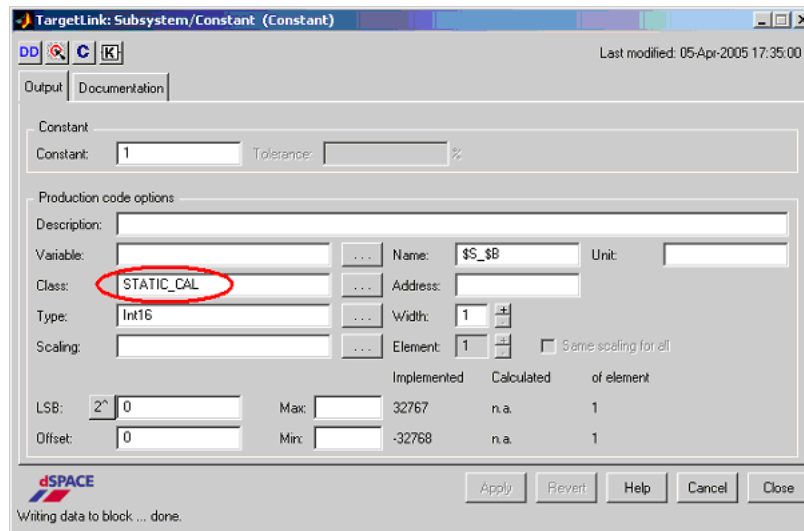


Fig. 138: The variable class for the output variable of the *Constant* block is specified as *STATIC_CAL*. The class has the volatile property set.

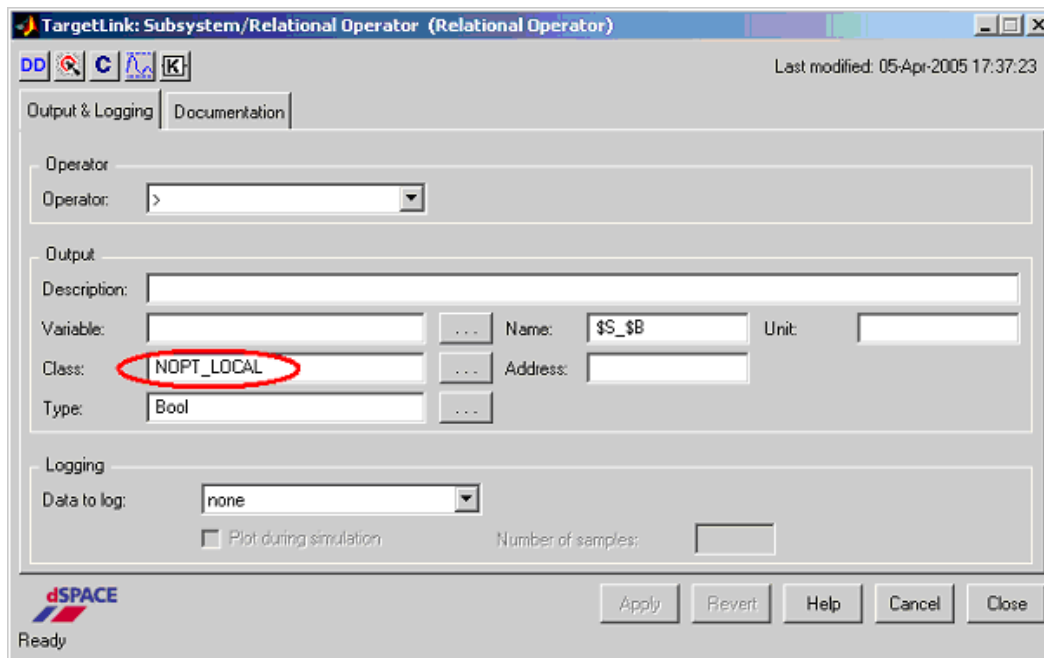


Fig. 139: Specification of a non-erasable output variable for the *Relational Operator* block. The output variable then appears in the code, preventing the volatile constant from serving as a second operand in a logical expression.

List of Documents

- [1] *TargetLink Production Code Generation Guide*, dSPACE, 2004.
- [2] *TargetLink Advanced Practices Guide*, dSPACE, 2004.
- [3] *MISRA Compliance Documentation For TargetLink 2.0*, dSPACE, 2004.
- [4] *MISRA C:2004 Compliance Documentation For TargetLink 2.1*, dSPACE, 2005.
- [5] *Using MATLAB*, The MathWorks, 2002.
- [6] *Using Simulink*, The MathWorks, 2002.
- [7] *Stateflow User's Guide*, The MathWorks, 2003.
- [8] *Controller Style Guidelines for Production Intent Using MATLAB, Simulink and Stateflow V1.0.0*, The MathWorks, 2001.