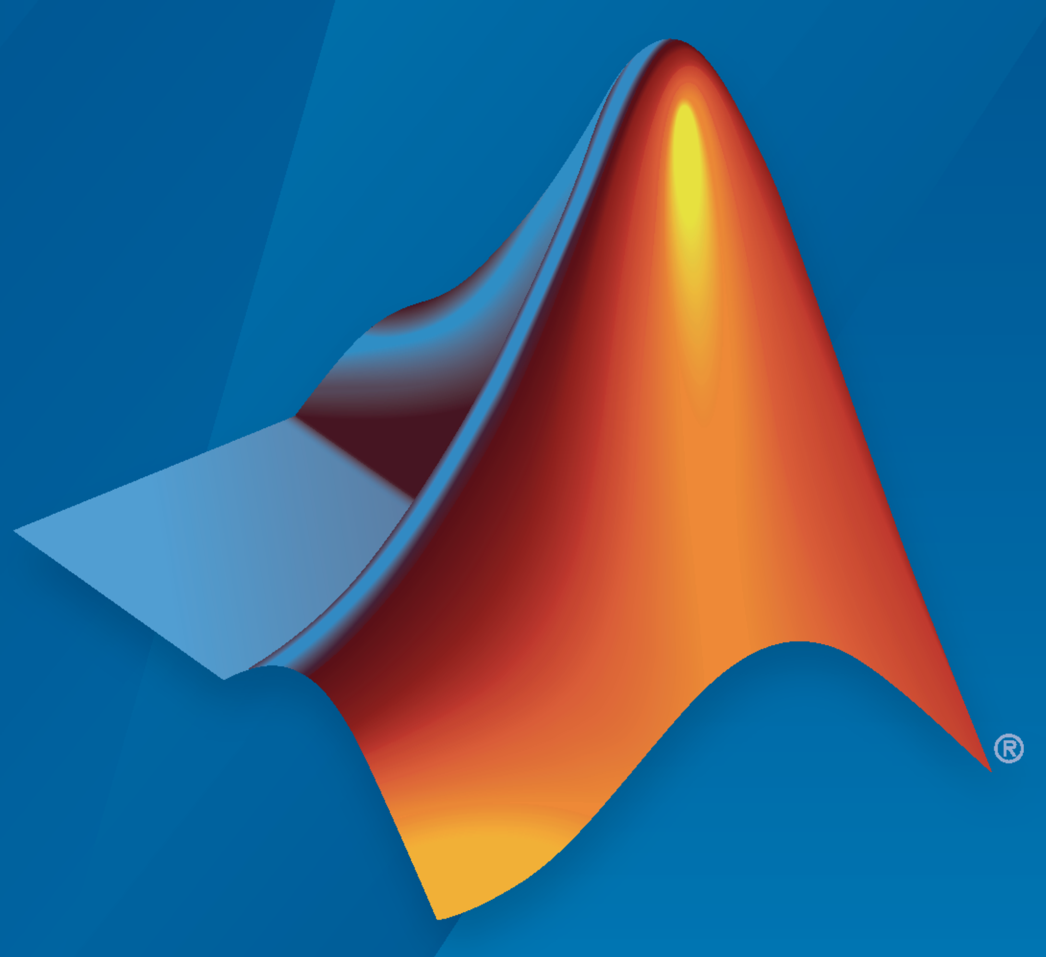
Embedded Coder®

Reference



MATLAB® & SIMULINK®

R2021a

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

Embedded Coder® Reference

© COPYRIGHT 2011–2021 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

April 2011	Online only	New for Version 6.0 (Release 2011a)
September 2011	Online only	Revised for Version 6.1 (Release 2011b)
March 2012	Online only	Revised for Version 6.2 (Release 2012a)
September 2012	Online only	Revised for Version 6.3 (Release 2012b)
March 2013	Online only	Revised for Version 6.4 (Release 2013a)
September 2013	Online only	Revised for Version 6.5 (Release 2013b)
March 2014	Online only	Revised for Version 6.6 (Release 2014a)
October 2014	Online only	Revised for Version 6.7 (Release 2014b)
March 2015	Online only	Revised for Version 6.8 (Release 2015a)
September 2015	Online only	Revised for Version 6.9 (Release 2015b)
October 2015	Online only	Rereleased for Version 6.8.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 6.10 (Release 2016a)
September 2016	Online only	Revised for Version 6.11 (Release 2016b)
March 2017	Online only	Revised for Version 6.12 (Release 2017a)
September 2017	Online only	Revised for Version 6.13 (Release 2017b)
March 2018	Online only	Revised for Version 7.0 (Release 2018a)
September 2018	Online only	Revised for Version 7.1 (Release 2018b)
March 2019	Online only	Revised for Version 7.2 (Release 2019a)
September 2019	Online only	Revised for Version 7.3 (Release 2019b)
March 2020	Online only	Revised for Version 7.4 (Release 2020a)
September 2020	Online only	Revised for Version 7.5 (Release 2020b)
March 2021	Online only	Revised for Version 7.6 (Release 2021a)

Check Bug Reports for Issues and Fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

Contents

1 Embedded Coder Functions

2 Simulink Coder Functions

3 Embedded Coder Blocks

4 Simulink Coder Blocks

5 Embedded Coder Parameters: Advanced Parameters

Use only existing shared code 5-2
Description 5-2
Settings 5-2
Dependency 5-2
Command-Line Information 5-2
Recommended Settings 5-2

Use Embedded Coder Features 5-3
Description 5-3
Settings 5-3
Dependencies 5-3
Command-Line Information 5-3

Feature set for selected hardware board 5-4
Settings 5-4

Remove reset function 5-5
Description 5-5
Settings 5-5
Dependencies 5-5

Command-Line Information 5-5

Remove disable function 5-6
Description 5-6
Settings 5-6
Dependencies 5-6
Command-Line Information 5-6

6 Code Generation Parameters: AUTOSAR

Model Configuration Parameters: Code Generation AUTOSAR 6-2

Code Generation: AUTOSAR Code Generation Options Tab Overview 6-3
Configuration 6-3
To get help on an option 6-3
Tip 6-3

Generate XML file for schema version 6-4
Description 6-4
Settings 6-4
Tips 6-4
Command-Line Information 6-4

Maximum SHORT-NAME length 6-6
Description 6-6
Settings 6-6
Tip 6-6
Command-Line Information 6-6

Use AUTOSAR compiler abstraction macros 6-7
Description 6-7
Settings 6-7
Tip 6-7
Command-Line Information 6-7

Support root-level matrix I/O using one-dimensional arrays 6-8
Description 6-8
Settings 6-8
Tips 6-8
Command-Line Information 6-8

Generate XML file for schema version 6-9
Description 6-9
Settings 6-9
Tips 6-9
Command-Line Information 6-9

Maximum SHORT-NAME length 6-10
Description 6-10
Settings 6-10
Tip 6-10

Command-Line Information 6-10

Transport layer 6-11
Description 6-11
Settings 6-11
Tip 6-11
Command-Line Information 6-11

IP address 6-12
Description 6-12
Settings 6-12
Tip 6-12
Command-Line Information 6-12

Port 6-13
Description 6-13
Settings 6-13
Tip 6-13
Command-Line Information 6-13

Verbose 6-14
Description 6-14
Settings 6-14
Tip 6-14
Command-Line Information 6-14

Use custom XCP Slave 6-15
Description 6-15
Settings 6-15
Tip 6-15
Command-Line Information 6-15

7 Code Generation Parameters: Code Placement

Model Configuration Parameters: Code Generation Code Placement 7-2

Code Generation: Code Placement Tab Overview 7-3
Configuration 7-3
To get help on an option 7-3

Data definition 7-4
Description 7-4
Settings 7-4
Dependency 7-4
Command-Line Information 7-4
Recommended Settings 7-5

Data definition filename 7-6
Description 7-6
Settings 7-6
Dependency 7-6

Command-Line Information **7-6**
Recommended Settings **7-6**

Data declaration **7-8**
Description **7-8**
Settings **7-8**
Dependency **7-8**
Command-Line Information **7-8**
Recommended Settings **7-9**

Data declaration filename **7-10**
Description **7-10**
Settings **7-10**
Dependency **7-10**
Command-Line Information **7-10**
Recommended Settings **7-10**

Use owner from data object for data definition placement **7-12**
Description **7-12**
Settings **7-12**
Command-Line Information **7-12**
Recommended Settings **7-12**

#include file delimiter **7-13**
Description **7-13**
Settings **7-13**
Dependency **7-13**
Command-Line Information **7-13**
Recommended Settings **7-13**

Signal display level **7-14**
Description **7-14**
Settings **7-14**
Dependency **7-14**
Command-Line Information **7-14**
Recommended Settings **7-14**

Parameter tune level **7-15**
Description **7-15**
Settings **7-15**
Dependency **7-15**
Command-Line Information **7-15**
Recommended Settings **7-15**

File packaging format **7-16**
Description **7-16**
Settings **7-16**
Command-Line Information **7-17**
Recommended Settings **7-17**

Header files **7-18**
Description **7-18**
Settings **7-18**
Dependency **7-18**
Command-Line Information **7-18**

Recommended Settings 7-19

Source files 7-20
Description 7-20
Settings 7-20
Dependency 7-20
Command-Line Information 7-20
Recommended Settings 7-21

Data files 7-22
Description 7-22
Settings 7-22
Dependency 7-22
Command-Line Information 7-22
Recommended Settings 7-22

Rate Transition block code 7-24
Description 7-24
Settings 7-24
Dependencies 7-24
Command-Line Information 7-24
Recommended Settings 7-24

8 Code Generation Parameters: Code Style

Model Configuration Parameters: Code Style 8-2

Code Generation: Code Style Tab Overview 8-4
Configuration 8-4
To get help on an option 8-4

Parentheses level 8-5
Description 8-5
Settings 8-5
Command-Line Information 8-5
Recommended Settings 8-5

Preserve operand order in expression 8-7
Description 8-7
Settings 8-7
Command-Line Information 8-7
Recommended Settings 8-7

Preserve condition expression in if statement 8-8
Description 8-8
Settings 8-8
Command-Line Information 8-8
Recommended Settings 8-8

Convert if-elseif-else patterns to switch-case statements 8-10
Description 8-10

Settings . . . 8-10
Command-Line Information . . . 8-11
Recommended Settings . . . 8-11

Preserve extern keyword in function declarations . . . **8-12**
Description . . . 8-12
Settings . . . 8-12
Command-Line Information . . . 8-12
Recommended Settings . . . 8-12

Preserve static keyword in function declarations . . . **8-14**
Description . . . 8-14
Settings . . . 8-14
Dependency . . . 8-14
Command-Line Information . . . 8-14
Recommended Settings . . . 8-14

Suppress generation of default cases for Stateflow switch statements if unreachable . . . **8-16**
Description . . . 8-16
Settings . . . 8-16
Command-Line Information . . . 8-16
Recommended Settings . . . 8-16

Replace multiplications by powers of two with signed bitwise shifts . . . **8-18**
Description . . . 8-18
Settings . . . 8-18
Command-Line Information . . . 8-18
Recommended Settings . . . 8-18

Allow right shifts on signed integers . . . **8-20**
Description . . . 8-20
Settings . . . 8-20
Command-Line Information . . . 8-20
Recommended Settings . . . 8-20

Casting modes . . . **8-22**
Description . . . 8-22
Settings . . . 8-22
Command-Line Information . . . 8-23
Recommended Settings . . . 8-23

Array container type . . . **8-24**
Description . . . 8-24
Settings . . . 8-24
Command-Line Information . . . 8-24
Recommended Settings . . . 8-24

Indent style . . . **8-25**
Description . . . 8-25
Settings . . . 8-25
Command-Line Information . . . 8-25
Recommended Settings . . . 8-26

Indent size . . . 8-27
Description . . . 8-27
Settings . . . 8-27
Command-Line Information . . . 8-27
Recommended Settings . . . 8-27

Newline style . . . 8-28
Description . . . 8-28
Settings . . . 8-28
Command-Line Information . . . 8-28
Recommended Settings . . . 8-28

Maximum line width . . . 8-29
Description . . . 8-29
Settings . . . 8-29
Example . . . 8-29
Command-Line Information . . . 8-29
Recommended Settings . . . 8-29

9 Code Generation Parameters: Code Generation

Model Configuration Parameters: Code Generation . . . 9-2

Set Objectives — Code Generation Advisor Dialog Box . . . 9-6
Description . . . 9-6
Settings . . . 9-6
Dependency . . . 9-7
Command-Line Information . . . 9-7

10 Code Generation Parameters: Optimization

Model Configuration Parameters: Code Generation Optimization . . . 10-2

Pass reusable subsystem outputs as . . . 10-5
Description . . . 10-5
Settings . . . 10-5
Dependencies . . . 10-5
Command-Line Information . . . 10-5
Recommended Settings . . . 10-5

Remove root level I/O zero initialization . . . 10-7
Description . . . 10-7
Settings . . . 10-7
Dependencies . . . 10-7
Command-Line Information . . . 10-8
Recommended Settings . . . 10-8

Remove internal data zero initialization **10-9**
Description **10-9**
Settings **10-9**
Dependencies **10-9**
Command-Line Information **10-10**
Recommended Settings **10-10**

Level **10-11**
Description **10-11**
Settings **10-11**
Dependencies **10-11**
Tips **10-11**
Command-Line Information **10-15**
Mapping Between Command-Line Parameter Value and UI Parameter Setting **10-15**
Recommended Settings **10-15**

Priority **10-16**
Description **10-16**
Settings **10-16**
Dependencies **10-16**
Tips **10-16**
Command-Line Information **10-19**
Recommended Settings **10-20**

Specify custom optimizations **10-21**
Description **10-21**
Settings **10-21**
Dependencies **10-21**
Tips **10-21**
Command-Line Information **10-21**
Recommended Settings **10-22**

Reuse global block outputs **10-23**
Description **10-23**
Settings **10-23**
Dependencies **10-23**
Command-Line Information **10-23**
Recommended Settings **10-23**

Perform in-place updates for Assignment and Bus Assignment blocks **10-25**
Description **10-25**
Settings **10-25**
Dependency **10-25**
Command-Line Information **10-25**
Recommended Settings **10-25**

Reuse buffers for Data Store Read and Data Store Write blocks **10-27**
Description **10-27**
Settings **10-27**
Dependency **10-27**
Command-Line Information **10-27**
Recommended Settings **10-27**

Simplify array indexing 10-29
Description 10-29
Settings 10-29
Dependencies 10-29
Command-Line Information 10-29
Recommended Settings 10-29

Pack Boolean data into bitfields 10-31
Description 10-31
Settings 10-31
Dependencies 10-31
Command-Line Information 10-31
Recommended Settings 10-31

Bitfield declarator type specifier 10-33
Description 10-33
Settings 10-33
Tip 10-33
Dependency 10-33
Command-Line Information 10-33
Recommended Settings 10-33

Reuse buffers of different sizes and dimensions 10-35
Settings 10-35
Dependencies 10-35
Tips 10-35
Command-Line Information 10-35
Recommended Settings 10-35

Generate parallel for-loops 10-37
Description 10-37
Settings 10-37
Dependency 10-37
Command-Line Information 10-37
Recommended Settings 10-37

Optimize global data access 10-38
Description 10-38
Settings 10-38
Dependencies 10-38
Command-Line Information 10-38
Recommended Settings 10-38

Optimize block operation order in the generated code 10-40
Description 10-40
Settings 10-40
Dependency 10-40
Command-Line Information 10-40
Recommended Settings 10-40

Optimize using the specified minimum and maximum values 10-42
Description 10-42
Settings 10-42
Tips 10-42
Dependencies 10-43

Command-Line Information 10-43
Recommended Settings 10-43

Remove Code from Tunable Parameter Expressions That Saturate Out-of-Range Values **10-45**
Description 10-45
Settings 10-45
Dependencies 10-45
Command-Line Information 10-45
Recommended Settings 10-45

Remove code that protects against division arithmetic exceptions . . . **10-47**
Description 10-47
Settings 10-47
Dependencies 10-47
Command-Line Information 10-47
Recommended Settings 10-47

Use signal labels to guide buffer reuse **10-49**
Description 10-49
Settings 10-49
Dependencies 10-49
Tips 10-49
Command-Line Information 10-49
Recommended Settings 10-49

Operator to represent Bitwise and Logical Operator blocks **10-51**
Description 10-51
Settings 10-51
Dependencies 10-51
Command-Line Information 10-51
Recommended Settings 10-51

11 Code Generation Parameters: Comments

Model Configuration Parameters: Comments **11-2**
Auto generated comments 11-2
Custom comments 11-2

Trace to model using **11-5**
Description 11-5
Settings 11-5
Dependency 11-5
Command-Line Information 11-5
Recommended Settings 11-5

Operator annotations **11-7**
Description 11-7
Settings 11-7
Tips 11-7
Dependency 11-7

Command-Line Information **11-7**
Recommended Settings **11-7**

Simulink block descriptions **11-9**
Description **11-9**
Settings **11-9**
Dependency **11-9**
Command-Line Information **11-9**
Recommended Settings **11-9**

Simulink data object descriptions **11-11**
Description **11-11**
Settings **11-11**
Dependency **11-11**
Command-Line Information **11-11**
Recommended Settings **11-11**

Custom comments (MPT objects only) **11-13**
Description **11-13**
Settings **11-13**
Dependency **11-13**
Command-Line Information **11-13**
Recommended Settings **11-13**

Custom comments function **11-15**
Description **11-15**
Settings **11-15**
Tip **11-15**
Dependency **11-15**
Command-Line Information **11-15**
Recommended Settings **11-15**

Stateflow object descriptions **11-17**
Description **11-17**
Settings **11-17**
Dependency **11-17**
Command-Line Information **11-17**
Recommended Settings **11-17**

Requirements in block comments **11-19**
Description **11-19**
Settings **11-19**
Dependency **11-19**
Tips **11-19**
Command-Line Information **11-19**
Recommended Settings **11-20**

MATLAB user comments **11-21**
Description **11-21**
Settings **11-21**
Dependency **11-21**
Command-Line Information **11-21**
Recommended Settings **11-21**

Comment style 11-22
Description 11-22
Settings 11-22
Dependencies 11-22
Command-Line Information 11-22
Recommended Settings 11-23

Insert Polyspace comments 11-24
Description 11-24
Settings 11-24
Dependency 11-24
Command-Line Information 11-24
Recommended Settings 11-24

12 Code Generation Parameters: Report

Model Configuration Parameters: Code Generation Report 12-2

Generate static code metrics 12-4
Description 12-4
Settings 12-4
Dependencies 12-4
Command-Line Information 12-4
Recommended Settings 12-4

Code-to-model 12-5
Description 12-5
Settings 12-5
Dependencies 12-5
Command-Line Information 12-5
Recommended Settings 12-5

Model-to-code 12-7
Description 12-7
Settings 12-7
Dependencies 12-7
Command-Line Information 12-7
Recommended Settings 12-8

Configure 12-9
Description 12-9
Dependency 12-9

Eliminated / virtual blocks 12-10
Description 12-10
Settings 12-10
Dependencies 12-10
Command-Line Information 12-10
Recommended Settings 12-10

Traceable Simulink blocks 12-12
Description 12-12
Settings 12-12
Dependencies 12-12
Command-Line Information 12-12
Recommended Settings 12-12

Traceable Stateflow objects 12-14
Description 12-14
Settings 12-14
Dependencies 12-14
Command-Line Information 12-14
Recommended Settings 12-14

Traceable MATLAB functions 12-16
Description 12-16
Settings 12-16
Dependencies 12-16
Command-Line Information 12-16
Recommended Settings 12-16

Summarize which blocks triggered code replacements 12-18
Description 12-18
Settings 12-18
Dependencies 12-18
Command-Line Information 12-18
Recommended Settings 12-18

Generate model Web view 12-20
Description 12-20
Settings 12-20
Dependencies 12-20
Command-Line Information 12-20
Recommended Settings 12-20

13 Code Generation Parameters: Custom Code

Model Configuration Parameters: Code Generation Custom Code 13-2

14 Model Configuration Parameters: Code Generation Interface

Model Configuration Parameters: Code Generation Interface 14-2

Support: floating-point numbers 14-8
Description 14-8
Settings 14-8
Dependencies 14-8

Command-Line Information **14-8**
Recommended Settings **14-8**

Support: complex numbers **14-10**
Description **14-10**
Settings **14-10**
Dependencies **14-10**
Command-Line Information **14-10**
Recommended Settings **14-10**

Support: absolute time **14-11**
Description **14-11**
Settings **14-11**
Dependencies **14-11**
Command-Line Information **14-11**
Recommended Settings **14-11**

Support: continuous time **14-13**
Description **14-13**
Settings **14-13**
Dependencies **14-13**
Command-Line Information **14-14**
Recommended Settings **14-14**

Support: variable-size signals **14-15**
Description **14-15**
Settings **14-15**
Dependencies **14-15**
Command-Line Information **14-15**
Recommended Settings **14-15**

Pass root-level I/O as **14-16**
Description **14-16**
Settings **14-16**
Dependencies **14-16**
Command-Line Information **14-16**
Recommended Settings **14-16**

Remove error status field in real-time model data structure **14-18**
Description **14-18**
Settings **14-18**
Dependencies **14-18**
Command-Line Information **14-18**
Recommended Settings **14-18**

Include model types in model class **14-20**
Description **14-20**
Settings **14-20**
Dependencies **14-20**
Command-Line Information **14-20**
Recommended Settings **14-20**

Ignore custom storage classes **14-22**
Description **14-22**
Settings **14-22**

Tips 14-22
Dependencies 14-22
Command-Line Information 14-22
Recommended Settings 14-22

Ignore test point signals **14-24**
Description 14-24
Settings 14-24
Dependencies 14-24
Command-Line Information 14-24
Recommended Settings 14-24

Support non-inlined S-functions **14-26**
Description 14-26
Settings 14-26
Tip 14-26
Dependencies 14-26
Command-Line Information 14-26
Recommended Settings 14-26

Use dynamic memory allocation for model initialization **14-28**
Description 14-28
Settings 14-28
Dependencies 14-28
Command-Line Information 14-28
Recommended Settings 14-28

Use dynamic memory allocation for model block instantiation **14-30**
Description 14-30
Settings 14-30
Dependencies 14-30
Command-Line Information 14-30
Recommended Settings 14-31

Terminate function required **14-32**
Description 14-32
Settings 14-32
Dependencies 14-32
Command-Line Information 14-32
Recommended Settings 14-32

Combine signal/state structures **14-33**
Description 14-33
Settings 14-33
Tips 14-33
Dependencies 14-34
Command-Line Information 14-34
Recommended Settings 14-34

Implement each data store block as a unique access point **14-35**
Description 14-35
Settings 14-35
Dependencies 14-35
Command-Line Information 14-35
Recommended Settings 14-35

Generate separate internal data per entry-point function **14-36**
Description **14-36**
Settings **14-36**
Tips **14-36**
Dependencies **14-37**
Command-Line Information **14-37**
Recommended Settings **14-37**

Multiword type definitions **14-39**
Description **14-39**
Settings **14-39**
Tips **14-39**
Dependencies **14-40**
Command-Line Information **14-40**
Recommended Settings **14-40**

Generate destructor **14-41**
Description **14-41**
Settings **14-41**
Dependencies **14-41**
Command-Line Information **14-41**
Recommended Settings **14-41**

MAT-file variable name modifier **14-42**
Description **14-42**
Settings **14-42**
Dependency **14-42**
Command-Line Information **14-42**
Recommended Settings **14-42**

Existing shared code **14-43**
Description **14-43**
Settings **14-43**
Command-Line Information **14-43**
Recommended Settings **14-43**

15 Code Generation Parameters: Identifiers

Model Configuration Parameters: Code Generation Identifiers **15-2**

Global variables **15-4**
Description **15-4**
Settings **15-4**
Tips **15-4**
Dependency **15-5**
Command-Line Information **15-5**
Recommended Settings **15-5**

Global types **15-6**
Description **15-6**
Settings **15-6**

Tips 15-6
Dependency 15-7
Command-Line Information 15-7
Recommended Settings 15-7

Field name of global types 15-8
Description 15-8
Settings 15-8
Tips 15-8
Dependency 15-8
Command-Line Information 15-9
Recommended Settings 15-9

Subsystem methods 15-10
Description 15-10
Settings 15-10
Tips 15-10
Dependency 15-11
Command-Line Information 15-11
Recommended Settings 15-11

Subsystem method arguments 15-12
Description 15-12
Settings 15-12
Tips 15-12
Dependencies 15-12
Command-Line Information 15-13
Recommended Settings 15-13

Local temporary variables 15-14
Description 15-14
Settings 15-14
Tips 15-14
Dependency 15-15
Command-Line Information 15-15
Recommended Settings 15-15

Local block output variables 15-16
Description 15-16
Settings 15-16
Tips 15-16
Dependency 15-16
Command-Line Information 15-16
Recommended Settings 15-17

Constant macros 15-18
Description 15-18
Settings 15-18
Tips 15-18
Dependency 15-19
Command-Line Information 15-19
Recommended Settings 15-19

Shared utilities identifier format 15-20
Description 15-20

Settings 15-20
Tips 15-20
Dependency 15-21
Command-Line Information 15-21
Recommended Settings 15-21

Minimum mangle length **15-22**
Description 15-22
Settings 15-22
Tips 15-22
Dependency 15-22
Command-Line Information 15-22
Recommended Settings 15-22

System-generated identifiers **15-24**
Description 15-24
Settings 15-24
Dependencies 15-27
Command-Line Information 15-27
Recommended Settings 15-28

Generate scalar inlined parameters as **15-29**
Description 15-29
Settings 15-29
Dependencies 15-29
Command-Line Information 15-29
Recommended Settings 15-29
Improve Code Readability by Generating Block Parameter Values as Macros 15-29

Signal naming **15-32**
Description 15-32
Settings 15-32
Dependencies 15-32
Limitation 15-32
Command-Line Information 15-32
Recommended Settings 15-33

M-function **15-34**
Description 15-34
Settings 15-34
Tip 15-34
Dependencies 15-34
Command-Line Information 15-35
Recommended Settings 15-35

Parameter naming **15-36**
Description 15-36
Settings 15-36
Dependencies 15-36
Limitation 15-36
Command-Line Information 15-36
Recommended Settings 15-36

M-function 15-38
Description 15-38
Settings 15-38
Tip 15-38
Dependencies 15-38
Command-Line Information 15-39
Recommended Settings 15-39

#define naming 15-40
Description 15-40
Settings 15-40
Dependencies 15-40
Command-Line Information 15-40
Recommended Settings 15-40

M-function 15-42
Description 15-42
Settings 15-42
Tip 15-42
Dependencies 15-42
Command-Line Information 15-43
Recommended Settings 15-43

Custom token text 15-44
Description 15-44
Settings 15-44
Dependencies 15-44
Command-Line Information 15-44
Recommended Settings 15-44

Shared checksum length 15-45
Description 15-45
Settings 15-45
Tip 15-45
Dependencies 15-45
Command-Line Information 15-45
Recommended Settings 15-45

EMX array utility functions identifier format 15-46
Description 15-46
Settings 15-46
Tips 15-46
Dependencies 15-46
Command-Line Information 15-47
Recommended Settings 15-47

EMX array types identifier format 15-48
Description 15-48
Settings 15-48
Tips 15-48
Dependencies 15-48
Command-Line Information 15-49
Recommended Settings 15-49

16 Code Generation Parameters: Data Type Replacement

Model Configuration Parameters: Code Generation Data Type Replacement **16-2**
Configure Data Type Replacements Programmatically **16-2**

Code Generation: Data Type Replacement Tab **16-4**
Configuration **16-4**
To get help on an option **16-4**

Replace data type names in the generated code **16-5**
Description **16-5**
Settings **16-5**
Dependencies **16-5**
Command-Line Information **16-5**
Recommended Settings **16-6**

Replacement Name: double **16-7**
Description **16-7**
Settings **16-7**
Dependency **16-7**
Command-Line Information **16-7**
Recommended Settings **16-8**

Replacement Name: single **16-9**
Description **16-9**
Settings **16-9**
Dependency **16-9**
Command-Line Information **16-9**
Recommended Settings **16-10**

Replacement Name: int32 . . . . . **16-11**
Description **16-11**
Settings **16-11**
Dependency **16-11**
Command-Line Information **16-11**
Recommended Settings **16-11**

Replacement Name: int16 . . . . . **16-13**
Description **16-13**
Settings **16-13**
Dependency **16-13**
Command-Line Information **16-13**
Recommended Settings **16-13**

Replacement Name: int8 . . . . . **16-15**
Description **16-15**
Settings **16-15**
Dependency **16-15**
Command-Line Information **16-15**
Recommended Settings **16-15**

Replacement Name: uint32 16-17
Description 16-17
Settings 16-17
Dependency 16-17
Command-Line Information 16-17
Recommended Settings 16-17

Replacement Name: uint16 16-19
Description 16-19
Settings 16-19
Dependency 16-19
Command-Line Information 16-19
Recommended Settings 16-19

Replacement Name: uint8 16-21
Description 16-21
Settings 16-21
Dependency 16-21
Command-Line Information 16-21
Recommended Settings 16-21

Replacement Name: boolean 16-23
Description 16-23
Settings 16-23
Dependency 16-23
Command-Line Information 16-23
Recommended Settings 16-24

Replacement Name: int 16-25
Description 16-25
Settings 16-25
Dependency 16-25
Command-Line Information 16-25
Recommended Settings 16-26

Replacement Name: uint 16-27
Description 16-27
Settings 16-27
Dependency 16-27
Command-Line Information 16-27
Recommended Settings 16-28

Replacement Name: char 16-29
Description 16-29
Settings 16-29
Dependency 16-29
Command-Line Information 16-29
Recommended Settings 16-29

Replacement Name: uint64 16-31
Description 16-31
Settings 16-31
Dependency 16-31
Command-Line Information 16-31
Recommended Settings 16-32

Replacement Name: int64 16-33
Description 16-33
Settings 16-33
Dependency 16-33
Command-Line Information 16-33
Recommended Settings 16-34

17 Memory Sections Parameters on the Code Generation Pane

Code Generation: Memory Sections Tab Overview 17-2
Configuration 17-2
To get help on an option 17-2

Package 17-3
Description 17-3
Settings 17-3
Tip 17-3
Command-Line Information 17-3
Recommended Settings 17-3

Refresh package list 17-5
Description 17-5
Tip 17-5

Initialize/Terminate 17-6
Description 17-6
Settings 17-6
Command-Line Information 17-6
Recommended Settings 17-6

Execution 17-7
Description 17-7
Settings 17-7
Command-Line Information 17-7
Recommended Settings 17-7

Shared utility 17-8
Description 17-8
Settings 17-8
Command-Line Information 17-8
Recommended Settings 17-8

Constants 17-9
Description 17-9
Settings 17-9
Command-Line Information 17-9
Recommended Settings 17-9

Inputs/Outputs 17-11
Description 17-11
Settings 17-11

Command-Line Information 17-11
Recommended Settings 17-11

Internal data 17-13
Description 17-13
Settings 17-13
Command-Line Information 17-13
Recommended Settings 17-13

Parameters 17-15
Description 17-15
Settings 17-15
Command-Line Information 17-15
Recommended Settings 17-15

Validation results 17-16
Description 17-16
Settings 17-16
Recommended Settings 17-16

18 Code Generation Parameters: Templates

Model Configuration Parameters: Code Generation Templates 18-2

Code Generation: Templates Tab Overview 18-3
Configuration 18-3
To get help on an option 18-3

Code templates: Source file (*.c) template 18-4
Description 18-4
Settings 18-4
Command-Line Information 18-4
Recommended Settings 18-4

Code templates: Header file (*.h) template 18-5
Description 18-5
Settings 18-5
Command-Line Information 18-5
Recommended Settings 18-5

Data templates: Source file (*.c) template 18-6
Description 18-6
Settings 18-6
Command-Line Information 18-6
Recommended Settings 18-6

Data templates: Header file (*.h) template 18-7
Description 18-7
Settings 18-7
Command-Line Information 18-7
Recommended Settings 18-7

File customization template **18-8**
Description **18-8**
Settings **18-8**
Command-Line Information **18-8**
Recommended Settings **18-8**

Generate an example main program **18-9**
Description **18-9**
Settings **18-9**
Tips **18-9**
Dependencies **18-9**
Command-Line Information **18-10**
Recommended Settings **18-10**

Target operating system **18-11**
Description **18-11**
Settings **18-11**
Dependencies **18-11**
Command-Line Information **18-11**
Recommended Settings **18-11**

19 Code Generation Parameters: Verification

Model Configuration Parameters: Code Generation Verification **19-2**

Code Generation: Verification Tab Overview **19-3**
Configuration **19-3**
To get help on an option **19-3**

Measure task execution time **19-4**
Description **19-4**
Settings **19-4**
Dependencies **19-4**
Command-Line Information **19-4**
Recommended Settings **19-4**

Measure function execution times **19-6**
Description **19-6**
Settings **19-6**
Dependencies **19-6**
Command-Line Information **19-6**
Recommended Settings **19-6**

Workspace variable **19-8**
Description **19-8**
Settings **19-8**
Dependency **19-8**
Command-Line Information **19-8**
Recommended Settings **19-8**

Save options 19-9
Description 19-9
Settings 19-9
Dependency 19-9
Command-Line Information 19-10
Recommended Settings 19-10

Third-party tool 19-11
Description 19-11
Settings 19-11
Dependencies 19-11
Command-Line Information 19-11
Recommended Settings 19-11

Enable portable word sizes 19-13
Description 19-13
Settings 19-13
Dependencies 19-13
Command-Line Information 19-13
Recommended Settings 19-13

Enable source-level debugging for SIL 19-15
Description 19-15
Settings 19-15
Command-Line Information 19-15
Recommended Settings 19-15

Create block 19-16
Description 19-16
Settings 19-16
Command-Line Information 19-16
Recommended Settings 19-16

20 Configuration Parameters

Recommended Settings Summary for Model Configuration Parameters 20-2

21 Parameters for Creating Protected Models

Create Protected Model 21-2
Create Protected Model: Overview 21-2
Open read-only view of model 21-3
Simulate 21-3
Use generated code 21-4
Code interface 21-4
Content type 21-5

Use generated HDL code **21-6**
Destination folder **21-6**
Contents **21-7**
Create harness model for protected model **21-7**
Name of project archive (.mlproj) **21-8**

22 Model Advisor Checks

Embedded Coder Checks **22-2**
Embedded Coder Checks Overview **22-2**
Check for blocks not recommended for C/C++ production code deployment **22-3**
Check configuration parameters for generation of inefficient saturation code **22-4**
Identify lookup table blocks that generate expensive out-of-range checking code **22-5**
Check output types of logic blocks **22-6**
Check the hardware implementation **22-7**
Identify questionable software environment specifications **22-8**
Identify questionable code instrumentation (data I/O) **22-9**
Identify blocks generating inefficient algorithms **22-10**
Check configuration parameters for MISRA C:2012 . . . . **22-11**
Check for blocks not recommended for MISRA C:2012 . . . . **22-13**
Check for unsupported block names **22-15**
Check usage of Assignment blocks **22-15**
Check for switch case expressions without a default case **22-17**
Check for missing error ports for AUTOSAR receiver interfaces **22-18**
Check bus object names that are used as bus element names **22-19**
Check configuration parameters for secure coding standards **22-19**
Check for blocks not recommended for secure coding standards **22-21**
Identify questionable subsystem settings **22-22**
Check for blocks not supported for row-major code generation **22-23**
Identify TLC S-Functions with unset array layout **22-24**
Identify blocks that generate expensive fixed-point and saturation code **22-24**
Check for missing const qualifiers in model functions **22-26**
Identify questionable fixed-point operations **22-27**
Identify blocks that generate expensive rounding code **22-29**
Check for bitwise operations on signed integers **22-29**
Check for recursive function calls **22-30**
Check for equality and inequality operations on floating-point values . . **22-30**
Check integer word length **22-31**
Check block names **22-32**

23 Embedded Coder Tools

24 C/C++ Functions That Support Symbolic Dimensions for Simulink Function Blocks

25 Apps

Embedded Coder Functions

activateConfigSet

Activate configuration set of model

Syntax

`cgvObj.activateConfigSet(configSetName)`

Description

`cgvObj``.activateConfigSet(configSetName)` specifies the active configuration set for the model, only while the model is executed by *`cgvObj`*. *`cgvObj`* is a handle to a `cgv.CGV` object. `configSetName` is the name of a configuration set object, `Simulink.ConfigSet`, which already exists in the model. The original configuration set for the model is restored after execution of the `cgv.CGV` object.

See Also

Topics
"Activate a Configuration Set"
"Programmatic Code Generation Verification"

addAdditionalHeaderFile

Add header file to array of header files for code replacement table entry

Syntax

`addAdditionalHeaderFile(hEntry,headerFile)`

Description

`addAdditionalHeaderFile(hEntry,headerFile)` adds a specified additional header file to the array of additional header files for a code replacement table entry.

This function adds `-I` to the compile line in the generated makefile.

Examples

Specify Additional Header and Source Files

This example shows how to use the `addAdditionalHeaderFile` function with `addAdditionalIncludePath`, `addAdditionalSourceFile`, and `addAdditionalSourcePath` to specify additional header and source files fully for a code replacement table entry.

```
% Path to external header and source files
libdir = fullfile('$(MATLAB_ROOT)','..', '..', 'lib');

op_entry = RTW.TflCOperationEntry;
.
.
.
addAdditionalHeaderFile(op_entry, 'all_additions.h');
addAdditionalIncludePath(op_entry, fullfile(libdir, 'include'));

addAdditionalSourceFile(op_entry, 'all_additions.c');
addAdditionalSourcePath(op_entry, fullfile(libdir, 'src'));
```

Input Arguments

`hEntry` — Handle to a code replacement table entry
handle

`hEntry` is a handle to a code replacement table entry previously returned by instantiating a code replacement table entry class, such as *`hEntry`* `= RTW.TflCFunctionEntry` or *`hEntry`* `= RTW.TflCOperationEntry`.

Example: `op_entry`

`headerFile` — Name of additional header file
character vector | string scalar

`headerFile` is a character vector or string scalar that specifies an additional header file.

Example: 'all_additions.h'

See Also

`addAdditionalIncludePath` | `addAdditionalSourceFile` | `addAdditionalSourcePath`

Topics

"Define Code Replacement Library Optimizations"

Introduced in R2007b

addAdditionalIncludePath

Add include path to array of include paths for code replacement table entry

Syntax

`addAdditionalIncludePath(hEntry,path)`

Description

`addAdditionalIncludePath(hEntry,path)` adds a specified additional include path to the array of additional include paths for a code replacement table entry.

This function adds `-I` to the compile line in the generated makefile.

Examples

Specify Path to Additional Header and Source Files

This example shows how to use the `addAdditionalIncludePath` function with `addAdditionalHeaderFile`, `addAdditionalSourceFile`, and `addAdditionalSourcePath` to specify the path to additional header and source files fully for a code replacement table entry.

```
% Path to external header and source files
libdir = fullfile('$(MATLAB_ROOT)','..', '..', 'lib');

op_entry = RTW.TflCOperationEntry;
.
.
.
addAdditionalHeaderFile(op_entry, 'all_additions.h');
addAdditionalIncludePath(op_entry, fullfile(libdir, 'include'));

addAdditionalSourceFile(op_entry, 'all_additions.c');
addAdditionalSourcePath(op_entry, fullfile(libdir, 'src'));
```

Input Arguments

`hEntry` — Handle to a code replacement table entry
handle

The *`hEntry`* is a handle to a code replacement table entry previously returned by instantiating a code replacement table entry class, such as *`hEntry`* `= RTW.TflCFunctionEntry` or *`hEntry`* `= RTW.TflCOperationEntry`.

Example: `op_entry`

`path` — Path to an additional header file
character vector | string scalar

The *`path`* is a character vector or string scalar that specifies the full path to an additional header file. The character vector or string scalar can include tokens (for example, `$myfolder$`, where

`myfolder` is a variable defined as a character vector, cell array of character vectors, or string array in the MATLAB® workspace).

Example: `fullfile(libdir,'include')`

See Also

`addAdditionalHeaderFile` | `addAdditionalSourceFile` | `addAdditionalSourcePath`

Topics

"Define Code Replacement Library Optimizations"

Introduced in R2007b

addAdditionalLinkObj

Add link object to array of link objects for code replacement table entry

Syntax

`addAdditionalLinkObj(hEntry,linkObj)`

Description

`addAdditionalLinkObj(hEntry,linkObj)` adds a specified additional link object to the array of additional link objects for a code replacement table entry.

Examples

Specify an Additional Link Object

This example shows how to use the `addAdditionalLinkObj` function with `addAdditionalLinkObjPath` to specify an additional link object file fully for a code replacement table entry.

```
% Path to external object files
libdir = fullfile('$(MATLAB_ROOT)','..', '..', 'lib');

op_entry = RTW.TflCOperationEntry;
...
addAdditionalLinkObj(op_entry, 'addition.o');
addAdditionalLinkObjPath(op_entry, fullfile(libdir, 'bin'));
```

Input Arguments

`hEntry` — Handle to a code replacement table entry
handle

The *`hEntry`* is a handle to a code replacement table entry previously returned by instantiating a code replacement table entry class, such as *`hEntry`* `= RTW.TflCFunctionEntry` or *`hEntry`* `= RTW.TflCOperationEntry`.

Example: `op_entry`

`linkObj` — Name of an additional link object
character vector | string scalar

The *`linkObj`* is a character vector or string scalar that specifies an additional link object.

Example: `'addition.o'`

See Also

`addAdditionalLinkObjPath`

Topics
"Define Code Replacement Library Optimizations"

Introduced in R2007b

addAdditionalLinkObjPath

Add link object path to array of link object paths for code replacement table entry

Syntax

`addAdditionalLinkObjPath(hEntry,path)`

Description

`addAdditionalLinkObjPath(hEntry,path)` adds a specified additional link object path to the array of additional link object paths for a code replacement table entry.

Examples

Specify Path to Additional Link Object

This example shows how to use the `addAdditionalLinkObjPath` function with `addAdditionalLinkObj` to specify the path to an additional link object file fully for a code replacement table entry.

```
% Path to external object files
libdir = fullfile('$(MATLAB_ROOT)','..', '..', 'lib');

op_entry = RTW.TflCOperationEntry;
...
addAdditionalLinkObj(op_entry, 'addition.o');
addAdditionalLinkObjPath(op_entry, fullfile(libdir, 'bin'));
```

Input Arguments

`hEntry` — Handle to a code replacement table entry
handle

The *`hEntry`* is a handle to a code replacement table entry previously returned by instantiating a code replacement entry class, such as *`hEntry`* `= RTW.TflCFunctionEntry` or *`hEntry`* `= RTW.TflCOperationEntry`.

Example: `op_entry`

`path` — Path to an additional link object
character vector | string scalar

The *`path`* is a character vector or string scalar that specifies the full path to an additional link object. The character vector or string scalar can include tokens (for example, `$myfolder$`, where `myfolder` is a variable defined as a character vector, cell array of character vectors, or string array in the MATLAB workspace).

Example: `op_entry`

See Also
`addAdditionalLinkObj`

Topics
"Define Code Replacement Library Optimizations"

Introduced in R2008a

addAdditionalSourceFile

Add source file to array of source files for code replacement table entry

Syntax

`addAdditionalSourceFile(hEntry,sourceFile)`

Description

`addAdditionalSourceFile(hEntry,sourceFile)` adds a specified additional source file to the array of additional source files for a code replacement table entry.

This function adds `-I` to the compile line in the generated makefile.

Examples

Specify Additional Header and Source Files

This example shows how to use the `addAdditionalSourceFile` function with `addAdditionalHeaderFile`, `addAdditionalIncludePath`, and `addAdditionalSourcePath` to specify additional header and source files fully for a code replacement table entry.

```
% Path to external header and source files
libdir = fullfile('$(MATLAB_ROOT)','..', '..', 'lib');

op_entry = RTW.TflCOperationEntry;
.
.
.
addAdditionalHeaderFile(op_entry, 'all_additions.h');
addAdditionalIncludePath(op_entry, fullfile(libdir, 'include'));

addAdditionalSourceFile(op_entry, 'all_additions.c');
addAdditionalSourcePath(op_entry, fullfile(libdir, 'src'));
```

Input Arguments

`hEntry` — Handle to a code replacement table entry
handle

The *`hEntry`* is a handle to a code replacement table entry previously returned by instantiating a code replacement table entry class, such as *`hEntry`* `= RTW.TflCFunctionEntry` or *`hEntry`* `= RTW.TflCOperationEntry`.

Example: `op_entry`

`sourceFile` — Name of an additional source file
character vector | string scalar

The *`sourceFile`* is a character vector or string scalar specifying an additional source file.

Example: 'all_additions.c'

See Also

addAdditionalHeaderFile | addAdditionalIncludePath | addAdditionalSourcePath

Topics

"Define Code Replacement Library Optimizations"

Introduced in R2007b

addAdditionalSourcePath

Add source path to array of source paths for code replacement table entry

Syntax

`addAdditionalSourcePath(hEntry,path)`

Description

`addAdditionalSourcePath(hEntry,path)` adds a specified additional source file path to the array of additional source file paths for a code replacement table.

This function adds `-I` to the compile line in the generated makefile.

Examples

Specify Path to Additional Header and Source Files

This example shows how to use the `addAdditionalSourcePath` function with `addAdditionalHeaderFile`, `addAdditionalIncludePath`, and `addAdditionalSourceFile` to specify path to additional header and source files fully for a code replacement table entry.

```
% Path to external header and source files
libdir = fullfile('$(MATLAB_ROOT)','..', '..', 'lib');

op_entry = RTW.TflCOperationEntry;
.
.
.
addAdditionalHeaderFile(op_entry, 'all_additions.h');
addAdditionalIncludePath(op_entry, fullfile(libdir, 'include'));

addAdditionalSourceFile(op_entry, 'all_additions.c');
addAdditionalSourcePath(op_entry, fullfile(libdir, 'src'));
```

Input Arguments

`hEntry` — Handle to a code replacement table entry
handle

The *`hEntry`* is a handle to a code replacement table entry previously returned by instantiating a code replacement table entry class, such as *`hEntry`* `= RTW.TflCFunctionEntry` or *`hEntry`* `= RTW.TflCOperationEntry`.

Example: `op_entry`

`path` — Path to an additional source file
character vector | string scalar

The *`path`* is a character vector or string scalar specifying the full path to an additional source file. The character vector or string scalar can include tokens (for example, `$myfolder$`, where

`myfolder` is a variable defined as a character vector, cell array of character vectors, or string array in the MATLAB workspace).

Example: `fullfile(libdir, 'src')`

See Also

`addAdditionalHeaderFile` | `addAdditionalIncludePath` | `addAdditionalSourceFile`

Topics

"Define Code Replacement Library Optimizations"

Introduced in R2007b

addAlgorithmProperty

Add algorithm properties for code replacement table entry

Syntax

`addAlgorithmProperty(hEntry, name-value)`

Arguments

hEntry

Handle to a code replacement table entry previously returned by instantiating a code replacement entry class, such as `hEntry = RTW.TflCFunctionEntry` or `hEntry = RTW.TflCOperationEntry`.

name-value

Algorithm property, specified as a comma-separated pair consisting of the name of an algorithm property and one or more algorithm values. Specify multiple values as a cell array of character vectors.

Name	Values
'BPPower2Spacing'	'off'|'on'
'ExtrapMethod'	'Clip'|'Linear'
'IndexSearchMethod'	'Evenly spaced points'|'Linear search'|'Binary search'
'InterpMethod'	'Linear point-slope'|'Linear Lagrange'|'Flat'|'Nearest'
'RemoveProtectionInput'	'off'|'on'
'RndMeth'	'Ceiling'|'Convergent'|'Floor'|'Nearest'|'Round'|'Simplest'|'Zero'
'SaturateOnIntegerOverflow'	'off'|'on'
'SupportTunableTableSize'	'off'|'on'
'UseLastTableValue'	'off'|'on'
'UseRowMajorAlgorithm'	'off'|'on'

Description

The `addAlgorithmProperty` function adds algorithm property settings to the conceptual representation of a code replacement table entry. For example, use this function to adjust the algorithms applied by lookup table functions.

Examples

In the following example, the `addAlgorithmProperty` function configures the code generator to apply the following methods when replacing code for the `lookup1D` function:

- Clip extrapolation
- Linear interpolation
- Binary or linear index search

```
hLib = RTW.TflTable;

hEnt = RTW.TflCFunctionEntry;
hEnt.setTflCFunctionEntryParameters( ...
                    'Key',                        'lookup1D', ...
                    'Priority',                   100, ...
                    'ImplementationName',         'my_Lookup1D_Repl', ...
                    'ImplementationHeaderFile', 'my_Lookup1D.h', ...
                    'ImplementationSourceFile', 'my_Lookup1D.c', ...
                    'GenCallback', 'RTW.copyFileToBuildDir');

arg = hEnt.getTflArgFromString('y1','double');
arg.IOType = 'RTW_IO_OUTPUT';
hEnt.addConceptualArg(arg);

arg = hEnt.getTflArgFromString('u1','double');
hEnt.addConceptualArg(arg);

arg = RTW.TflArgMatrix('u2','RTW_IO_INPUT','double');
arg.DimRange = [0 0; Inf Inf];
hEnt.addConceptualArg(arg);

arg = RTW.TflArgMatrix('u3', 'RTW_IO_INPUT', 'double');
arg.DimRange = [0 0; Inf Inf];
hEnt.addConceptualArg(arg);

hEnt.addAlgorithmProperty('ExtrapMethod', 'Clip');
hEnt.addAlgorithmProperty('InterpMethod', 'Linear point-slope');
hEnt.addAlgorithmProperty('IndexSearchMethod', 'Linear search');
```

See Also

`getTflArgFromString`

Topics

“Lookup Table Function Code Replacement”
“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”

Introduced in R2014b

addArgConf

Add argument configuration information for Simulink model port to model-specific C function prototype

Syntax

`addArgConf(obj, portName, category, argName, qualifier)`

Description

`addArgConf(obj, portName, category, argName, qualifier)` method adds argument configuration information for a port in your ERT-based Simulink® model to a model-specific C function prototype. You specify the name of the model port, the argument category (`'Value'` or `'Pointer'`), the argument name, and the argument type qualifier (for example, `'const'`).

The order of `addArgConf` calls determines the argument position for the port in the function prototype, unless you change the order by other means, such as the `setArgPosition` method.

If a port has an existing argument configuration, subsequent calls to `addArgConf` with the same port name overwrite the previous argument configuration of the port.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.ModelSpecificCPrototype` or `obj = RTW.getFunctionSpecification(modelName)`.
portName	Character vector specifying the unqualified name of an inport or outport in your Simulink model.
category	Character vector specifying the argument category, either `'Value'` or `'Pointer'`.
argName	Character vector specifying a valid C identifier.
qualifier	Character vector specifying the argument type qualifier: `'none'`, `'const'`, `'const *'`, or `'const * const'`.

Examples

In the following example, you use the `addArgConf` method to add argument configuration information for ports `Input` and `Output` in an ERT-based version of `rtwdemo_counter`. After executing these commands, click the **Configure Model Functions** button on the **Interface** pane of the Configuration Parameters dialog box to open the Model Interface dialog box and confirm that the `addArgConf` commands succeeded.

```
rtwdemo_counter
set_param(gcs,'SystemTargetFile','ert.tlc')

%% Create a function control object
a=RTW.ModelSpecificCPrototype

%% Add argument configuration information for Input and Output ports
```

```
addArgConf(a,'Input','Pointer','inputArg','const *')
addArgConf(a,'Output','Pointer','outputArg','none')

%% Attach the function control object to the model
attachToModel(a,gcs)
```

Alternatives

You can use the Configure C Step Function Interface dialog box to customize the base rate C step function for a rate-based model. See “Configure Name and Arguments for Individual Step Functions”.

See Also

`attachToModel`

Topics

“Configure C Code Generation for Model Entry-Point Functions”

addBaseline

Add baseline file for comparison

Syntax

```
cgvObj.addBaseline(inputName,baselineFile)
cgvObj.addBaseline(inputName,baselineFile,toleranceFile)
```

Description

cgvObj.`addBaseline(inputName,baselineFile)` associates a baseline data file to an `inputName` in *cgvObj*. *cgvObj* is a handle to a `cgv.CGV` object. If a baseline file is present, when you call `cgv.CGV.run`, *cgvObj* automatically compares baseline data to the result data of the current execution of *cgvObj*.

cgvObj.`addBaseline(inputName,baselineFile,toleranceFile)` includes an optional tolerance file to apply when comparing the baseline data to the result data of the current execution of *cgvObj*.

Input Arguments

inputName

A unique numeric or character identifier assigned to the input data associated with `baselineFile`

baselineFile

A MAT-file containing baseline data

toleranceFile

File containing the tolerance specification, which is created using `createToleranceFile`

Examples

A typical workflow for defining baseline data in a `cgv.CGV` object and then comparing the baseline data to the execution data is as follows:

1. Create a `cgv.CGV` object for a model.
2. Add input data to the `cgv.CGV` object by calling `addInputData`.
3. Add the baseline file to the `cgv.CGV` object by calling `addBaseline`. which associates the `inputName` for input data in the `cgv.CGV` object with input data stored in the `cgv.CGV` object as the baseline data.
4. Run the `cgv.CGV` object by calling `run`, which automatically compares the baseline data to the result data in this execution.
5. Call `getStatus` to determine the results of the comparison.

See Also

`addInputData` | `createToleranceFile` | `getStatus` | `run`

Topics

"Verify Numerical Equivalence with CGV"

addHeaderReportFcn

Add callback function to execute before executing input data in object

Syntax

cgvObj.addHeaderReportFcn(CallbackFcn)

Description

cgvObj.addHeaderReportFcn(CallbackFcn) adds a callback function to *cgvObj*. *cgvObj* is a handle to a cgv.CGV object. run calls *CallbackFcn* before executing input data included in *cgvObj*. The callback function signature is:

```
CallbackFcn(cgvObj)
```

Examples

The callback function, *HeaderReportFcn*, is added to cgv.CGV object, *cgvObj*

```
cgvObj.addHeaderReportFcn(@HeaderReportFcn);
```

where *HeaderReportFcn* is defined as:

```
function HeaderReportFcn(cgvObj)
...
end
```

See Also

run

Topics

"Callbacks for Customized Model Behavior"

addPostExecFcn

Add callback function to execute after each input data file is executes

Syntax

cgvObj.addPostExecFcn(CallbackFcn)

Description

cgvObj.addPostExecFcn(CallbackFcn) adds a callback function to *cgvObj*. *cgvObj* is a handle to a cgv.CGV object. run calls *CallbackFcn* after each input data file is executed for the model. The callback function signature is:

```
CallbackFcn(cgvObj, inputIndex)
```

inputIndex is a unique numerical identifier associated with input data in the *cgvObj*.

Examples

The callback function, *PostExecutionFcn*, is added to cgv.CGV object, *cgvObj*

```
cgvObj.addPostExecFcn(@PostExecutionFcn);
```

where *PostExecutionFcn* is defined as:

```
function PostExecutionFcn(cgvObj, inputIndex)
...
end
```

See Also

run

Topics

"Callbacks for Customized Model Behavior"

addPostExecReportFcn

Add callback function to execute after each input data file executes

Syntax

cgvObj.addPostExecReportFcn(CallbackFcn)

Description

cgvObj.addPostExecReportFcn(CallbackFcn) adds a callback function to *cgvObj*. *cgvObj* is a handle to a cgv.CGV object. run calls *CallbackFcn* after each input data file is executed for the model. The callback function signature is:

```
CallbackFcn(cgvObj, inputIndex)
```

inputIndex is a unique numeric identifier associated with input data in the *cgvObj*.

Examples

The callback function, *PostExecutionReportFcn*, is added to cgv.CGV object, *cgvObj*

```
cgvObj.addPostExecReportFcn(@PostExecutionReportFcn);
```

where *PostExecutionReportFcn* is defined as:

```
function PostExecutionReportFcn(cgvObj, inputIndex)
...
end
```

See Also

run

Topics

"Callbacks for Customized Model Behavior"

addPreExecFcn

Add callback function to execute before each input data file executes

Syntax

cgvObj.addPreExecFcn(CallbackFcn)

Description

cgvObj.addPreExecFcn(CallbackFcn) adds a callback function to *cgvObj*. *cgvObj* is a handle to a cgv.CGV object. run calls *CallbackFcn* before executing each input data file in *cgvObj*. The callback function signature is:

```
CallbackFcn(cgvObj, inputIndex)
```

inputIndex is a unique numeric identifier associated with input data in *cgvObj*.

Examples

The callback function, *PreExecutionFcn*, is added to cgv.CGV object, *cgvObj*

```
cgvObj.addPreExecFcn(@PreExecutionFcn);
```

where *PreExecutionFcn* is defined as:

```
function PreExecutionFcn(cgvObj, inputIndex)
...
end
```

See Also

run

Topics

"Callbacks for Customized Model Behavior"

addPreExecReportFcn

Add callback function to execute before each input data file executes

Syntax

cgvObj.addPreExecReportFcn(CallbackFcn)

Description

cgvObj.addPreExecReportFcn(CallbackFcn) adds a callback function to *cgvObj*. *cgvObj* is a handle to a cgv.CGV object. run calls *CallbackFcn* before executing each input data file in *cgvObj*. The callback function signature is:

```
CallbackFcn(cgvObj, inputIndex)
```

inputIndex is a unique numerical identifier associated with input data in *cgvObj*.

Examples

The callback function, *PreExecutionReportFcn*, is added to cgv.CGV object, *cgvObj*

```
cgvObj.addPreExecReportFcn(@PreExecutionReportFcn);
```

where *PreExecutionReportFcn* is defined as:

```
function PreExecutionReportFcn(cgvObj, inputIndex)
...
end
```

See Also

run

Topics

"Callbacks for Customized Model Behavior"

addTrailerReportFcn

Add callback function to execute after the input data executes

Syntax

`cgvObj.addTrailerReportFcn(CallbackFcn)`

Description

`cgvObj.addTrailerReportFcn(CallbackFcn)` adds a callback function to *cgvObj*. *cgvObj* is a handle to a `cgv.CGV` object. `run` executes the input data files in *cgvObj* and then calls *CallbackFcn*. The callback function signature is:

```
CallbackFcn(cgvObj)
```

Examples

The callback function, *TrailerReportFcn*, is added to `cgv.CGV` object, *cgvObj*

```
cgvObj.addTrailerReportFcn(@TrailerReportFcn);
```

where *TrailerReportFcn* is defined as:

```
function TrailerReportFcn(cgvObj)
...
end
```

See Also

`run`

Topics

"Callbacks for Customized Model Behavior"

addCheck

Package: `rtw.codegenObjectives`

Add check to code generation objective

Syntax

`addCheck(objective, checkID)`

Description

`addCheck(objective, checkID)` adds the specified check to the specified objective in the Code Generation Advisor. When you select the objective, the Code Generation Advisor includes the check, unless another objective with a higher priority excludes the check.

Examples

Create a Custom Code Generation Objective

Create a custom objective named `Reduced RAM Example` that runs checks and verifies parameter values to confirm that the model is configured to reduce the RAM used by the generated code.

Create a file `sl_customization.m` to contain a callback function that creates the custom objective.

```
function sl_customization(cm)
%SL_CUSTOMIZATION objective customization callback

objCustomizer = cm.ObjectiveCustomizer;
index = objCustomizer.addCallbackObjFcn(@addObjectives);
objCustomizer.callbackFcn{index}();

end
```

Create and configure the objective in the `addObjectives` function. Set the name of the objective and add checks and parameters to verify. Then register the objective in the Code Generation Advisor.

```
function addObjectives

% Create the custom objective
obj = rtw.codegenObjectives.Objective('ex_ram_1');
setObjectiveName(obj, 'Reduce RAM Example');

% Add parameters to the objective
addParam(obj, 'InlineParams', 'on');
addParam(obj, 'BooleanDataType', 'on');
addParam(obj, 'OptimizeBlockIOStorage', 'on');
addParam(obj, 'EnhancedBackFolding', 'on');
addParam(obj, 'BooleansAsBitfields', 'on');

% Add additional checks to the objective
% The Code Generation Advisor automatically includes 'Check model
% configuration settings against code generation objectives' in every
% objective.
addCheck(obj, 'Identify unconnected lines, input ports, and output ports');
addCheck(obj, 'Check model and local libraries for updates');
```

```
%Register the objective
register(obj);

end
```

Input Arguments

objective — Code generation objective
rtw.codegenObjectives.Objective object

Code generation objective, specified as a `rtw.codegenObjectives.Objective` object.

checkID — Identifier of check
character vector | string scalar

Identifier of check that you want to add, specified as a character vector or string scalar.

Example: `'mathworks.codegen.CodeInstrumentation'`

See Also

`Simulink.ModelAdvisor`

Topics
"Create Custom Code Generation Objectives"
`Simulink.ModelAdvisor`

Introduced in R2009a

addComplexTypeAlignment

Specify alignment boundary of a complex type

Syntax

`addComplexTypeAlignment(hDataAlign,baseType,alignment)`

Description

`addComplexTypeAlignment(hDataAlign,baseType,alignment)` specifies the alignment boundary of real and complex data members of a complex type.

The starting memory address of the real and imaginary part of complex variables produced by the code generator with the specified type are a multiple of the specified alignment boundary. The code generator replaces operations in generated code when both of these conditions are true:

- A code replacement table entry has a complex argument with a data alignment requirement that is less than or equal to the alignment boundary value
- The entry satisfies all other code replacement match criteria.

To use this function, your code replacement library registration file must include additional compiler data alignment information, such as alignment syntax.

Examples

Specify Alignment Boundary for Complex Types

This example shows how to specify a 16-byte alignment boundary for complex `int8` types by adding the `addComplexTypeAlignment` line to your code replacement library registration file.

```
function rtwTargetInfo(cm)
% rtwTargetInfo function to register a code replacement library (CRL)
% for use with  code generation

  % Register the CRL defined in local function locCrlRegFcn
  cm.registerTargetInfo(@locCrlRegFcn);

end % End of RTWTARGETINFO

% Local function to define a CRL containing crl_table_mmul_4x4_single_align
function thisCrl = locCrlRegFcn

  % create an alignment specification object, assume gcc
  as = RTW.AlignmentSpecification;
  as.AlignmentType = {'DATA_ALIGNMENT_LOCAL_VAR', ...
                      'DATA_ALIGNMENT_GLOBAL_VAR', ...
                      'DATA_ALIGNMENT_STRUCT_FIELD'};
  as.AlignmentSyntaxTemplate = '__attribute__((aligned(%n)))';
  as.SupportedLanguages={'c', 'c++'};

  % add the alignment specification object
  da = RTW.DataAlignment;
  da.addAlignmentSpecification(as);
  da.addComplexTypeAlignment('int8', 16);

  % add the data alignment object to target characteristics
  tc = RTW.TargetCharacteristics;
  tc.DataAlignment = da;
```

```
  % Instantiate a CRL registry entry
  thisCrl = RTW.TflRegistry;

  % Define the CRL properties
  thisCrl.Name = 'Data Alignment Example';
  thisCrl.Description = 'Example of replacement with data alignment';
  thisCrl.TableList = {'crl_table_mmul_4x4_single_align'};
    thisCrl.TargetCharacteristics = tc;

end % End of LOCCRLREGFCN
```

Input Arguments

`hDataAlign` — Handle to a data alignment object

handle

The *`hDataAlign`* is a handle to a data alignment object, previously returned by *`hDataAlign`* `= RTW.DataAlignment`.

Example: `da`

`baseType` — Specifies a built-in data type

character vector | string scalar

The *`baseType`* is a character vector or string scalar that specifies a built-in data type such as `int8` or `long`.

Example: `'int8'`

`alignment` — Specifies the alignment boundary

positive integer

The *`alignment`* is a positive integer that is a power of 2 and does not exceed 128. This value specifies the alignment boundary.

Example: `16`

See Also

Topics

"Data Alignment for Code Replacement"
"Define Code Replacement Library Optimizations"
"Code You Can Replace from MATLAB Code"
"Code You Can Replace From Simulink Models"

Introduced in R2014a

addConceptualArg

Add conceptual argument to array of conceptual arguments for code replacement table entry

Syntax

`addConceptualArg(hEntry,arg)`

Description

`addConceptualArg(hEntry,arg)` adds a specified conceptual argument to the array of conceptual arguments for a code replacement table entry.

Examples

Add Conceptual Arguments for Ports

This example shows how the `addConceptualArg` function adds conceptual arguments for the output operand and the two input operands for an addition operation.

```
hLib = RTW.TflTable;

% Create entry for addition of built-in uint8 data type
op_entry = RTW.TflCOperationEntry;
op_entry.setTflCOperationEntryParameters( ...
    'Key',                      'RTW_OP_ADD', ...
    'Priority',                 90, ...
    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes',            {'RTW_ROUND_UNSPECIFIED'}, ...
    'ImplementationName',       'u8_add_u8_u8', ...
    'ImplementationHeaderFile', 'u8_add_u8_u8.h', ...
    'ImplementationSourceFile', 'u8_add_u8_u8.c' );

arg = hLib.getTflArgFromString('y1','uint8');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.addConceptualArg(arg);

arg = hLib.getTflArgFromString('u1','uint8');
op_entry.addConceptualArg(arg);

arg = hLib.getTflArgFromString('u2','uint8');
op_entry.addConceptualArg(arg);

op_entry.copyConceptualArgsToImplementation();

addEntry(hLib, op_entry);
```

Input Arguments

`hEntry` — Handle to a code replacement table entry
handle

The *hEntry* is a handle to a code replacement table entry previously returned by instantiating a code replacement entry class, such as `hEntry = RTW.TflCFunctionEntry` or `hEntry = RTW.TflCOperationEntry`.

Example: `op_entry`

arg — Argument added to the array of conceptual arguments
character vector | string scalar

The *arg* is the argument, such as returned by `arg = getTflArgFromString(name, datatype)`, added to the array of conceptual arguments for the code replacement table entry.

Example: `'hLib.getTflArgFromString('y1','uint8')'`

See Also

`getTflArgFromString`

Topics

“Define Code Replacement Library Optimizations”
“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”

Introduced in R2007b

addDWorkArg

Add DWork argument for semaphore entry in code replacement table

Syntax

`addDWorkArg(hEntry,arg)`

Description

`addDWorkArg(hEntry,arg)` adds a specified DWork argument to the arguments for a semaphore entry in a code replacement table.

Examples

Add a DWork Argument

This example shows how to use the `addDWorkArg` function to add a DWork argument named `d1` to the arguments for a semaphore entry in a code replacement table.

```
hLib = RTW.TflTable;

% specify semaphore init function.
hEnt = RTW.TflCSemaphoreEntry;
hEnt.setTflCSemaphoreEntryParameters( ...
  'Key',                        'RTW_SEM_INIT', ...
  'Priority',                   30, ...
  'ImplementationName',         'mySemCreate', ...
  'ImplementationHeaderFile',   'mySem.h', ...
  'ImplementationSourceFile',   'mySem.c', ...
  'ImplementationHeaderPath',   LibPath, ...
  'ImplementationSourcePath',   LibPath, ...
  'GenCallback',                'RTW.copyFileToBuildDir', ...
  'SideEffects',                true);

% specify conceptual operands and result
arg = hLib.getTflArgFromString('y1', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
hEnt.addConceptualArg(arg);
arg = hLib.getTflArgFromString('u1', 'void');
hEnt.addConceptualArg(arg);

% specify replacement function signature
arg=hLib.getTflArgFromString('y1','void');
hEnt.Implementation.setReturn(arg);
arg.IOType = 'RTW_IO_OUTPUT';

% DWork Arg
arg = hLib.getTflDWorkFromString('d1','void*');
hEnt.addDWorkArg(arg);
```

```
addEntry(hLib, hEnt);
```

Input Arguments

`hEntry` — Handle to a code replacement table entry

handle

The *`hEntry`* is a handle to a code replacement table entry previously returned by instantiating a code replacement semaphore table entry class, using *`hEntry`* `= RTW.TflCSemaphoreEntry`.

Example: `sem_entry`

`arg` — Argument added to the arguments for the table entry

character vector | string scalar

Argument, such as returned by *`arg`* `= getTflDWorkFromString(`*`name`*`,` *`datatype`*`)`, added to the arguments for the code replacement table entry.

Example: `'hLib.getTflDWorkFromString('d1','void*')'`

See Also

`getTflDWorkFromString`

Topics

"Define Code Replacement Library Optimizations"
"Code You Can Replace from MATLAB Code"
"Code You Can Replace From Simulink Models"

Introduced in R2011b

addConfigSet

Add configuration set

Syntax

```
cgvObj.addConfigSet(configSet)
cgvObj.addConfigSet('configSetName')
cgvObj.addConfigSet('file','configSetFileName')
cgvObj.addConfigSet('file','configSetFileName','variable', 'configSetName')
```

Description

`cgvObj.addConfigSet(configSet)` is an optional method that adds the configuration set to the object. *cgvObj* is a handle to a `cgv.CGV` object. *configSet* is a variable that specifies a configuration set.

`cgvObj.addConfigSet('configSetName')` is an optional method that adds the configuration set to the object. *configSetName* is a character vector that specifies the name of the configuration set in the workspace.

`cgvObj.addConfigSet('file','configSetFileName')` is an optional method that adds the configuration set to the object. *configSetFileName* is a character vector that specifies the name of the file that contains only one configuration set.

`cgvObj.addConfigSet('file','configSetFileName','variable', 'configSetName')` is an optional method that adds the configuration set to the object. The file contains one or more configuration sets. Specify the name of the configuration set to use.

This method replaces the configuration parameter values in the model with the values from the configuration set that you add. The object applies the configuration set when you call the `run` method. You can add only one configuration set for each `cgv.CGV` object.

See Also

Topics
"Manage Configuration Sets for a Model"
"Programmatic Code Generation Verification"

addEntry

Add table entry to collection of table entries registered in code replacement table

Syntax

`addEntry(hTable,entry)`

Description

`addEntry(hTable,entry)` adds a function or operator entry that you have constructed to the collection of table entries registered in a code replacement table.

Examples

Add Operator Entry to Code Replacement Table

This example shows how to use the `addEntry` function to add an operator entry to a code replacement table after the entry is constructed.

```
hLib = RTW.TflTable;

% Create an entry for addition of built-in uint8 data type
op_entry = RTW.TflCOperationEntry;
op_entry.setTflCOperationEntryParameters( ...
    'Key',                      'RTW_OP_ADD', ...
    'Priority',                 90, ...
    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes',            {'RTW_ROUND_UNSPECIFIED'}, ...
    'ImplementationName',       'u8_add_u8_u8', ...
    'ImplementationHeaderFile', 'u8_add_u8_u8.h', ...
    'ImplementationSourceFile', 'u8_add_u8_u8.c' );

arg = hLib.getTflArgFromString('y1','uint8');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.addConceptualArg( arg );

arg = hLib.getTflArgFromString('u1','uint8');
op_entry.addConceptualArg( arg );

arg = hLib.getTflArgFromString('u2','uint8');
op_entry.addConceptualArg( arg );

op_entry.copyConceptualArgsToImplementation();

addEntry(hLib, op_entry);
```

Input Arguments

`hTable` — Handle to a code replacement table
handle

The *hTable* is a handle to a code replacement table previously returned by *hTable* = RTW.TflTable.

Example: hLib

entry — Handle to a function or operator entry
handle

The *entry* is a handle to a function or operator entry that you have constructed after calling *hEntry* = RTW.TflCFunctionEntry or *hEntry* = RTW.TflCOperationEntry.

Example: op_entry

See Also

Topics
"Define Code Replacement Library Optimizations"
"Code You Can Replace from MATLAB Code"
"Code You Can Replace From Simulink Models"

Introduced in R2007b

addInputData

Add input data

Syntax

cgvObj.addInputData(inputName, inputDataFile)

Description

cgvObj.addInputData(inputName, inputDataFile) adds an input data file to *cgvObj*. *cgvObj* is a handle to a cgv.CGV object. inputName is a unique identifier, which *cgvObj* associates with the input data in inputDataFile.

Input Arguments

inputName

inputName is a unique numeric or character identifier, which is associated with the input data in inputDataFile.

inputDataFile

inputDataFile is an input data file, with or without the .mat extension. *cgvObj* uses the input data when the model executes during cgv.CGV.run. If the input file is in the working folder, the *cgvObj* does not require the path. addInputData does not qualify that the contents of inputDataFile relate to the inputs of the model. Data that is not used by the model will not throw a warning or error.

Tips

- When calling addInputData you can modify configuration parameters by including their settings in the input file, inputDataFile.
- If you omit calling addInputData before executing the model, the cgv.CGV object runs once using data in the base workspace.
- The *cgvObj* uses the inputName to identify the input data associated with output data and output data files. *cgvObj* passes inputName to a callback function to identify the input data that the callback function uses.

See Also

run

Topics

"Verify Numerical Equivalence with CGV"

addParam

Package: rtw.codegenObjectives

Add parameters to code generation objective

Syntax

addParam(objective, param, value)

Description

addParam(objective, param, value) adds the specified parameter to the objective and defines the value of the parameter that the Code Generation Advisor verifies in **Check model configuration settings against code generation objectives**.

Examples

Create a Custom Code Generation Objective

Create a custom objective named Reduced RAM Example that runs checks and verifies parameter values to confirm that the model is configured to reduce the RAM used by the generated code.

Create a file sl_customization.m to contain a callback function that creates the custom objective.

```
function sl_customization(cm)
%SL_CUSTOMIZATION objective customization callback

objCustomizer = cm.ObjectiveCustomizer;
index = objCustomizer.addCallbackObjFcn(@addObjectives);
objCustomizer.callbackFcn{index}();

end
```

Create and configure the objective in the addObjectives function. Set the name of the objective and add checks and parameters to verify. Then register the objective in the Code Generation Advisor.

```
function addObjectives

% Create the custom objective
obj = rtw.codegenObjectives.Objective('ex_ram_1');
setObjectiveName(obj, 'Reduce RAM Example');

% Add parameters to the objective
addParam(obj, 'InlineParams', 'on');
addParam(obj, 'BooleanDataType', 'on');
addParam(obj, 'OptimizeBlockIOStorage', 'on');
addParam(obj, 'EnhancedBackFolding', 'on');
addParam(obj, 'BooleansAsBitfields', 'on');

% Add additional checks to the objective
% The Code Generation Advisor automatically includes 'Check model
% configuration settings against code generation objectives' in every
% objective.
addCheck(obj, 'Identify unconnected lines, input ports, and output ports');
addCheck(obj, 'Check model and local libraries for updates');
```

```
%Register the objective
register(obj);

end
```

Input Arguments

objective — Code generation objective
rtw.codegenObjectives.Objective object

Code generation objective, specified as a `rtw.codegenObjectives.Objective` object.

param — Name of parameter
character vector | string scalar

Name of parameter to add to the objective, specified as a character vector or string scalar.

value — Parameter value
character vector | string scalar

Parameter value to verify in the Code Generation Advisor, specified as a character vector or string scalar.

See Also

`get_param`

Topics
"Create Custom Code Generation Objectives"

Introduced in R2009a

addPostLoadFiles

Add files required by model

Syntax

`cgvObj.addPostLoadfiles({FileList})`

Description

`cgvObj.addPostLoadfiles({FileList})` is an optional method that adds a list of MATLAB and MAT-files to the object. *cgvObj* is a handle to a `cgv.CGV` object. *cgvObj* executes and loads the files after opening the model and before running tests. *FileList* is a cell array of names of MATLAB and MAT-files in the testing directory that the model requires to run.

Note Subsequent `cgvObj.addPostLoadFiles` calls to the same `cgv.CGV` object replaces the list of MATLAB and MAT-files of that object.

See Also

Topics
"Verify Numerical Equivalence with CGV"
"Callbacks for Customized Model Behavior"

annotate

Package: `coder.profile`

Color profiled model components or open model with profiled components colored

Syntax

`annotate(myExecutionProfile)`

Description

`annotate(myExecutionProfile)` colors the profiled model components blue. If the model is closed, this command opens the model, with profiled components colored blue. Clicking a blue component opens a window that displays execution-time metrics for generated code.

Examples

Annotate Model

To annotate the profiled model, use the annotate function and the `executionProfile` workspace variable.

```
annotate(executionProfile);
```

Input Arguments

`myExecutionProfile` — Variable specifies annotation
workspace variable

When you run a SIL or PIL simulation with code execution profiling, the software generates the workspace variable `executionProfile`, specified in **Configuration Parameters > Code Generation > Verification > Workspace variable**.

Example: `executionProfile`

See Also

`report`

Topics
"Code Execution Profiling with SIL and PIL"
"View and Compare Code Execution Times"

Introduced in R2016b

attachToModel

Attach model-specific C function prototype to loaded ERT-based Simulink model

Syntax

`attachToModel(obj, modelName)`

Description

`attachToModel(obj, modelName)` attaches a model-specific C function prototype to a loaded ERT-based Simulink model.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.ModelSpecificCPrototype`.
modelName	Character vector specifying the name of a loaded ERT-based Simulink model to which the object is going to be attached.

Alternatives

Use the Configure C Step Function Interface dialog box to customize the base rate C step function for a rate-based model. See “Configure Name and Arguments for Individual Step Functions”.

See Also

Topics
“Configure C Code Generation for Model Entry-Point Functions”

cgv.CGV

Represent a model as a code generation verification object that has methods to configure and execute a model in a variety of simulation modes to automate verification of numerical equivalence of executing the generated code

Description

A code generation verification (`cgv`) object executes a model in different environments such as, simulation, Software-In-the-Loop (SIL), or Processor-In-the-Loop (PIL) and stores numerical results. Using the `cgv.CGV` class methods, you can create a script to verify that the model and the generated code produce numerically equivalent results.

`cgv.CGV` and `cgv.Config` use two of the same properties. Before executing a `cgv.CGV` object, use `cgv.Config` to verify the model configured for the mode of execution that you specify. If the top model is set to normal simulation mode, referenced models set to PIL mode are changed to Accelerator mode.

Creation

`cgvObj = cgv.CGV(model_name)` creates a handle to a code generation verification object using the default parameter values. `model_name` is the name of the model that you are verifying.

`cgvObj = cgv.CGV(model_name,Name,Value)` constructs the object using the parameter values, specified as `Name,Value` pair arguments. Parameter names and values are not case sensitive.

The `cgv.CGV` function accepts these arguments:

model_name

Name of the model that you are verifying.

Name-Value Pair Arguments

Optional comma-separated pairs of `Name,Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in a variety of orders, such as `Name1,Value1,…,NameN,ValueN`.

- `'ComponentType'` — Define the SIL or PIL approach

  If `topmodel` (default), top-model SIL or PIL simulation mode and standalone code interface.

  If `modelblock`, model block SIL or PIL simulation mode and model reference code interface mode.

  If mode of execution is simulation (`Connectivity` is `sim`), choosing either value for `ComponentType` does not alter simulation results.
- `Connectivity` — Specify mode of execution

  If `sim` or `normal` (default), mode of execution is Normal simulation.

If `sil`, mode of execution is SIL.

If `pil`, mode of execution is PIL.

Example: "Test Model for Numerical Equivalence" on page 1-45

Properties

`Description` — Object description
`' '` (null character vector) (default)

Specify a description of the object.

`Name` — Object name
`' '` (null character vector) (default)

Specify a name for the object.

Object Functions

activateConfigSet	Activate configuration set of model
addBaseline	Add baseline file for comparison
addConfigSet	Add configuration set
addHeaderReportFcn	Add callback function to execute before executing input data in object
addInputData	Add input data
addPostExecFcn	Add callback function to execute after each input data file is executes
addPostExecReportFcn	Add callback function to execute after each input data file executes
addPostLoadFiles	Add files required by model
addPreExecFcn	Add callback function to execute before each input data file executes
addPreExecReportFcn	Add callback function to execute before each input data file executes
addTrailerReportFcn	Add callback function to execute after the input data executes
compare	Compare signal data
copySetup	Create copy of cgv.CGV object
createToleranceFile	Create file correlating tolerance information with signal names
getOutputData	Get output data
getSavedSignals	Display list of signal names to command line
getStatus	Return execution status
plot	Create plot for signal or multiple signals
run	Execute CGV object
setMode	Specify mode of execution
setOutputDir	Specify folder
setOutputFile	Specify output data file name

Examples

Test Model for Numerical Equivalence

The general workflow for testing a model for numerical equivalence using the `cgv.CGV` class is to:

Create a `cgv.CGV` object, `cgvObj`, for each mode of execution and use the `cgv.CGV` set up methods to configure the model for each execution. The set up methods are:

- `addInputData`
- `addPostLoadFiles`
- `setOutputDir`
- `setOutputFile`
- `addCallBack`
- `addConfigSet`

Run the model for each mode of execution using the `cgvObj.run` method.

Use the `cgv.CGV` access methods to get and evaluate the data. The access methods are:

- `getOutputData`
- `getSavedSignals`
- `plot`
- `compare`

An object should be run only once. After the object is run, the set up methods are not used for that object. You then use the access methods for verifying the numerical equivalence of the results.

Note Simulink Test™ is a separate product that provides additional capabilities for SIL and PIL testing, for example, test sequence construction and test management.

See Also

`cgv.Config`

Topics

"Verify Numerical Equivalence with CGV"

Introduced in R2009b

cgv.Config

Check and modify model configuration parameter values

Description

A code generation verification configuration (`cgv.Config`) supports checking and optionally modifying models for compatibility with various modes of execution that use generated code, such as, Software-In-the-Loop (SIL) or Processor-In-the-Loop (PIL).

To execute the model in the mode that you specify, you might need to make additional modifications to the configuration parameter values or the model beyond those configured by the `cgv.Config` object.

By default, `cgv.Config` modifies configuration parameter values to the values that it recommends, but does not save the model. Alternatively, you can use `cgv.Config` parameters to modify the default specification. For more information, see the properties, `ReportOnly` and `SaveModel`.

If you use `cgv.Config` to modify a model, do not use referenced configuration sets in that model. If a model uses a referenced configuration set, update the model with a copy of the configuration set, by using the `Simulink.ConfigSetRef.getRefConfigSet` method.

If you use `cgv.Config` on a model that executes a callback function, the callback function might modify configuration parameter values each time the model loads. The callback function might revert changes that `cgv.Config` made. If this change occurs, the model might not be set up for SIL or PIL. For more information, see “Callbacks for Customized Model Behavior”.

Creation

`cfgObj = cgv.Config(model_name)` creates a handle to a `cgv.Config` object, `cfgObj`, using default values for properties. `model_name` is the name of the model that you are checking and optionally configuring.

`cfgObj = cgv.Config(model_name, Name, Value)` constructs the object using options, specified as parameter name and value pairs. Parameter names and values are not case sensitive.

The `cgv.Config` function accepts these arguments:

model_name

Name of the model that you are verifying.

Name-Value Pair Arguments

Optional comma-separated pairs of `Name,Value` arguments, where `Name` is an argument name or an property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in a variety of orders, such as `Name1,Value1,…,NameN,ValueN`.

Properties

CheckOutports — Check for compatible outports
on (default) | off

Specify whether to compile the model and check that the model outports configuration is compatible with the cgv.CGV object. If your script fixes errors reported by cgv.Config, you can set CheckOutports to off.

- If on, compile the model and check the model outports configuration.
- If off, do not compile the model or check the model outports configuration.

Example: 'CheckOutports','on'

ComponentType — Define the SIL or PIL approach
topmodel (default) | modelblock

If mode of execution is simulation (connectivity is sim), choosing either value for ComponentType does not alter simulation results. However, cgv.Config recommends configuration parameter values based on the value of ComponentType.

If topmodel, the top-model SIL or PIL simulation mode and standalone code interface.

If modelblock, model block SIL or PIL simulation mode and model reference code interface.

Example: 'ComponentType','topmodel'

Connectivity — Specify mode of execution
sim (default) | sil | pil

If sim, the mode of execution is simulation. Recommends changes to a subset of the configuration parameters that SIL and PIL targets require.

If sil, the mode of execution is SIL. Requires that the system target file is set to 'ert.tlc' and that you do not use your own external target. Recommends changes to the configuration parameters that SIL targets require.

If pil, the mode of execution is PIL with custom connectivity that you provide using the PIL Connectivity API. Recommends changes to the configuration parameters that PIL targets with custom connectivity require.

Example: 'Connectivity','sim'

LogMode — Select log mode
SignalLogging | SaveOutput

Specify the **Signal logging** and **Output** parameters on the **Data Import/Export** pane of the Configuration Parameters dialog box.

If set to SignalLogging, the object logs signal data to a MATLAB workspace variable during execution. This parameter selects the **Data Import/Export** > **Signal logging** parameter in the Configuration Parameters dialog box.

If set to SaveOutput, the object saves output data to a MATLAB workspace variable during execution. This parameter selects **Data Import/Export** > **Output** parameter in the Configuration Parameters dialog box. The **Output** parameter does not save bus outputs.

Example: `'SignalLogging',''`

`ReportOnly` — Select report or modify configuration
`off` (default) | `on`

The `ReportOnly` property specifies whether `cgv.Config` modifies the recommended values of the configuration parameters of the model. If you set `ReportOnly` to `on`, `SaveModel` must be `off`.

Example: `'ReportOnly','off'`

`SaveModel` — Select save model
`off` (default) | `on`

Specify whether to save the model with the configuration parameter values recommended by `cgv.Config`. If you set `SaveModel` to `'on'`, `ReportOnly` must be `'off'`.

Example: `'SaveModel','off'`

Object Functions

configModel	Determine and change configuration parameter values
displayReport	Display results of comparing configuration parameter values
getReportData	Return results of comparing configuration parameter values

Examples

Configure Model for Top-Model SIL

Configure the `rtwdemo_iec61508` model for top-model SIL. Then view the changes at the MATLAB Command Window

```
% Create a cgv.Config object and configure the model for top-model SIL.
load_system('rtwdemo_iec61508');
set_param('rtwdemo_iec61508', 'SaveFormat', 'StructureWithTime');
cgvCfg = cgv.Config('rtwdemo_iec61508', 'LogMode', 'SaveOutput', ...
    'connectivity', 'sil');
cgvCfg.configModel();

% Display the results of what the cgv.Config object changed.
cgvCfg.displayReport();

% Close the rtwdemo_iec61508 model.
bdclose('rtwdemo_iec61508');
```

See Also

`cgv.CGV`

Topics
"Programmatic Code Generation Verification"

Introduced in R2009b

coder.dataAlignment

Specify data alignment for global or entry-point/exported function input and output arguments

Syntax

```
coder.dataAlignment('varName',align_value)
```

Description

`coder.dataAlignment('varName',align_value)` specifies data alignment in MATLAB code for the variable (`varName`), which is imported data or global (exported) data. The code generator aligns the imported or exported data to the alignment boundary (`align_value`).

Examples

Data Alignment for Imported Data

An example function that specifies data alignment for imported data.

```
function y = importedDataExampleFun(x1,x2)

coder.dataAlignment('x1',16);      % Specifies information
coder.dataAlignment('x2',16);    % Specifies information
coder.dataAlignment('y',16);     % Specifies information

y = x1 + x2;

end
```

Data Alignment for Exported Data

An example function that specifies data alignment for exported data.

```
function a = exportedDataExampleFun(b)

global z;
coder.dataAlignment('z',8);

a = b + z;

end
```

Input Arguments

'varName' — Variable name
character array

The *varName* is a character array of the variable name that requires alignment information specification.

align_value — Data alignment boundary value
integer

The *align_value* is an integer number which should be a power of 2, from 2 through 128. This number specifies the power-of-2 byte alignment boundary.

Limitations

Limitations on variables supported by `coder.dataAlignment` directive:

- Only use `coder.dataAlignment` to specify alignment information for function inputs, outputs, and global variables.
- `coder.dataAlignment` supports only matrix types, including matrix of complex types.
- For exported custom storage classes (CSCs), `coder.dataAlignment` supports only `ExportedGlobal`. You can specify alignment information for any imported CSCs.
- The code generator ignores `coder.dataAlignment` for non-ERT or non-ERT derived system target files.
- Global variables tagged using the `coder.dataAlignment` directive from within a MATLAB function block are ignored. Set the alignment value on the corresponding Data Store Memory.
- The `coder.dataAlignment` function generates an error if a code replacement library is not specified.

See Also

`codegen`

Topics
"Data Alignment for Code Replacement"
"Define Code Replacement Library Optimizations"
"What Is Code Replacement Customization?"
"What Is Code Replacement?"

Introduced in R2017a

coder.Dictionary class

Package: `coder`

Configure Embedded Coder Dictionary

Description

An object of the `coder.Dictionary` class represents an Embedded Coder Dictionary. Use the object to perform operations on the Embedded Coder Dictionary, such as load packages of definitions and gain access to sections of the dictionary.

A `coder.Dictionary` object contains three `coder.dictionary.Section` objects, which represent the sections of an Embedded Coder Dictionary: Storage Classes, Memory Sections, and Function Customization Templates. A `coder.dictionary.Section` object contains `coder.dictionary.Entry` objects, which represent the definitions stored in that section of the Embedded Coder Dictionary. The name of the section identifies the type of code definitions that the section contains. To access the sections of an Embedded Coder Dictionary, use a `coder.Dictionary` object. To access the dictionary entries within the section, use a `coder.dictionary.Section` object.

Creation

The functions `coder.dictionary.create` and `coder.dictionary.open` create a `coder.Dictionary` object.

Methods

Public Methods

getSection	Return coder.dictionary.Section object that represents Embedded Coder Dictionary section
getSections	Return coder.dictionary.Section objects of an Embedded Coder Dictionary
loadPackage	Load package of code definitions into Embedded Coder Dictionary
unloadPackage	Unload package of code definitions from Embedded Coder Dictionary
refreshPackage	Refresh package definitions in Embedded Coder Dictionary
resetToDefault	Restore Embedded Coder Dictionary to default state
valid	Determine whether coder.Dictionary object represents a valid Embedded Coder Dictionary
empty	Determine whether Embedded Coder Dictionary is empty

Examples

Create Embedded Coder Dictionary in Simulink Data Dictionary

Create a data dictionary to contain an Embedded Coder Dictionary.

```
dataDictionary = Simulink.data.dictionary.create('DataDictionary.sldd');
```

Create an Embedded Coder Dictionary in the data dictionary and use a `coder.Dictionary` object to represent the Embedded Coder Dictionary.

```
coderDictionary = coder.dictionary.create(dataDictionary)
```

```
coderDictionary = 

  Dictionary with Sections:

                     StorageClasses: [1×1 coder.dictionary.Section]
                     MemorySections: [1×1 coder.dictionary.Section]
    FunctionCustomizationTemplates: [1×1 coder.dictionary.Section]
```

The Embedded Coder Dictionary contains three `coder.dictionary.Section` objects, each of which represents a section of the dictionary.

See Also

Embedded Coder Dictionary | `coder.dictionary.Entry` | `coder.dictionary.Section`

Introduced in R2019b

coder.dictionary.copy

Package: `coder.dictionary`

Copy code generation definitions between models and data dictionaries

Syntax

`copy(sourceName,destinationName)`

Description

`copy(sourceName,destinationName)` copies code generation definitions, such as storage classes, from the Embedded Coder Dictionary in `sourceName` to the Embedded Coder Dictionary in `destinationName`.

If a code generation definition in `sourceName` has the same name as a definition in `destinationName`, `copy` copies the source entry into the destination, and then renames the copy.

To share code definitions between models, use a Simulink data dictionary, as described in "Share Embedded Coder Dictionary Definition Between Models". To copy individual code definitions, use the Embedded Coder Dictionary dialog box or `copyEntry`. For general information about Embedded Coder Dictionaries and code generation definitions, see "Define Storage Classes, Memory Sections, and Function Templates for Software Architecture".

Examples

Copy Code Definitions from Model to Model

In the Embedded Coder Dictionary of the example model `rtwdemo_roll`, create a storage class. Then, copy the storage class to the Embedded Coder Dictionary in `rtwdemo_rtwecintro`.

Open the model `rtwdemo_roll` and represent the Embedded Coder Dictionary by using a `coder.Dictionary` object. Use this object to access the Storage Classes section of the dictionary and represent the section by using a `coder.dictionary.Section` object.

```
rtwdemo_roll
coderDictionary = coder.dictionary.open('rtwdemo_roll');
storageClassesSect = getSection(coderDictionary, 'StorageClasses');
```

Add a storage class definition named `MyStorageClass` to the Storage Classes section. The storage class definition uses the default property settings.

```
newEntry = addEntry(storageClassesSect,'MyStorageClass')
```

```
newEntry = 

  Entry with properties:

          Name: 'MyStorageClass'
    DataSource: 'rtwdemo_roll'
```

Save a copy of `rtwdemo_roll` in your current folder. Saving the model saves the storage class in the Embedded Coder Dictionary.

Open the other model, `rtwdemo_rtwecintro`.

```
rtwdemo_rtwecintro
```

Copy the contents of the Embedded Coder Dictionary in `rtwdemo_roll` to the Embedded Coder Dictionary in `rtwdemo_rtwecintro`.

```
coder.dictionary.copy('rtwdemo_roll','rtwdemo_rtwecintro')
```

Open the Embedded Coder Dictionary for `rtwdemo_rtwecintro`.

In the Embedded Coder Dictionary window, on the **Storage Classes** tab, the storage class `MyStorageClass` appears.

Input Arguments

sourceName — Source model file or data dictionary

character vector | string scalar

Source model file or data dictionary, specified as a character vector or string scalar.

- A model must be loaded (for example, by using `load_system`) or open.

  You do not need to specify the `.slx` file extension.
- A dictionary must be open in the Model Explorer, in the current folder, or on the MATLAB path.

  You must specify the `.sldd` file extension.

Example: `'myLoadedModel'`

Example: `'myDictionary.sldd'`

Data Types: `char`

destinationName — Destination model file or data dictionary

character vector | string scalar

Destination model file or data dictionary, specified as a character vector or string scalar.

- A model must be loaded (for example, by using `load_system`) or open.

  You do not need to specify the `.slx` file extension.
- A dictionary must be open in the Model Explorer, in the current folder, or on the MATLAB path.

  You must specify the `.sldd` file extension.

Example: `'myLoadedModel'`

Example: `'myDictionary.sldd'`

Data Types: `char`

See Also

Embedded Coder Dictionary | `coder.dictionary.Entry` | `copyEntry`

Topics

"Define Storage Classes, Memory Sections, and Function Templates for Software Architecture"
"Configure Default C Code Generation for Categories of Data Elements and Functions"

Introduced in R2018a

coder.dictionary.create

Package: `coder.dictionary`

Create Embedded Coder Dictionary and `coder.Dictionary` object

Syntax

```
coderDictionaryObj = coder.dictionary.create(sourceName)
```

Description

`coderDictionaryObj = coder.dictionary.create(sourceName)` creates an Embedded Coder Dictionary in the model or Simulink data dictionary identified by `sourceName`. The function returns a `coder.Dictionary` object that represents the new Embedded Coder Dictionary.

When the source model or data dictionary already has an Embedded Coder Dictionary, use `coder.dictionary.open` to access the `coder.Dictionary` object. An Embedded Coder Dictionary is created when you open a model in the Embedded Coder app or when you open the Embedded Coder Dictionary dialog box for a model or a data dictionary.

Examples

Create Embedded Coder Dictionary in Simulink Data Dictionary

Create a data dictionary.

```
dataDictionary = Simulink.data.dictionary.create('DataDictionary.sldd');
```

Create an Embedded Coder Dictionary in the data dictionary.

```
coderDictionary = coder.dictionary.create(dataDictionary);
```

```
coderDictionary = 

  Dictionary with Sections:

                    StorageClasses: [1×1 coder.dictionary.Section]
                    MemorySections: [1×1 coder.dictionary.Section]
    FunctionCustomizationTemplates: [1×1 coder.dictionary.Section]
```

Input Arguments

`sourceName` — Target model file or data dictionary
character vector | string scalar | `Simulink.data.Dictionary` object

Name of target model file or data dictionary, specified as a character vector, string scalar, or `Simulink.data.Dictionary` object.

- A model must be loaded (for example, by using `load_system`) or open.

You do not need to specify the `.slx` file extension.
- A dictionary must be open in the Model Explorer, in the current folder, or on the MATLAB path.

  You must specify the `.sldd` file extension.

Example: `'myLoadedModel'`

Example: `'myDictionary.sldd'`

Data Types: `char`

Output Arguments

`coderDictionaryObj` — New Embedded Coder Dictionary
`coder.Dictionary` object

Newly created Embedded Coder Dictionary, returned as a `coder.Dictionary` object.

See Also
Embedded Coder Dictionary | `coder.Dictionary`

Topics
"Create Code Definitions Programmatically"

Introduced in R2019b

coder.dictionary.exist

Package: `coder.dictionary`

Determine whether Embedded Coder Dictionary exists in model or data dictionary

Syntax

```
tf = coder.dictionary.exist(sourceName)
```

Description

`tf = coder.dictionary.exist(sourceName)` returns `true` if a model or Simulink data dictionary `sourceName` contains an Embedded Coder Dictionary.

An Embedded Coder Dictionary is created when you open a model in the Embedded Coder app or when you open the Embedded Coder Dictionary dialog box for a model or a data dictionary.

Examples

Create an Embedded Coder Dictionary in a Data Dictionary

Create a Simulink data dictionary. Then, determine if the data dictionary contains an Embedded Coder Dictionary. Use the result to create an Embedded Coder Dictionary or to open the existing dictionary.

Create a Simulink data dictionary.

```
dataDictionary = Simulink.data.dictionary.create('DataDictionary.sldd');
```

Determine whether the data dictionary contains an Embedded Coder Dictionary. Open the existing Embedded Coder Dictionary or create and open an Embedded Coder Dictionary in the data dictionary.

```
if coder.dictionary.exist('DataDictionary.sldd')

    myCoderDict = coder.dictionary.open('DataDictionary.sldd');

else

    myCoderDict = coder.dictionary.create('DataDictionary.sldd');

end
```

Because the data dictionary did not contain an Embedded Coder Dictionary, the code creates a coder dictionary in the data dictionary.

Input Arguments

`sourceName` — Name of model or Simulink data dictionary
character vector | string scalar

Name of the model or Simulink data dictionary, specified as a character vector or string scalar.

Example: `'rtwdemo_roll'`

Output Arguments

`tf` — True or false result
`1` | `0` | logical array

True or false result, returned as `1` or `0` of data type `logical`.

See Also

`coder.Dictionary` | `coder.dictionary.create` | `coder.dictionary.open` | `coder.dictionary.remove`

Topics
"Create Code Definitions Programmatically"

Introduced in R2020b

coder.dictionary.move

Package: coder.dictionary

Migrate code generation definitions between models and data dictionaries

Syntax

move(sourceName,destinationName)

Description

move(sourceName,destinationName) moves code generation definitions, such as storage classes, from the Embedded Coder Dictionary in sourceName to the Embedded Coder Dictionary in destinationName. The definitions are removed from sourceName. To copy code definitions from one Embedded Coder Dictionary to another, use coder.dictionary.copy.

If a code generation definition in sourceName has the same name as a definition in destinationName, move moves the source entry into the destination, and then renames the entry in the destination.

Use this function to:

- Move code generation definitions from a model to a Simulink data dictionary. For more information, see “Share Embedded Coder Dictionary Definition Between Models”.
- In a hierarchy of referenced Simulink data dictionaries, move the Embedded Coder Dictionary from one data dictionary to another. In a hierarchy of referenced dictionaries, only one dictionary can store an Embedded Coder Dictionary.

For general information about Embedded Coder Dictionaries and code generation definitions, see “Define Storage Classes, Memory Sections, and Function Templates for Software Architecture”.

Examples

Move Code Definitions from Model to Model

Create a storage class in the Embedded Coder Dictionary of the example model rtwdemo_roll. Then, move the storage class to the Embedded Coder Dictionary in rtwdemo_rtwecintro.

Open the model rtwdemo_roll and represent the Embedded Coder Dictionary by using a coder.Dictionary object. Use this object to access the Storage Classes section of the dictionary and represent the section by using a coder.dictionary.Section object.

```
rtwdemo_roll
coderDictionary = coder.dictionary.open('rtwdemo_roll');
storageClassesSect = getSection(coderDictionary, 'StorageClasses');
```

Add a storage class definition named MyStorageClass to the Storage Classes section. The storage class definition uses the default property settings.

```
newEntry = addEntry(storageClassesSect,'MyStorageClass')
```

```
newEntry = 

  Entry with properties:

          Name: 'MyStorageClass'
    DataSource: 'rtwdemo_roll'
```

Save a copy of `rtwdemo_roll` in your current folder. Saving the model saves the storage class in the Embedded Coder Dictionary.

Open the other model, `rtwdemo_rtwecintro`.

```
rtwdemo_rtwecintro
```

Move the contents of the Embedded Coder Dictionary in `rtwdemo_roll` to the Embedded Coder Dictionary in `rtwdemo_rtwecintro`.

```
coder.dictionary.move('rtwdemo_roll','rtwdemo_rtwecintro')
```

Open the Embedded Coder Dictionary for `rtwdemo_rtwecintro`.

In the Embedded Coder Dictionary window, on the **Storage Classes** tab, the storage class `MyStorageClass` appears. The storage class no longer exists in `rtwdemo_roll`.

Input Arguments

`sourceName` — Source model file or data dictionary

character vector | string scalar

Source model file or data dictionary, specified as a character vector or string scalar.

- A model must be loaded (for example, by using `load_system`) or open.

  You do not need to specify the `.slx` file extension.
- A dictionary must be open in the Model Explorer, in the current folder, or on the MATLAB path.

  You must specify the `.sldd` file extension.

Example: `'myLoadedModel'`

Example: `'myDictionary.sldd'`

Data Types: `char`

`destinationName` — Destination model file or data dictionary

character vector | string scalar

Destination model file or data dictionary, specified as a character vector or string scalar.

- A model must be loaded (for example, by using `load_system`) or open.

  You do not need to specify the `.slx` file extension.
- A dictionary must be open in the Model Explorer, in the current folder, or on the MATLAB path.

  You must specify the `.sldd` file extension.

Example: `'myLoadedModel'`

Example: `'myDictionary.sldd'`

Data Types: `char`

See Also

Embedded Coder Dictionary

Topics

"Migrate Definitions from Model File to Shared Data Dictionary"
"Define Storage Classes, Memory Sections, and Function Templates for Software Architecture"
"Configure Default C Code Generation for Categories of Data Elements and Functions"

Introduced in R2018a

coder.dictionary.open

Package: `coder.dictionary`

Open Embedded Coder Dictionary object

Syntax

`coderDictionaryObj = coder.dictionary.open(sourceName)`

Description

`coderDictionaryObj = coder.dictionary.open(sourceName)` opens the Embedded Coder Dictionary in the model or Simulink data dictionary identified by `sourceName`. The function returns a `coder.Dictionary` object representing the Embedded Coder Dictionary.

Examples

Open Existing Embedded Coder Dictionary

Open the model `rtwdemo_roll`, which has an Embedded Coder Dictionary.

```
rtwdemo_roll
```

Open the Embedded Coder Dictionary and represent it with a `coder.Dictionary` object.

```
coderDictionaryObj = coder.dictionary.open('rtwdemo_roll')
```

```
coderDictionaryObj = 

  Dictionary with Sections:

                   StorageClasses: [1×1 coder.dictionary.Section]
                   MemorySections: [1×1 coder.dictionary.Section]
    FunctionCustomizationTemplates: [1×1 coder.dictionary.Section]
```

Input Arguments

`sourceName` — Target model file or data dictionary
character vector | string scalar | `Simulink.data.Dictionary` object

Name of the model file or data dictionary that contains the Embedded Coder Dictionary, specified as a character vector.

- A model must be loaded (for example, by using `load_system`) or open.

  You do not need to specify the `.slx` file extension.
- A dictionary must be open in the Model Explorer, in the current folder, or on the MATLAB path.

  You must specify the `.sldd` file extension.

Example: `'myLoadedModel'`

Example: `'myDictionary.sldd'`

Data Types: `char`

Output Arguments

`coderDictionaryObj` — Embedded Coder Dictionary
`coder.Dictionary` object

Embedded Coder Dictionary, returned as a `coder.Dictionary` object.

See Also

Embedded Coder Dictionary | `coder.Dictionary`

Introduced in R2019b

coder.dictionary.remove

Package: coder.dictionary

Remove Embedded Coder Dictionary from model or Simulink data dictionary

Syntax

remove(sourceName)

Description

remove(sourceName) removes Embedded Coder Dictionary definitions from your model or Simulink data dictionary identified by sourceName. When you remove the Embedded Coder Dictionary from a model, you remove custom definitions and definitions from packages that you have loaded. The model still contains the local dictionary with definitions from the SimulinkBuiltIn package. When you remove the Embedded Coder Dictionary from a model that is not linked to a Simulink data dictionary with definitions, the remaining local dictionary contains definitions from the Simulink package. When you remove the Embedded Coder Dictionary from a Simulink data dictionary, you remove the entire Embedded Coder Dictionary, including its packages and definitions.

Use this function to:

- Remove Embedded Coder Dictionary definitions from a model.
- In a hierarchy of referenced Simulink data dictionaries, remove the Embedded Coder Dictionary from a data dictionary. In a hierarchy of referenced dictionaries, only one dictionary can store an Embedded Coder Dictionary.

To migrate code generation definitions from one source to another (for example, from a model file to a Simulink data dictionary), consider using coder.dictionary.move.

Examples

Remove Embedded Coder Dictionary from Model File

When you open the Embedded Coder Dictionary window for a model (see “Open the Embedded Coder Dictionary” on page 23-0) or open the Code perspective for a model, Simulink creates an Embedded Coder Dictionary in the model file. In this example, create a code generation definition (a storage class) in the Embedded Coder Dictionary of the example model rtwdemo_roll, then remove the Embedded Coder Dictionary from the model.

Open the model rtwdemo_roll and represent the Embedded Coder Dictionary by using a coder.Dictionary object. Use this object to access the Storage Classes section of the dictionary and represent the section by using a coder.dictionary.Section object.

```
rtwdemo_roll
coderDictionary = coder.dictionary.open('rtwdemo_roll');
storageClassesSect = getSection(coderDictionary, 'StorageClasses');
```

Add a storage class definition named `MyStorageClass` to the Storage Classes section. The storage class definition uses the default property settings.

```
newEntry = addEntry(storageClassesSect,'MyStorageClass')
```

```
newEntry = 

  Entry with properties:

          Name: 'MyStorageClass'
    DataSource: 'rtwdemo_roll'
```

At the command prompt, remove the Embedded Coder Dictionary from the model.

```
coder.dictionary.remove('rtwdemo_roll')
```

Now, the model file contains an Embedded Coder Dictionary with only the code generation definitions from the `Simulink` and `SimulinkBuiltIn` packages.

Input Arguments

`sourceName` — Target model file or data dictionary
character vector | string scalar

Target model file or data dictionary, specified as a character vector or string scalar.

- A model must be loaded (for example, by using `load_system`) or open.

  You do not need to specify the `.slx` file extension.
- A dictionary must be open in the Model Explorer, in the current folder, or on the MATLAB path.

  You must specify the `.sldd` file extension.

Example: `'myLoadedModel'`

Example: `'myDictionary.sldd'`

Data Types: `char`

Tip

To use `coder.dictionary.remove` on a data dictionary that references other data dictionaries, you must:

1. Temporarily remove references to dictionaries that also contain code generation definitions.
2. Use `coder.dictionary.remove` on the target dictionary.
3. Restore the dictionary references that you removed.

See Also

Embedded Coder Dictionary

Topics

"Define Storage Classes, Memory Sections, and Function Templates for Software Architecture"
"Configure Default C Code Generation for Categories of Data Elements and Functions"

Introduced in R2018a

copyEntry

Class: `coder.dictionary.Section`
Package: `coder.dictionary`

Copy Embedded Coder Dictionary entry

Syntax

```
newEntry = copyEntry(sectionObj,defName)
newEntry = copyEntry(sectionObj,defName,targetCoderDict)
```

Description

`newEntry = copyEntry(sectionObj,defName)` copies the definition named `defName` in the Embedded Coder dictionary section represented by `sectionObj`. The `copyEntry` syntax returns a `coder.dictionary.Entry` object that represents the new definition.

`newEntry = copyEntry(sectionObj,defName,targetCoderDict)` creates a copy of the definition in the target Embedded Coder Dictionary represented by `targetCoderDict`.

Input Arguments

`sectionObj` — Source Embedded Coder Dictionary section
`coder.dictionary.Section` object

Source Embedded Coder Dictionary section that contains the code definition, specified as a `coder.dictionary.Section` object. The section name identifies the type of code definition that `addEntry` creates.

`defName` — Name of Embedded Coder Dictionary definition
character vector | string scalar

Name of Embedded Coder Dictionary definition that you want to copy, specified as a string.

Example: `'StorageClass2'`

`targetCoderDict` — Target Embedded Coder Dictionary
`coder.Dictionary` object

Target Embedded Coder Dictionary, specified as a `coder.Dictionary` object.

Output Arguments

`newEntry` — Embedded Coder Dictionary entry
`coder.dictionary.Entry` object

New Embedded Coder Dictionary entry, returned as a `coder.dictionary.Entry` object. The new entry represents a copy of the code definition in the target section of the Embedded Coder Dictionary.

Examples

Copy Storage Class in Embedded Coder Dictionary

Open the model `rtwdemo_roll` and represent the Embedded Coder Dictionary by using a `coder.Dictionary` object. The dictionary contains an example storage class definition named `SignalStruct`.

```
rtwdemo_roll
coderDictionary = coder.dictionary.open('rtwdemo_roll');
```

Create a `coder.dictionary.Section` object that represents the Storage Classes section of the Embedded Coder Dictionary.

```
storageClassesSect = getSection(coderDictionary, 'StorageClasses');
```

Copy the storage class definition `SignalStruct`. The new storage class has the name `SignalStruct_copy`.

```
newEntry = copyEntry(storageClassesSect,'SignalStruct')
```

```
newEntry = 

  Entry with properties:

          Name: 'SignalStruct_copy'
    DataSource: 'rtwdemo_roll'
```

Copy Storage Class to Different Embedded Coder Dictionary

Open the model `rtwdemo_roll` and represent the Embedded Coder Dictionary with a `coder.Dictionary` object. The dictionary contains an example storage class definition named `SignalStruct`.

```
rtwdemo_roll
coderDictionary = coder.dictionary.open('rtwdemo_roll');
```

Create a `coder.dictionary.Section` object that represents the Storage Classes section of the Embedded Coder Dictionary.

```
storageClassesSect = getSection(coderDictionary, 'StorageClasses');
```

Create a data dictionary that contains an Embedded Coder Dictionary to store the copy of `SignalStruct`.

```
newSLDD = Simulink.data.dictionary.create('newSLDD.sldd');
slddCoderDictionary = coder.dictionary.create('newSLDD.sldd');
```

Copy the storage class definition `SignalStruct` from the model's Embedded Coder Dictionary to the data dictionary. The new storage class has the name `SignalStruct`.

```
newEntry = copyEntry(storageClassesSect,'SignalStruct',slddCoderDictionary)
```

```
newEntry = 
```

```
  Entry with properties:

          Name: 'SignalStruct'
    DataSource: 'X:\newSLDD.sldd'
```

See Also

coder.Dictionary | coder.dictionary.Entry

Introduced in R2019b

empty

Class: `coder.Dictionary`
Package: `coder`

Determine whether Embedded Coder Dictionary is empty

Syntax

`tf = empty(coderDictionaryObj)`

Description

`tf = empty(coderDictionaryObj)` returns `true` if the `coder.Dictionary` object `coderDictionaryObj` is empty. When you remove the Embedded Coder Dictionary from a model or a data dictionary by using `coder.dictionary.remove`, the Embedded Coder Dictionary becomes empty.

Input Arguments

`coderDictionaryObj` — Embedded Coder Dictionary object
`coder.Dictionary` object

Embedded Coder Dictionary, specified as a `coder.Dictionary` object.

Output Arguments

`tf` — True or false result
`1` | `0` | logical array

True or false result, returned as a `1` or `0` of data type `logical`.

Examples

Check If Embedded Coder Dictionary Object Is Empty

Create a data dictionary named `'dataDict.sldd'` in the base workspace, and then create an Embedded Coder Dictionary in the data dictionary. In the base workspace, represent the Embedded Coder Dictionary by using the `coder.Dictionary` object `myCoderDictObj`.

```
Simulink.data.dictionary.create('dataDict.sldd');
myCoderDictObj = coder.dictionary.create('dataDict.sldd');
```

Check if the Embedded Coder Dictionary is empty. When you created the Embedded Coder Dictionary, definitions from the `Simulink` package were loaded into it, so the dictionary is initially not empty.

```
empty(myCoderDictObj)
```

```
ans =

  logical

   0
```

Remove the Embedded Coder Dictionary from the data dictionary and again check to see if it is empty.

```
coder.dictionary.remove('dataDict.sldd');
empty(myCoderDictObj)
```

```
ans =

  logical

   1
```

See Also

coder.Dictionary

Introduced in R2019b

getSection

Class: coder.Dictionary
Package: coder

Return coder.dictionary.Section object that represents Embedded Coder Dictionary section

Syntax

```
sectionObj = getSection(coderDictionaryObj,sectionName)
```

Description

sectionObj = getSection(coderDictionaryObj,sectionName) returns a coder.dictionary.Section object that represents one section, sectionName, of an Embedded Coder Dictionary, which coderDictionaryObj represents. Use the section to access the code definitions of the type identified by the section name.

Input Arguments

coderDictionaryObj — Embedded Coder Dictionary containing section
coder.Dictionary object

Embedded Coder Dictionary containing section that you want to access, specified as a coder.Dictionary object. Before you use this function, represent the dictionary with a coder.Dictionary object by using, for example, the coder.dictionary.create function or the coder.dictionary.open function.

sectionName — Name of section
'StorageClasses' | 'MemorySections' | 'FunctionCustomizationTemplates'

Name of section that you want to access in the Embedded Coder Dictionary, specified as a string. The section name identifies the type of code definitions that the section contains.

Example: 'StorageClasses'

Output Arguments

sectionObj — Embedded Coder Dictionary section
coder.dictionary.Section object

Section in the Embedded Coder Dictionary, returned as a coder.dictionary.Section object. The section contains coder.dictionary.Entry objects that represent code definitions of the type identified by sectionName.

Examples

Access Storage Classes in Embedded Coder Dictionary

Open the model `rtwdemo_roll` and represent the Embedded Coder Dictionary by using a `coder.Dictionary` object. Assign the `coder.Dictionary` object to the variable `coderDictObj`.

```
rtwdemo_roll
coderDictObj = coder.dictionary.open('rtwdemo_roll')
```

```
coderDictObj = 

  Dictionary with Sections:

                   StorageClasses: [1×1 coder.dictionary.Section]
                   MemorySections: [1×1 coder.dictionary.Section]
    FunctionCustomizationTemplates: [1×1 coder.dictionary.Section]
```

Represent the Storage Classes section of the Embedded Coder Dictionary by using a `coder.dictionary.Section` object named `SCSectObj`.

```
SCSectObj = getSection(coderDictObj,'StorageClasses')
```

```
SCSectObj = 

  Section with properties:

    Name: 'StorageClasses'
```

See Also

`coder.Dictionary` | `coder.dictionary.Section`

Introduced in R2019b

getSections

Class: coder.Dictionary
Package: coder

Return coder.dictionary.Section objects of an Embedded Coder Dictionary

Syntax

```
sections = getSections(coderDictionaryObj)
```

Description

sections = getSections(coderDictionaryObj) returns the coder.dictionary.Section objects contained in the Embedded Coder Dictionary that coderDictionaryObj represents. Use the section objects to access code definitions of different types identified by the section names.

Input Arguments

coderDictionaryObj — Embedded Coder Dictionary
coder.Dictionary object

Embedded Coder Dictionary, specified as a coder.Dictionary object. Before you use this function, represent the dictionary with a coder.Dictionary object by using, for example, the coder.dictionary.create function or the coder.dictionary.open function.

Output Arguments

sections — Sections of Embedded Coder Dictionary
array of coder.dictionary.Section objects

Sections of the Embedded Coder Dictionary, returned as an array of coder.dictionary.Section objects.

Examples

Access Sections in Embedded Coder Dictionary

Open the model rtwdemo_roll and represent the Embedded Coder Dictionary by using a coder.Dictionary object. Assign the coder.Dictionary object to the variable coderDictObj.

```
rtwdemo_roll
coderDictObj = coder.dictionary.open('rtwdemo_roll')
```

```
coderDictObj = 

  Dictionary with Sections:

                      StorageClasses: [1×1 coder.dictionary.Section]
```

```
                    MemorySections: [1×1 coder.dictionary.Section]
    FunctionCustomizationTemplates: [1×1 coder.dictionary.Section]
```

Access the sections of the dictionary by using an array of the corresponding `coder.dictionary.Section` objects.

```
sectionsArray = getSections(coderDictObj)
```

```
sectionsArray = 

  1×3 Section array with properties:

    Name
```

Use each `coder.dictionary.Section` object to access the entries in that section of the dictionary. The first section contains entries that represent function customization template definitions.

```
sectObj = sectionsArray(1)
```

```
sectObj = 

  Section with properties:

    Name: 'FunctionCustomizationTemplates'
```

See Also

`coder.Dictionary` | `coder.dictionary.Entry` | `coder.dictionary.Section`

Introduced in R2019b

loadPackage

Class: coder.Dictionary
Package: coder

Load package of code definitions into Embedded Coder Dictionary

Syntax

```
loadPackage(coderDictionaryObj,pkgName)
```

Description

`loadPackage(coderDictionaryObj,pkgName)` loads the code definitions in the package `pkgName` into the Embedded Coder Dictionary that `coderDictionaryObj` represents.

Input Arguments

coderDictionaryObj — Embedded Coder Dictionary object
`coder.Dictionary` object

Embedded Coder Dictionary, specified as a `coder.Dictionary` object.

pkgName — package
character vector | string scalar

Package of code definitions to load into the Embedded Coder Dictionary.

Example: `'Simulink'`

Examples

Load Package Definitions into Embedded Coder Dictionary

Open the model `rtwdemo_roll` and represent the Embedded Coder Dictionary by using a `coder.Dictionary` object. Assign the `coder.Dictionary` object to the variable `coderDictObj`.

```
rtwdemo_roll
coderDictObj = coder.dictionary.open('rtwdemo_roll');
```

Load the package `mpt` into the Embedded Coder Dictionary. You can configure the model to use definitions from the package.

```
loadPackage(coderDictObj,'mpt')
```

See Also

`coder.Dictionary`

Topics

“Refer to Code Generation Definitions in a Package” on page 23-6

Introduced in R2019b

refreshPackage

Class: `coder.Dictionary`
Package: `coder`

Refresh package definitions in Embedded Coder Dictionary

Syntax

`refreshPackage(coderDictionaryObj)`

Description

`refreshPackage(coderDictionaryObj)` refreshes code definitions stored in packages, which you create and modify by using the Custom Storage Class Designer. You do not need to refresh code definitions that you create in the Embedded Coder Dictionary. The `refreshPackage` function unloads and reloads the packages that the Embedded Coder Dictionary represented by `coderDictionaryObj` refers to. When you change the code definitions in a package, use `refreshPackage` to reload the package in each Embedded Coder Dictionary that refers to the package.

Input Arguments

`coderDictionaryObj` — Embedded Coder Dictionary object
`coder.Dictionary` object

Embedded Coder Dictionary, specified as a `coder.Dictionary` object.

Examples

Refresh Package Definitions in Embedded Coder Dictionary

Open the model `rtwdemo_roll` and represent the Embedded Coder Dictionary by using a `coder.Dictionary` object. Assign the `coder.Dictionary` object to the variable `coderDictObj`.

```
rtwdemo_roll
coderDictObj = coder.dictionary.open('rtwdemo_roll');
```

Refresh the packages in the Embedded Coder Dictionary. If you changed definitions in the packages, the changes now appear in the Embedded Coder Dictionary.

```
refreshPackage(coderDictObj)
```

See Also

`coder.Dictionary`

Topics
"Change Code Generation Definitions"

Introduced in R2019b

resetToDefault

Class: coder.Dictionary
Package: coder

Restore Embedded Coder Dictionary to default state

Syntax

```
resetToDefault(coderDictionaryObj)
```

Description

`resetToDefault(coderDictionaryObj)` restores the Embedded Coder Dictionary represented by `coderDictionaryObj` to its default state. The default Embedded Coder Dictionary includes definitions only from the `Simulink` and `SimulinkBuiltIn` packages. `resetToDefault` removes custom definitions from the dictionary.

Input Arguments

coderDictionaryObj — Embedded Coder Dictionary object
`coder.Dictionary` object

Embedded Coder Dictionary specified as a `coder.Dictionary` object.

Examples

Reset Embedded Coder Dictionary

Open the model `rtwdemo_roll` and represent the Embedded Coder Dictionary with a `coder.Dictionary` object. Assign the `coder.Dictionary` object to the variable `coderDictObj`. The Embedded Coder Dictionary for `rtwdemo_roll` contains two example storage classes and the default storage classes.

```
rtwdemo_roll
coderDictObj = coder.dictionary.open('rtwdemo_roll');
```

Reset the Embedded Coder Dictionary. Resetting the dictionary removes the example storage classes and leaves only the definitions in the `Simulink` and `SimulinkBuiltIn` packages.

```
resetToDefault(coderDictObj)
```

See Also
`coder.Dictionary`

Introduced in R2019b

unloadPackage

Class: coder.Dictionary
Package: coder

Unload package of code definitions from Embedded Coder Dictionary

Syntax

unloadPackage(coderDictionaryObj,pkgName)

Description

unloadPackage(coderDictionaryObj,pkgName) unloads the code definitions in the package pkgName from the Embedded Coder Dictionary that coderDictionaryObj represents.

Input Arguments

coderDictionaryObj — Embedded Coder Dictionary object
coder.Dictionary object

Embedded Coder Dictionary, specified as a coder.Dictionary object.

pkgName — package
character vector | string scalar

Package of code definitions to unload from the Embedded Coder Dictionary.

Example: 'Simulink'

Example: 'mpt'

Examples

Unload Package Definitions from Embedded Coder Dictionary

Open the model rtwdemo_roll and represent the Embedded Coder Dictionary by using a coder.Dictionary object. Assign the coder.Dictionary object to the variable coderDictObj.

```
rtwdemo_roll
coderDictObj = coder.dictionary.open('rtwdemo_roll');
```

Unload the package Simulink from the dictionary. The definitions are not available for configuring model elements.

```
unloadPackage(coderDictObj,'Simulink')
```

See Also
loadPackage

Introduced in R2019b

valid

Class: `coder.Dictionary`
Package: `coder`

Determine whether `coder.Dictionary` object represents a valid Embedded Coder Dictionary

Syntax

```
tf = valid(coderDictionaryObj)
```

Description

`tf = valid(coderDictionaryObj)` returns `true` if the `coder.Dictionary` object `coderDictionaryObj` is valid. A `coder.Dictionary` object is valid if it represents an existing Embedded Coder Dictionary. When you remove the Embedded Coder Dictionary, the `coder.Dictionary` object is not valid.

Input Arguments

`coderDictionaryObj` — Embedded Coder Dictionary object
`coder.Dictionary` object

Embedded Coder Dictionary, specified as a `coder.Dictionary` object.

Output Arguments

`tf` — True or false result
`1` | `0` | logical array

True or false result, returned as a `1` or `0` of data type `logical`.

Examples

Check If Embedded Coder Dictionary Object Is Valid

Create a data dictionary named `'dataDict.sldd'` in the base workspace and create an Embedded Coder Dictionary in the data dictionary. Represent the Embedded Coder Dictionary by using the `coder.Dictionary` object `myCoderDictObj` in the base workspace.

```
Simulink.data.dictionary.create('dataDict.sldd');
myCoderDictObj = coder.dictionary.create('dataDict.sldd');
```

Check whether the `coder.Dictionary` object in the base workspace represents a valid Embedded Coder Dictionary.

```
valid(myCoderDictObj)
```

```
ans =
```

```
  logical

   1
```

Remove the Embedded Coder Dictionary from the data dictionary and check the `coder.Dictionary` object again. When you remove the Embedded Coder Dictionary, the `coder.Dictionary` object is not valid.

```
coder.dictionary.remove('dataDict.sldd');
valid(myCoderDictObj)
```

```
ans =

  logical

   0
```

See Also
`coder.Dictionary`

Introduced in R2019b

coder.dictionary.Entry class

Package: `coder.dictionary`

Configure Embedded Coder Dictionary definition

Description

An object of the `coder.dictionary.Entry` class represents one definition of an Embedded Coder Dictionary. In this API, the object is called an *entry*. The information that the object represents is a *definition*. In this documentation, *definition* refers to the definition of the entry object.

A `coder.Dictionary` object contains three `coder.dictionary.Section` objects, which represent the sections of an Embedded Coder Dictionary: Storage Classes, Memory Sections, and Function Customization Templates. A `coder.dictionary.Section` object contains `coder.dictionary.Entry` objects, which represent the definitions stored in that section of the Embedded Coder Dictionary. The name of the section identifies the type of code definitions that the section contains. To access the sections of an Embedded Coder Dictionary, use a `coder.Dictionary` object. To access the dictionary entries within the section, use a `coder.dictionary.Section` object.

Creation

The functions `addEntry`, `getEntry`, `copyEntry`, and `find` create `coder.dictionary.Entry` objects.

Code Definition Properties

A `coder.dictionary.Entry` object has these properties.

`Name` — Name of code definition
character vector

Name of the coder dictionary definition that the entry represents. The name must be unique among the definitions in the section of the dictionary.

`DataSource` — Location of code definition
character vector

The location of the code definition. This property is read-only.

To access the properties of the code definition that an entry represents, use the `getAvailableProperties`, `get`, and `set` methods. Each type of code definition has available properties listed.

Storage Classes

`Name` — Name of storage class
`StorageClass1` (default) | character vector | string scalar

Name of the storage class. The name must be unique among the storage classes in the dictionary.

For lists of built-in and example storage classes that Simulink provides, see “Choose Storage Class for Controlling Data Representation in Generated Code”.

`Description` — Purpose and functionality of storage class
character vector | string scalar

Custom text that you can use to describe the purpose and functionality of the storage class.

`DataSource` — Location of code definition
character vector

The location of the code definition. This property is read-only.

`DataAccess` — Specification to access the data
`Direct` (default) | `Function`

Specification to access data associated with the model. Access the data directly (`Direct`) or through customizable `get` and `set` functions (`Function`). For more information, see “Access Data Through Functions by Using Storage Classes in Embedded Coder Dictionary”.

Dependencies

`DataScope` — Specification to generate data definition
`Exported` (default) | `Imported`

Specification that the generated code defines the data (`Exported`) or import (`Imported`) the data definition from external code. Built-in storage classes and storage classes in packages such as Simulink can use other scope options, such as `File`.

Dependencies

`Header File` — Name of header file that declares data
`$N.h` (default) | character vector | string scalar

Name of the header file that declares the data, specified as a name or naming rule. A naming rule includes a combination of text and tokens. Valid tokens are listed in this table.

Token	Description
`$R`	Name of root model
`$N`	Name of associated data element
`$G`	Name of storage class
`$U`	User token text, which you specify for a model as described in “Identifier Format Control”

Dependencies

`Definition File` — Name of source file that defines data
`$N.c` (default) | character vector | string scalar

Name of the source file that defines the data, specified as a name or naming rule. A naming rule includes a combination of text and tokens. Valid tokens are listed in this table.

Token	Description
$R	Name of root model
$N	Name of associated data element
$G	Name of storage class
$U	User token text, which you specify for a model as described in "Identifier Format Control"

Dependencies

Setting `DataScope` to `Imported` disables `DefinitionFile`. To include your external source code file in the build process, use model configuration parameters. For an example, see "Configure Data Interface".

`AccessMode` — Specification to access data through functions

`Value` (default) | `Pointer`

Specification for the storage class to access data associated with the model through functions by using `Value` or `Pointer`. For more information, see "Access Data Through Functions by Using Storage Classes in Embedded Coder Dictionary".

Dependencies

This property is enabled only when you set `DataAccess` to `Function`.

`AllowedAccess` — Specification to allow access to data through functions

`ReadWrite` (default) | `ReadOnly` | `WriteOnly`

Specification for the storage class to allow read and write (`ReadWrite`), read-only (`ReadOnly`), or write-only (`WriteOnly`) access to the data.

Dependencies

This property is enabled only when you set `DataAccess` to `Function`.

`GetFunctionName` — Name of the `get` function that fetches the associated data

`get_$N$M` (default) | character vector | string scalar

Name of the `get` function that fetches the associated data, specified as a name or naming rule. A naming rule includes a combination of text and tokens. Valid tokens are listed in this table.

Token	Description
$N	Name of associated data element (required)
$R	Name of root model
$M	Mangle text that ensures uniqueness
$U	User token text. See "Identifier Format Control".

Dependencies

This property is enabled only when you set `DataAccess` to `Function` and `AllowedAccess` to `ReadWrite` or `ReadOnly`.

SetFunctionName — Name of the set function that modifies the associated data

set_NM (default) | character vector | string scalar

Name of the set function that fetches the modified data, specified as a name or naming rule. A naming rule includes a combination of text and tokens. Valid tokens are listed in this table.

Token	Description
$N	Name of associated data element (required)
$R	Name of root model
$M	Mangle text that ensures uniqueness
$U	User token text. See “Identifier Format Control”.

Dependencies

This property is enabled only when you set DataAccess to Function and AllowedAccess to ReadWrite or WriteOnly.

DifferentInstanceDataSettings — Specification to assign separate storage settings for single-instance data and multi-instance data

false (default) | true

Specification for the storage class to use either the storage settings that you specify for single-instance data or the settings that you specify for multi-instance data. When you apply the storage class to a data item, the Embedded Coder Dictionary determines if it is a single-instance storage class or a multi-instance storage class by the type of data and by the context of the model within the model reference hierarchy.

Dependencies

Selecting this property enables the properties SingleInstanceStorageType, MultiInstanceStorageType, and MultiInstanceStructureTypeName, MultiInstanceStructureInstanceName.

StorageType — Specification to aggregate data into a structure

Unstructured (default) | Structured

Specification to aggregate the data that uses the storage class into a structure in the generated code. Each data element appears in the code as a field of the structure. To create a structure, use Structured.

Dependencies

Setting this property to Structured enables StructureTypeName and StructureInstanceName.

StructureTypeName — Name of structure type

RNGM (default) | character vector | string scalar

Name of the structure type in the generated code, specified as a name or a naming rule. A naming rule includes a combination of text and tokens. Valid tokens are listed in this table.

Token	Description
$R	Name of root model

Token	Description
$N	Base name of associated function, such as `step`
$G	Name of storage class
$U	User token text, which you specify for a model as described in “Identifier Format Control”
$M	Name-mangling text inserted, if necessary, to avoid name collisions

Dependencies

Setting `StorageType` to `Structured` enables this property.

`StructureInstanceName` — Name of structure variable

`$G$N$M` (default) | character vector | string scalar

Name of the structure variable in the generated code, specified as a name or a naming rule. A naming rule includes a combination of text and tokens. Valid tokens are listed in this table.

Token	Description
$R	Name of root model
$N	Base name of associated function, such as `step`
$G	Name of storage class
$U	User token text, which you specify for a model as described in “Identifier Format Control”
$M	Name-mangling text inserted, if necessary, to avoid name collisions

Dependencies

Setting `StorageType` to `Structured` enables this property.

`SingleInstanceStorageType` — Specification to aggregate single-instance data into a structure

`Structured` (default) | `Unstructured`

Specification to aggregate the single-instance data that uses the storage class into a structure in the generated code. Each data element appears in the code as a field of the structure. To create a structure, use `Structured`.

Dependencies

`SingleInstanceStructureTypeName` — Name of structure type for single-instance data

`$R$N$G$M` (default) | character vector | string scalar

Name of the structure type in the generated code, specified as a name or a naming rule. A naming rule includes a combination of text and tokens. Valid tokens are listed in this table.

Token	Description
$R	Name of root model
$N	Base name of associated function, such as `step`

Token	Description
$G	Name of storage class
$U	User token text, which you specify for a model as described in "Identifier Format Control"
$M	Name-mangling text inserted, if necessary, to avoid name collisions

Dependencies

Setting SingleInstanceStorageType to Structured enables this property.

SingleInstanceStructureInstanceName — Name of structure variable for single-instance data

GN$M (default) | character vector | string scalar

Name of the structure variable in the generated code, specified as a name or a naming rule. A naming rule includes a combination of text and tokens. Valid tokens are listed in this table.

Token	Description
$R	Name of root model
$N	Base name of associated function, such as step
$G	Name of storage class
$U	User token text, which you specify for a model as described in "Identifier Format Control"
$M	Name-mangling text inserted, if necessary, to avoid name collisions

Dependencies

Setting SingleInstanceStorageType to Structured enables this property.

MultiInstanceStorageType — Specification to aggregate multi-instance data into a structure

Structured (default)

Specification to aggregate the single-instance data that uses the storage class into a structure in the generated code. Each data element appears in the code as a field of the structure. You cannot change the value of this property.

Dependencies

Setting the property UseDifferentPropSettingsForInstanceData to true enables this property.

MultiInstanceStructureTypeName — Name of structure type for multi-instance data

RNGM (default) | character vector | string scalar

Name of the structure type in the generated code, specified as a name or a naming rule. A naming rule includes a combination of text and tokens. Valid tokens are listed in this table.

Token	Description
$R	Name of root model

Token	Description
`$N`	Base name of associated function, such as `step`
`$G`	Name of storage class
`$U`	User token text, which you specify for a model as described in “Identifier Format Control”
`$M`	Name-mangling text inserted, if necessary, to avoid name collisions

Dependencies

`MultiInstanceStructureInstanceName` — Name of structure variable for multi-instance data

`$G$N$M` (default) | character vector | string scalar

Name of the structure variable in the generated code, specified as a name or a naming rule. A naming rule includes a combination of text and tokens. Valid tokens are listed in this table.

Token	Description
`$R`	Name of root model
`$N`	Base name of associated function, such as `step`
`$G`	Name of storage class
`$U`	User token text, which you specify for a model as described in “Identifier Format Control”
`$M`	Name-mangling text inserted, if necessary, to avoid name collisions

Dependencies

`DataInit` — How to initialize data

`Dynamic` (default) | `Static` | `None`

Specification that the generated codes initialize the data.

Dependencies

`MemorySection` — Location in memory to allocate data

`None` (default) | `coder.dictionary.Entry` object

Location in memory to allocate data, specified as a `coder.dictionary.Entry` object that represents a memory section that exists in the Embedded Coder Dictionary. For information about memory sections, see “Control Data and Function Placement in Memory by Inserting Pragmas”.

`PreserveDimensions` — Specification to preserve dimensions of multidimensional arrays

`false` (default) | `true`

Specification for the storage class to preserve dimensions of multidimensional arrays in the generated code. For more information, see “Preserve Dimensions of Multidimensional Arrays in Generated Code”.

`Const` — Specification to apply `const` qualifier
`false` (default) | `true`

Specification to apply the `const` qualifier to the data.

Dependencies

`Volatile` — Specification to apply `volatile` qualifier
`false` (default) | `true`

Specification to apply the `volatile` qualifier to the data.

`OtherQualifier` — Specification to apply a custom qualifier
character vector | string scalar

Specification to apply a custom qualifier to the data. For example, some memory architectures support qualifiers `far` and `huge`.

Do not use this property to apply the keyword `static`. Instead, use the built-in storage class `FileScope`, which you cannot apply with the Code Mappings editor. See "Choose Storage Class for Controlling Data Representation in Generated Code".

`AccessibleByParameters` — Whether to allow usage with model parameters
`false` (default) | `true`

Specification indicating whether to allow usage of the storage class with model parameters.

Dependencies

`AccessibleBySignals` — Whether to allow usage with model signals
`true` (default) | `false`

Specification indicating whether to allow usage of the storage class with model signals.

Dependencies

Function Customization Templates

`Name` — Name of function template
`FunctionTemplate1` (default) | character vector | string scalar

Name of the template. The name must be unique among the function templates in the dictionary. Embedded Coder provides the built-in templates listed in this table.

Template	Description
`ModelFunction`	In the Code Mappings editor, use for entry-point functions for initialization, execution, termination, and reset (see "Configure Default Code Generation for Functions")
`UtilityFunction`	In the Code Mappings editor, use for shared utility functions (see "Configure Default Code Generation for Functions")

`Description` — Purpose and functionality of function template
character vector | string scalar

Custom text that you can use to describe the purpose and functionality of the function template.

`FunctionName` — Names of generated functions
`$R$N` (default) | character vector | string scalar

Names of the functions in the generated code, specified as a naming rule. A naming rule includes a combination of text and tokens. Valid tokens are listed in this table.

Token	Description
`$R`	Name of root model
`$N`	Base name of associated function, such as `step`
`$U`	User token text, which you specify for a model as described in "Identifier Format Control"
`$C`	For shared utility functions, a checksum inserted to avoid name collisions
`$M`	Name-mangling text inserted, if necessary, to avoid name collisions

`MemorySection` — Location in memory to allocate function
`None` (default) | `coder.dictionary.Entry` object

Location in memory to allocate function, specified as a `coder.dictionary.Entry` object that represents a memory section that exists in the Embedded Coder Dictionary. For information about memory sections, see "Control Data and Function Placement in Memory by Inserting Pragmas".

Memory Sections

`Name` — Name of memory section
character vector | string scalar

Name of the memory section. The name must be unique among the memory sections in the dictionary. Embedded Coder provides the built-in memory sections listed in this table.

Memory Section	Description
`MemConst`	Apply the storage type qualifier `const` to the data.
`MemVolatile`	Apply the storage type qualifier `volatile` to the data.
`MemConstVolatile`	Apply the storage type qualifiers `const` and `volatile` to the data.

`Description` — Purpose and functionality of memory section
character vector | string scalar

Custom text that you can use to describe the purpose and functionality of the memory section.

`Comment` — Comment to insert in the generated code
character vector | string scalar

Code comment that the code generator includes with the pragmas or other decorations that you specify with `PreStatement` and `PostStatement`.

`PreStatement` — Code to insert before data or function code
character vector | string scalar

Code, such as pragmas, to insert before the definitions and declarations of the data or functions that are in the memory section.

You can use the token `$R` to represent the name of the model that uses the memory section.

When you set `StatementsSurround` to `EachVariable`, you can use the token `$N` to represent the name of each variable or function that uses the memory section.

`PostStatement` — Code to insert after data or function code
character vector | string scalar

Code, such as pragmas, to insert after the definitions and declarations of the data or functions that are in the memory section.

You can use the token `$R` to represent the name of the model that uses the memory section.

When you set `StatementsSurround` to `EachVariable`, you can use the token `$N` to represent the name of each variable or function that uses the memory section.

`StatementsSurround` — Specification to wrap data and functions separately or in a group
`EachVariable` (default) | `AllVariables`

Specification to insert code statements (`PreStatement` and `PostStatement`):

Methods

Public Methods

get	Get value of code definition property
set	Set value of code definition property
getAvailableProperties	Return properties for code definition
deleteEntry	Delete Embedded Coder Dictionary entry
valid	Determine if coder.dictionary.Entry object represents a valid code definition

Examples

Create Definition in Embedded Coder Dictionary

Open the model `rtwdemo_roll` and represent the Embedded Coder Dictionary by using the `coder.Dictionary` object `coderDictionary`. Use this object to access the Storage Classes section of the dictionary, which contains the storage class definitions.

```
rtwdemo_roll
coderDictionary = coder.dictionary.open('rtwdemo_roll');
```

Create a `coder.dictionary.Section` object that represents the Storage Classes section of the Embedded Coder Dictionary.

```
storageClassesSect = getSection(coderDictionary, 'StorageClasses');
```

Add a definition named `MyStorageClass` to the Storage Classes section. The storage class definition uses the default property settings. You can change these settings by using the `set` function.

```
newEntry = addEntry(storageClassesSect,'MyStorageClass')
```

```
newEntry = 

  Entry with properties:

          Name: 'MyStorageClass'
    DataSource: 'rtwdemo_roll'
```

See Also

Embedded Coder Dictionary | `coder.Dictionary` | `coder.dictionary.Section`

Introduced in R2019b

deleteEntry

Class: `coder.dictionary.Entry`
Package: `coder.dictionary`

Delete Embedded Coder Dictionary entry

Syntax

`deleteEntry(entryObj)`

Description

`deleteEntry(entryObj)` deletes an Embedded Coder Dictionary definition that the entry `entryObj` represents.

Input Arguments

`entryObj` — Embedded Coder Dictionary entry
`coder.dictionary.Entry` object

Embedded Coder Dictionary entry, specified as a `coder.dictionary.Entry` object.

Examples

Delete Storage Class from Embedded Coder Dictionary

Open the model `rtwdemo_roll` and represent the Embedded Coder Dictionary by using a `coder.Dictionary` object. Use this object to access the Storage Classes section of the dictionary and represent the section by using a `coder.dictionary.Section` object.

```
rtwdemo_roll
coderDictionary = coder.dictionary.open('rtwdemo_roll');
storageClassesSect = getSection(coderDictionary, 'StorageClasses');
```

Represent the example storage class `ParamStruct` by using a `coder.dictionary.Entry` object.

```
entryObj = getEntry(storageClassesSect,'ParamStruct');
```

Delete the `ParamStruct` storage class.

```
deleteEntry(entryObj)
```

See Also

`coder.Dictionary` | `coder.dictionary.Entry` | `coder.dictionary.Section`

Introduced in R2019b

get

Class: coder.dictionary.Entry
Package: coder.dictionary

Get value of code definition property

Syntax

```
propValue = get(entryObj,propName)
```

Description

propValue = get(entryObj,propName) returns the value of the property propName for the code definition that entryObj represents.

Input Arguments

entryObj — Embedded Coder Dictionary entry
coder.dictionary.Entry object

Embedded Coder Dictionary entry that represents the code definition, specified as a coder.dictionary.Entry object. Before you use this function, represent the code definition by using a coder.dictionary.Entry object. Use, for example, the addEntry function.

propName — property name
character vector | string scalar

Property of code definition, specified as a character vector or string scalar.

Output Arguments

propValue — property value
string | coder.dictionary.Entry object

Value of the property propName for the code definition that entryObj represents, returned as a string or a coder.dictionary.Entry object. For storage classes and function customization templates, the MemorySection property can have a value that is a coder.dictionary.Entry object representing a memory section definition.

Examples

Return Property Value of Storage Class Definition

Open the model rtwdemo_roll and represent the Embedded Coder Dictionary by using a coder.Dictionary object. Use this object to access the Storage Classes section of the dictionary and represent the section with a coder.dictionary.Section object.

```
rtwdemo_roll
coderDictionary = coder.dictionary.open('rtwdemo_roll');
storageClassesSect = getSection(coderDictionary, 'StorageClasses');
```

Represent the example storage class `ParamStruct` by using a `coder.dictionary.Entry` object.

```
entryObj = getEntry(storageClassesSect,'ParamStruct');
```

Return the value of the `DataInit` property of the storage class definition. This storage class uses static data initialization.

```
dataInitVal = get(entryObj,'DataInit')
```

```
dataInitVal =

    'Static'
```

See Also

`coder.Dictionary` | `coder.dictionary.Entry` | `coder.dictionary.Section`

Introduced in R2019b

getAvailableProperties

Class: coder.dictionary.Entry
Package: coder.dictionary

Return properties for code definition

Syntax

```
properties = getAvailableProperties(entryObj)
```

Description

`properties = getAvailableProperties(entryObj)` returns a cell array of the properties of the code definition that `entryObj` represents.

Input Arguments

`entryObj` — Embedded Coder Dictionary entry
`coder.dictionary.Entry` object

Embedded Coder Dictionary entry, specified as a `coder.dictionary.Entry` object.

Output Arguments

`properties` — code definition properties
cell array of property names

Code definition properties available for the definition that `entryObj` represents, returned as a cell array of character vectors. For a list of properties available for each type of code definition, see Embedded Coder Dictionary.

Examples

Get Properties of Storage Class Definition

Open the model `rtwdemo_roll` and represent the Embedded Coder Dictionary by using a `coder.Dictionary` object. Use this object to access the Storage Classes section of the dictionary and represent the section by using a `coder.dictionary.Section` object.

```
rtwdemo_roll
coderDictionary = coder.dictionary.open('rtwdemo_roll');
storageClassesSect = getSection(coderDictionary, 'StorageClasses');
```

Represent the example storage class `ParamStruct` by using a `coder.dictionary.Entry` object.

```
entryObj = getEntry(storageClassesSect,'ParamStruct');
```

Get a list of the properties of the `ParamStruc` storage class. To get and set the values of these properties, use the `get` and `set` methods.

```
properties = getAvailableProperties(entryObj)
```

See Also

coder.Dictionary | coder.dictionary.Entry | coder.dictionary.Section

Introduced in R2019b

set

Class: `coder.dictionary.Entry`
Package: `coder.dictionary`

Set value of code definition property

Syntax

`set(entryObj,Name1,Value1,...,NameN,ValueN)`

Description

`set(entryObj,Name1,Value1,...,NameN,ValueN)` sets the properties to the specified values for the Embedded Coder Dictionary definition that `entryObj` represents.

Input Arguments

`entryObj` — Embedded Coder Dictionary entry
`coder.dictionary.Entry` object

Embedded Coder Dictionary entry that represents the code definition, specified as a `coder.dictionary.Entry` object. Before you use this function, represent the code definition by using a `coder.dictionary.Entry` object. Use, for example, the `addEntry` function.

`Name1,Value1,...,NameN,ValueN` — Property specifications
name-value pairs representing properties and values

Property specifications for the definition, specified as one or more name-value pairs representing names and values of properties of the code definition. For a list of the properties of an Embedded Coder Dictionary definition, see `coder.dictionary.Entry`.

Example: `'MemorySection','memSect1'`

Example: `'DataScope','Exported'`

Examples

Set Property Value of Storage Class Definition

Open the model `rtwdemo_roll` and represent the Embedded Coder Dictionary by using a `coder.Dictionary` object. Use this object to access the Storage Classes section of the dictionary and represent the section by using a `coder.dictionary.Section` object.

```
rtwdemo_roll
coderDictionary = coder.dictionary.open('rtwdemo_roll');
storageClassesSect = getSection(coderDictionary, 'StorageClasses');
```

Represent the example storage class `ParamStruct` by using a `coder.dictionary.Entry` object.

```
entryObj = getEntry(storageClassesSect,'ParamStruct');
```

Change the value of the HeaderFile property of the storage class definition from $N.h to $N_params.h.

```
set(entryObj,'HeaderFile','$N_params.h')
```

See Also

coder.Dictionary | coder.dictionary.Entry | coder.dictionary.Section

Introduced in R2019b

valid

Class: coder.dictionary.Entry
Package: coder.dictionary

Determine if coder.dictionary.Entry object represents a valid code definition

Syntax

```
tf = valid(entryObj)
```

Description

tf = valid(entryObj) returns true if the coder.dictionary.Entry object entryObj is valid. A coder.dictionary.Entry object is valid if it represents an existing code definition. When you remove the code definition that the entry represents, the coder.dictionary.Entry object is not valid.

Input Arguments

entryObj — Embedded Coder Dictionary entry
coder.dictionary.Entry object

Embedded Coder Dictionary entry, specified as a coder.dictionary.Entry object.

Output Arguments

tf — True or false result
1 | 0 | logical array

True or false result, returned as a 1 or 0 of data type logical.

Examples

Check if Entry Object Is Valid

Open the model rtwdemo_roll and represent the Embedded Coder Dictionary by using a coder.Dictionary object. Use this object to access the Storage Classes section of the dictionary and represent the section by using a coder.dictionary.Section object.

```
rtwdemo_roll
coderDictionary = coder.dictionary.open('rtwdemo_roll');
storageClassesSect = getSection(coderDictionary, 'StorageClasses');
```

Represent the example storage class ParamStruct by using a coder.dictionary.Entry object.

```
entryObj = getEntry(storageClassesSect,'ParamStruct');
```

Check if the coder.dictionary.Entry object in the base workspace represents a valid code definition.

```
valid(entryObj)

ans =

  logical

   1
```

Remove the code definition from the Embedded Coder Dictionary and check the `coder.dictionary.Entry` object again. When you remove the code definition, the `coder.dictionary.Entry` object is not valid.

```
deleteEntry(storageClassesSect,'ParamStruct');
valid(entryObj)

ans =

  logical

   0
```

See Also

`coder.Dictionary` | `coder.dictionary.Entry` | `coder.dictionary.Section`

Introduced in R2019b

addEntry

Class: `coder.dictionary.Section`
Package: `coder.dictionary`

Add a new entry to Embedded Coder Dictionary section

Syntax

```
addEntry(sectionObj,defName)
entryObj = addEntry(sectionObj,entryName)
```

Description

`addEntry(sectionObj,defName)` adds a definition named `defName` to the Embedded Coder Dictionary section `sectionObj`, a `coder.dictionary.Section` object.

`entryObj = addEntry(sectionObj,entryName)` returns a `coder.dictionary.Entry` object that represents the new Embedded Coder Dictionary definition.

Input Arguments

`sectionObj` — Embedded Coder Dictionary section
`coder.dictionary.Section` object

Section in the Embedded Coder Dictionary, specified as a `coder.dictionary.Section` object. The section name identifies the type of code definition that `addEntry` creates.

`defName` — Name of Embedded Coder Dictionary definition
character vector | string scalar

Name of newly created Embedded Coder Dictionary definition, specified as a character vector or string scalar.

Example: `'StorageClass2'`

Output Arguments

`entryObj` — Embedded Coder Dictionary entry
`coder.dictionary.Entry` object

New Embedded Coder Dictionary entry, returned as a `coder.dictionary.Entry` object. The new entry represents the new code definition in the section of the Embedded Coder Dictionary.

Examples

Add Storage Class to Embedded Coder Dictionary

Open the model `rtwdemo_roll` and represent the Embedded Coder Dictionary by using a `coder.Dictionary` object. Use this object to access the Storage Classes section of the dictionary, which contains the storage class definitions.

```
rtwdemo_roll
coderDictionary = coder.dictionary.open('rtwdemo_roll');
```

Create a `coder.dictionary.Section` object that represents the Storage Classes section of the Embedded Coder Dictionary.

```
storageClassesSect = getSection(coderDictionary, 'StorageClasses');
```

Add a storage class definition named `MyStorageClass` to the Storage Classes section. The storage class definition uses the default property settings. You can change these settings by using the `set` function.

```
newEntry = addEntry(storageClassesSect,'MyStorageClass')
```

```
newEntry = 

  Entry with properties:

          Name: 'MyStorageClass'
    DataSource: 'rtwdemo_roll'
```

See Also

`coder.Dictionary` | `coder.dictionary.Entry` | `coder.dictionary.Section`

Introduced in R2019b

deleteEntry

Class: `coder.dictionary.Section`
Package: `coder.dictionary`

Delete Embedded Coder Dictionary entry

Syntax

```
deleteEntry(sectionObj,defName)
```

Description

`deleteEntry(sectionObj,defName)` deletes an Embedded Coder Dictionary definition that has the name `defName` from the section `sectionObj`, a `coder.dictionary.Section` object.

Input Arguments

`sectionObj` — Embedded Coder Dictionary section
`coder.dictionary.Section` object

Section in the Embedded Coder Dictionary that contains its definition, specified as a `coder.dictionary.Section` object.

`defName` — Name of Embedded Coder Dictionary definition
character vector | string scalar

Name of Embedded Coder Dictionary definition, specified as a character vector or string scalar. To get a list of definitions in a section, use the methods `find` and `get`.

Example: `'StorageClass2'`

Examples

Delete Storage Class from Embedded Coder Dictionary

Open the model `rtwdemo_roll` and represent the Embedded Coder Dictionary by using a `coder.Dictionary` object. Use this object to access the Storage Classes section of the dictionary, which contains the storage class definitions.

```
rtwdemo_roll
coderDictionary = coder.dictionary.open('rtwdemo_roll');
```

Create a `coder.dictionary.Section` object that represents the Storage Classes section of the Embedded Coder Dictionary.

```
storageClassesSect = getSection(coderDictionary, 'StorageClasses')
```

```
storageClassesSect = 

  Section with properties:
```

```
    Name: 'StorageClasses'
```

Get a list of the storage class definitions in the section. Use the `find` method to get the `coder.dictionary.Entry` objects in the section. Then, use the `get` method to list the names of the definitions that the entries represent.

```
entryObjects = find(coderDictionary, 'StorageClasses');
get(entryObjects,'Name')
```

```
ans =

  1×19 cell array

  Columns 1 through 4

    {'ExportedGlobal'}    {'ImportedExtern'}    {'ImportedExternP…'}    {'BitField'}

  Columns 5 through 9

    {'Const'}    {'Volatile'}    {'ConstVolatile'}    {'Define'}    {'ImportedDefine'}

  Columns 10 through 13

    {'ExportToFile'}    {'ImportFromFile'}    {'FileScope'}    {'Localizable'}

  Columns 14 through 18

    {'Struct'}    {'GetSet'}    {'CompilerFlag'}    {'Reusable'}    {'SignalStruct'}

  Column 19

    {'ParamStruct'}
```

Remove the example storage class `ParamStruct` from the dictionary.

```
deleteEntry(storageClassesSect,'ParamStruct');
```

See Also

`coder.Dictionary` | `coder.dictionary.Entry` | `coder.dictionary.Section`

Introduced in R2019b

exist

Class: coder.dictionary.Section
Package: coder.dictionary

Determine if code definition exists in Embedded Coder Dictionary section

Syntax

```
tf = exist(sectionObj,defName)
```

Description

`tf = exist(sectionObj,defName)` returns `true` if the Embedded Coder Dictionary section represented by `sectionObj` contains a definition with the name `defName`.

Input Arguments

sectionObj — Embedded Coder Dictionary section
coder.dictionary.Section object

Section in the Embedded Coder Dictionary, specified as a `coder.dictionary.Section` object.

defName — Name of Embedded Coder Dictionary definition
character vector | string scalar

Name of Embedded Coder Dictionary definition, specified as a character vector or string scalar.

Example: 'StorageClass2'

Output Arguments

tf — True or false result
1 | 0 | logical array

True or false result, returned as a `1` or `0` of data type `logical`.

Examples

Check If Storage Class Exists

Open the model `rtwdemo_roll` and represent the Embedded Coder Dictionary by using a `coder.Dictionary` object. Use this object to access the Storage Classes section of the dictionary, which contains the storage class definitions.

```
rtwdemo_roll
coderDictionary = coder.dictionary.open('rtwdemo_roll');
```

Create a `coder.dictionary.Section` object that represents the Storage Classes section of the Embedded Coder Dictionary.

```
storageClassesSect = getSection(coderDictionary, 'StorageClasses');
```

Check if the Storage Classes section contains the example storage class `SignalStruct`.

```
exist(storageClassesSect,'SignalStruct')
```

```
ans =

  logical

   1
```

Remove the storage class, and then see whether the definition still exists.

```
deleteEntry(storageClassesSect,'SignalStruct');
exist(storageClassesSect,'SignalStruct')
```

```
ans =

  logical

   0
```

See Also

`coder.Dictionary` | `coder.dictionary.Entry` | `coder.dictionary.Section`

Introduced in R2019b

find

Class: `coder.dictionary.Section`
Package: `coder.dictionary`

Search in Embedded Coder Dictionary section

Syntax

`foundEntries = find(sectionObj,Name1,Value1,...,NameN,ValueN)`

Description

`foundEntries = find(sectionObj,Name1,Value1,...,NameN,ValueN)` searches the Embedded Coder Dictionary section `sectionObj` by using search criteria `Name1,Value1,…,NameN,ValueN`, and returns an array of matching entries that were found in that section.

Input Arguments

`sectionObj` — Embedded Coder Dictionary section
`coder.dictionary.Section` object

Section to search in the Embedded Coder Dictionary, specified as a `coder.dictionary.Section` object.

`Name1,Value1,...,NameN,ValueN` — Search criteria
name-value pairs representing properties

Search criteria, specified as one or more name-value pairs representing names and values of properties of the definitions in the Embedded Coder Dictionary section. For a list of the properties of an Embedded Coder Dictionary entry, see `coder.dictionary.Entry`.

Example: `'MemorySection','memSect1'`

Example: `'DataScope','Exported'`

Output Arguments

`foundEntries` — Embedded Coder Dictionary entries matching search criteria
array of `coder.dictionary.Entry` objects

Embedded Coder Dictionary entries that match the specified search criteria, returned as an array of `coder.dictionary.Entry` objects.

Examples

Search Embedded Coder Dictionary for Exported Storage Classes

Open the model `rtwdemo_roll` and create an Embedded Coder Dictionary in the model. Represent the Embedded Coder Dictionary by using the `coder.Dictionary` object `coderDictionary`. Use

this object to access the Storage Classes section of the dictionary, which contains the storage class definitions.

```
rtwdemo_roll
coderDictionary = coder.dictionary.open('rtwdemo_roll');
```

Create a `coder.dictionary.Section` object that represents the Storage Classes section of the Embedded Coder Dictionary.

```
storageClassesSect = getSection(coderDictionary, 'StorageClasses');
```

Find the storage classes that have `DataScope` set to `Exported`.

```
exportedSCs = find(storageClassesSect,'DataScope','Exported')
```

```
exportedSCs =

  1×2 Entry array with properties:

    Name
    DataSource
```

See Also

`coder.Dictionary` | `coder.dictionary.Entry` | `coder.dictionary.Section`

Introduced in R2019b

getEntry

Class: `coder.dictionary.Section`
Package: `coder.dictionary`

Return `coder.dictionary.Entry` object that represents an Embedded Coder Dictionary definition

Syntax

`entryObj = getEntry(sectionObj,defName)`

Description

`entryObj = getEntry(sectionObj,defName)` returns a `coder.dictionary.Entry` object representing the Embedded Coder Dictionary definition that has the name `defName` found in the section `sectionObj`, a `coder.dictionary.Section` object.

Input Arguments

`sectionObj` — Embedded Coder Dictionary section
`coder.dictionary.Section` object

Embedded Coder Dictionary section containing the code definition, specified as a `coder.dictionary.Section` object. The section name identifies the type of code definition.

`defName` — Name of Embedded Coder Dictionary definition
character vector | string scalar

Name of Embedded Coder Dictionary definition, specified as a character vector or string scalar.

Example: `'StorageClass2'`

Output Arguments

`entryObj` — Embedded Coder Dictionary entry
`coder.dictionary.Entry` object

Embedded Coder Dictionary entry, returned as a `coder.dictionary.Entry` object. The entry represents the code definition in the Embedded Coder Dictionary section.

Examples

Get Storage Class from Embedded Coder Dictionary

Open the model `rtwdemo_roll` and represent the Embedded Coder Dictionary by using a `coder.Dictionary` object. Use this object to access the Storage Classes section of the dictionary, which contains the storage class definitions.

```
rtwdemo_roll
coderDictionary = coder.dictionary.open('rtwdemo_roll');
```

Create a `coder.dictionary.Section` object that represents the Storage Classes section of the Embedded Coder Dictionary.

```
storageClassesSect = getSection(coderDictionary, 'StorageClasses');
```

Represent the example storage class `ParamStruct` by using a `coder.dictionary.Entry` object.

```
entryObj = getEntry(storageClassesSect,'ParamStruct')
```

```
Entry with properties:

          Name: 'ParamStruct'
    DataSource: 'rtwdemo_roll'
```

See Also

`coder.Dictionary` | `coder.dictionary.Entry` | `coder.dictionary.Section`

Introduced in R2019b

coder.dictionary.Section class

Package: `coder.dictionary`

Configure Embedded Coder Dictionary section

Description

An object of the `coder.dictionary.Section` class represents one section of an Embedded Coder Dictionary, such as Storage Classes or Memory Sections. The object enables you to perform operations on the section such as adding or deleting entries.

A `coder.Dictionary` object contains three `coder.dictionary.Section` objects, which represent the sections of an Embedded Coder Dictionary: Storage Classes, Memory Sections, and Function Customization Templates. A `coder.dictionary.Section` object contains `coder.dictionary.Entry` objects, which represent the definitions stored in that section of the Embedded Coder Dictionary. The name of the section identifies the type of code definitions that the section contains. To access the sections of an Embedded Coder Dictionary, use a `coder.Dictionary` object. To access the dictionary entries within the section, use a `coder.dictionary.Section` object.

Creation

The function `getSection` creates a `coder.dictionary.Section` object.

Properties

`Name` — Name of Embedded Coder Dictionary Section
`'StorageClasses'` | `'MemorySections'` | `'FunctionCustomizationTemplates'`

Name of Embedded Coder Dictionary Section, returned as a character vector. This property is read-only.

Example: `'StorageClasses'`

Methods

Public Methods

addEntry	Add a new entry to Embedded Coder Dictionary section
getEntry	Return coder.dictionary.Entry object that represents an Embedded Coder Dictionary definition
copyEntry	Copy Embedded Coder Dictionary entry
deleteEntry	Delete Embedded Coder Dictionary entry
exist	Determine if code definition exists in Embedded Coder Dictionary section
find	Search in Embedded Coder Dictionary section

Examples

Create Embedded Coder Dictionary Section Object

Open the model `rtwdemo_roll` and represent the Embedded Coder Dictionary by using the `coder.Dictionary` object `coderDictionary`.

```
rtwdemo_roll
coderDictionary = coder.dictionary.open('rtwdemo_roll');
```

Create a `coder.dictionary.Section` object that represents the Memory Sections section of the Embedded Coder Dictionary.

```
memorySectionsSect = getSection(coderDictionary, 'MemorySections')
```

```
memorySectionsSect =

  Section with properties:

    Name: 'MemorySections'
```

The section contains `coder.dictionary.Entry` objects, each of which represents a built-in memory section definition.

See Also

Embedded Coder Dictionary | `coder.Dictionary` | `coder.dictionary.Entry`

Introduced in R2019b

coder.replace

Replace current MATLAB function implementation with code replacement library function in generated code

Syntax

`coder.replace(ifNoReplacement)`

Description

`coder.replace(ifNoReplacement)` replaces the current function implementation with a code replacement library function.

During code generation, when you call `coder.replace` in a MATLAB function, the code generator performs a code replacement library lookup for the function signature:

```
[y1_type, y2_type,..., yn_type]=fcn(x1_type, x2_type,...,xn_type)
```

The input data types are `x1_type, x2_type,...,xn_type` and the output types, derived from the implementation, are `y1_type, y2_type,..., yn_type`. If a match for the MATLAB function is found in a registered code replacement library, the contents of the MATLAB function are discarded and replaced with a call to the code replacement library function. If a match is not found, the code generates without replacement.

`coder.replace` only affects code generation and does not alter MATLAB code or MEX function generation. `coder.replace` is intended to replace a MATLAB function that has behavior equivalent to its replacement function implementation. If the MATLAB function body is empty or not equivalent to the replacement function implementation, it may be eliminated from the generated code. The MATLAB function prior to replacement is used for simulation. You are responsible for verifying the numeric result of simulation and code generation after replacement.

Examples

Replace a MATLAB Function with Custom Code

Replace a MATLAB function with a custom implementation that is registered in the code replacement library.

Write a MATLAB function, `calculate`, that you want to replace with a custom implementation, `replacement_calculate_impl.c`, in the generated code.

```
function y = calculate(x)
% Search in the code replacement library for replacement
% and use replacement function if available
% Error if not found
  coder.replace('-errorifnoreplacement');
  y = sqrt(x);
end
```

Write a MATLAB function, `top_function`, that calls `calculate`.

```
function out = top_function(in)
  p = calculate(in);
  out = exp(p);
end
```

Create a file named `crl_table_calculate.m` that describes the function entries for a code replacement table. The replacement function `replacement_calculate_impl.c` and header file `replacement_calculate_impl.h` must be on the path.

```
hLib = RTW.TflTable;

%---------- entry: calculate -----------
hEnt = RTW.TflCFunctionEntry;
setTflCFunctionEntryParameters(hEnt, ...
  'Key', 'calculate', ...
  'Priority', 100, ...
  'ArrayLayout', 'COLUMN_MAJOR', ...
  'ImplementationName', ...
  'replacement_calculate_impl', ...
  'ImplementationHeaderFile', ...
  'replacement_calculate_impl.h', ...
  'ImplementationSourceFile', ...
  'replacement_calculate_impl.c')
% Conceptual Args

arg = getTflArgFromString(hEnt, 'y1','double');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEnt, arg);

arg = getTflArgFromString(hEnt, 'u1','double');
addConceptualArg(hEnt, arg);

% Implementation Args

arg = getTflArgFromString(hEnt, 'y1','double');
arg.IOType = 'RTW_IO_OUTPUT';
hEnt.Implementation.setReturn(arg);

arg = getTflArgFromString(hEnt, 'u1','double');
hEnt.Implementation.addArgument(arg);

addEntry(hLib, hEnt);
```

Create an `rtwTargetInfo` file:

```
function rtwTargetInfo(tr)
% rtwTargetInfo function to register a code
% replacement library (CRL)
% for use with codegen

  % Register the CRL defined in local function locCrlRegFcn
  tr.registerTargetInfo(@locCrlRegFcn);

end % End of RTWTARGETINFO

function thisCrl = locCrlRegFcn

  % Instantiate a CRL registry entry
```

```
  thisCrl = RTW.TflRegistry;

  % Define the CRL properties
  thisCrl.Name = 'My calculate Example';
  thisCrl.Description = 'Demonstration of function replacement';
  thisCrl.TableList = {'crl_table_calculate'};
  thisCrl.BaseTfl = 'C89/C90 (ANSI)';
  thisCrl.TargetHWDeviceType = {'*'};

end % End of LOCCRLREGFCN
```

Refresh registration information. At the MATLAB command line, enter:

```
RTW.TargetRegistry.getInstance('reset');
```

Because the data type of `x` and `y` is `double`, `coder.replace` searches for `double = calculate(double)` in the Code Replacement Library. If it finds a match, `codegen` generates the following code:

```
real_T top_function(real_T in)
{
  real_T p;
  p = replacement_calculate_impl(in);
  return exp(p);
}
```

In the generated code, the replacement function `replacement_calculate_impl` replaces the MATLAB function `calculate`.

Input Arguments

`ifNoReplacement` — Selects whether the code generator produces a warning or error when no match is found
`'-errorifnoreplacement'` | `'-warnifnoreplacement'`

The *`ifNoReplacement`* argument selects whether the code generator issues an error or warning when no match is found. If this argument is omitted, the code generator does not issue an error or warning.

`coder.replace('-errorifnoreplacement')` replaces the current function implementation with a code replacement library function. If a match is not found, code generation stops. An error message describing the code replacement library lookup failure is generated.

`coder.replace('-warnifnoreplacement')` replaces the current function implementation with a code replacement library function. If match is not found, code is generated for the current function. A warning describing the code replacement library lookup failure is generated during code generation.

Example: `coder.replace()`

Tips

- `coder.replace` requires an Embedded Coder license.
- `coder.replace` is a code generation function and does not alter MATLAB code or MEX function generation.

- `coder.replace` is not intended to be called multiple times within a function.
- `coder.replace` is not intended to be used within conditional expressions and loops.
- `coder.replace` does not support saturation and rounding modes during code replacement library lookups.
- `coder.replace` does not support `varargout`.
- `coder.replace` does not support function replacement that requires data alignment.
- `coder.replace` does not support function replacement of MATLAB functions with variable-size inputs.

See Also

`codegen`

Topics

"Replace MATLAB Functions with Custom Code by Using the coder.replace Function"
"Define Code Replacement Library Optimizations"
"What Is Code Replacement Customization?"
"What Is Code Replacement?"

Introduced in R2012b

coder.report.generateCodeMetrics

Generate static code metrics report

Syntax

```
coder.report.generateCodeMetrics(model)
coder.report.generateCodeMetrics(subsystem)
coder.report.generateCodeMetrics( ___ ,Name,Value)
```

Description

`coder.report.generateCodeMetrics(model)` generates a static code metrics report for the code generated from `model` without generating a full code generation report.

You can also generate a static code metrics report in a code generation report by using the function `coder.report.generate` or by selecting **Generate Static Code Metrics for Model** in the Configuration Parameters dialog box.

`coder.report.generateCodeMetrics(subsystem)` generates the static code metrics report for the `subsystem`. The build folder for the subsystem must be present in the current working folder.

`coder.report.generateCodeMetrics( ___ ,Name,Value)` specifies options by using one or more `Name,Value` pair arguments.

Examples

Generate Static Code Metrics for Model

Generate code for a model, and then generate the static code metrics.

Open the model `rtwdemo_counter`.

```
open rtwdemo_counter
```

Turn off code generation report generation.

```
set_param('rtwdemo_counter','GenerateReport','off');
```

Build the model.

```
slbuild('rtwdemo_counter');
```

Generate a static code metrics report for the model.

```
coder.report.generateCodeMetrics('rtwdemo_counter');
```

Generate Static Code Metrics for Subsystem

Generate code and static code metrics for a subsystem.

Open the model `rtwdemo_counter`.

```
open rtwdemo_counter
```

Turn off code generation report generation.

```
set_param('rtwdemo_counter','GenerateReport','off');
```

Build the subsystem.

```
slbuild('rtwdemo_counter/Amplifier');
```

Generate a static code metrics report for the subsystem.

```
coder.report.generateCodeMetrics('rtwdemo_counter/Amplifier');
```

Specify Code Metrics Report File Name

Generate code metrics in a report that uses a custom file name.

Open the model `rtwdemo_counter`.

```
open rtwdemo_counter
```

Turn off code generation report generation.

```
set_param('rtwdemo_counter','GenerateReport','off');
```

Build the model.

```
slbuild('rtwdemo_counter');
```

Generate a static code metrics report for the model. Name the generated code metrics report `code_metrics.html`.

```
coder.report.generateCodeMetrics('rtwdemo_counter','FileName','code_metrics.html');
```

Input Arguments

`model` — Model name
character vector | string scalar

Model name specified as a character vector or string scalar.

Example: `'rtwdemo_counter'`

Data Types: `char`

`subsystem` — Subsystem name
character vector | string scalar

Subsystem name specified as a character vector or sting scalar.

Example: `'rtwdemo_counter/Amplifier'`

Data Types: `char`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'FileName','X:\code_metrics.html'` creates a static code metrics report named `code_metrics.html`.

`BuildDir` — Build folder
character vector | string scalar

Build folder that contains the generated code, specified as a character vector or string scalar.

Example: `'BuildDir','X:\rtwdemo_comments_ert_rtw'`

`FileName` — Name of report file
character vector | string scalar

Name of the generated static code metrics HTML file, specified as a character vector or string scalar.

Example: `'FileName','X:\code_metrics.html'`

See Also

`coder.report.generate`

Topics

"Static Code Metrics"
"Generate Static Code Metrics Report for Simulink Model"

Introduced in R2020b

coder.setupMISRAConfig

Configure parameters to improve generated code compliance with MISRA C and C++ guidelines

Syntax

`coder.setupMISRAConfig(cfg)`

Description

`coder.setupMISRAConfig(cfg)` sets up an Embedded Coder code generation configuration object with properties selected to improve the generated code compliance with MISRA® C:2012 and MISRA C++:2008 guidelines.

Examples

Configure Code Generation Configuration Object for Improved MISRA Compliance

Create an Embedded Coder code generation configuration object.

```
cfg = coder.config('lib', 'ecoder', true);
```

Set properties that might impact MISRA compliance.

```
coder.setupMISRAConfig(cfg);
```

The function `coder.setupMISRAConfig` sets property values according to the values shown in the table.

Property	Value for Improved MISRA Compliance
`CastingMode`	`'Standards'`
`CppNamespace`	Valid namespace name for C++
`DataTypeReplacement`	`'CoderTypedefs'`
`DynamicMemoryAllocation`	`'Off'`
`EnableRuntimeRecursion`	`false`
`EnableSignedLeftShifts`	`false`
`EnableSignedRightShifts`	`false`
`GenerateDefaultInSwitch`	`true`
`ParenthesesLevel`	`'Maximum'`
`TargetLangStandard`	`'C99 (ISO)'` for C, `'C++03 (ISO)'` for C++

If the `CppNamespace` property is unset, and the `TargetLang` property is `'C++'`, then `coder.setupMISRAConfig` sets the `CppNamespace` property to a default character vector, `'Codegen'`. Modify this value to a namespace name that is particular to your project.

Input Arguments

`cfg` — Embedded Coder code generation configuration object
`coder.EmbeddedCodeConfig` object

Embedded Coder configuration object for generating C/C++ code from MATLAB code. Create the object by using `coder.config`.

Example: cfg = coder.config('lib', 'ecoder',true)

See Also

Topics
"Generate C/C++ Code with Improved MISRA Compliance"

External Websites
`www.misra.org.uk`

Introduced in R2017b

coder.storageClass

Assign storage class to global variable

Syntax

```
coder.storageClass(global_name, storage_class)
```

Description

`coder.storageClass(global_name, storage_class)` assigns the storage class `storage_class` to the global variable `global_name`.

Assign the storage class to a global variable in a function that declares the global variable. You do not have to assign the storage class in more than one function.

You must have an Embedded Coder license to use `coder.storageClass`. Only when you use an Embedded Coder project or configuration object for generation of C/C++ libraries or executables does the code generation software recognize `coder.storageClass` calls.

Examples

Export Global Variables

In the function `addglobals_ex`, assign the `'ExportedGlobal'` storage class to the global variable `myglobalone` and the `'ExportedDefine'` storage class to the global variable `myglobaltwo`.

```
function y = addglobals_ex(x) %#codegen
% Define the global variables.
global myglobalone;
global myglobaltwo;

% Assign the storage classes.
coder.storageClass('myglobalone','ExportedGlobal');
coder.storageClass('myglobaltwo','ExportedDefine');
y = myglobalone + myglobaltwo + x;
end
```

Create a code configuration object for a library or executable.

```
cfg = coder.config('dll','ecoder', true);
```

Generate code. This example uses the `-globals` argument to specify the types and initial values of `myglobalone` and `myglobaltwo`. Alternatively, you can define global variables in the MATLAB global workspace. To specify the type of the input argument `x`, use the `-args` option.

```
codegen -config cfg -globals {'myglobalone', 1, 'myglobaltwo', 2} -args {1} addglobals_ex -report
```

From the initial values of `1` and `2`, `codegen` determines that `myglobalone` and `myglobaltwo` have the type `double`. `codegen` defines and declares the exported variables `myglobalone` and `myglobaltwo`. It generates code that initializes `myglobalone` to `1.0` and `myglobaltwo` to `2.0`.

To view the generated code for myglobaltwo and myglobalone, click the View report link.

- myglobaltwo is defined in the Exported data define section in addglobals_ex.h.

```
/* Exported data define */

/* Definition for custom storage class: ExportedDefine */
#define myglobaltwo                    2.0
```

- myglobalone is defined in the Variable Definitions section in addglobals_ex.c.

```
/* Variable Definitions */
/* Definition for custom storage class: ExportedGlobal */
double myglobalone;
```

- myglobalone is declared as extern in the Variable Declarations section in addglobals_ex.h.

```
/* Variable Declarations */
/* Declaration for custom storage class: ExportedGlobal */
extern double myglobalone;
```

- myglobalone is initialized in addglobals_ex_initialize.c.

```
/* Include Files */
#include "addglobals_ex_initialize.h"
#include "addglobals_ex.h"
#include "addglobals_ex_data.h"

/* Function Definitions */

/*
 * Arguments    : void
 * Return Type  : void
 */
void addglobals_ex_initialize(void)
{
  myglobalone = 1.0;
  isInitialized_addglobals_ex = true;
}
```

Import Global Variable

In the function addglobal_im, assign the 'ImportedExtern' storage class to the global variable myglobal.

```
function y = addglobal_im(x)

% Define the global variable.

global myglobal;

% Assign the storage classes.

coder.storageClass('myglobal','ImportedExtern');
y = myglobal + x;
end
```

Create a file c:\myfiles\myfile.c that defines and initializes the imported variable myglobal.

```
#include <stdio.h>

/* Variable definitions for imported variables */
double myglobal = 1.0;
```

Create a code configuration object. Configure the code generation parameters to include `myfile.c`. For output type `'lib'`, or if you generate source code only, you can generate code without providing this file. Otherwise, you must provide this file.

```
cfg = coder.config('dll','ecoder', true);
cfg.CustomSource = 'myfile.c';
cfg.CustomInclude = 'c:\myfiles';
```

Generate the code. This example uses the `-globals` argument to specify the type and initial value of `myglobal`. Alternatively, you can define global variables in the MATLAB global workspace. For imported global variables, the code generation software uses the initial values to determine only the type.

```
codegen -config cfg -globals {'myglobal', 1} -args {1}  addglobal_im -report
```

From the initial value `1`, `codegen` determines that `myglobal` has type `double`. `codegen` declares the imported global variable `myglobal`. It does not define `myglobal` or generate code that initializes `myglobal`. `myfile.c` provides the code that defines and initializes `myglobal`.

To view the generated code for `myglobal`, click the `View report` link.

`myglobal` is declared as `extern` in the `Variable Declarations` section in `addglobal_im_data.h`.

```
/* Variable Declarations */
/* Declaration for custom storage class: ImportedExtern */
extern double myglobal;
```

Import External Pointer

In the function `addglobal_imptr`, assign the `'ImportedExternPointer'` storage class to the global variable `myglobal`.

```
function y = addglobal_imptr(x)

% Define the global variable.

global myglobal;

% Assign the storage classes.

coder.storageClass('myglobal', 'ImportedExternPointer');
y = myglobal + x;
end
```

Create a file `c:\myfiles\myfile.c` that defines and initializes the imported global variable `myglobal`.

```
#include <stdio.h>
```

```
/* Variable definitions for imported variables */
double v = 1.0;
double *myglobal = &v;
```

Create a code configuration object. Configure the code generation parameters to include `myfile.c`. For output type `'lib'`, or if you generate source code only, you can generate code without providing this file. Otherwise, you must provide this file.

```
cfg = coder.config('dll','ecoder', true);
cfg.CustomSource = 'myfile.c';
cfg.CustomInclude = 'c:\myfiles';
```

Generate the code. This example uses the `-globals` argument to specify the type and initial value of the global variable `myglobal`. Alternatively, you can define global variables in the MATLAB global workspace. For imported global variables, the code generation software uses the initial values to determine only the type.

```
codegen -config cfg -globals {'myglobal', 1} -args {1} addglobal_imptr -report
```

From the initial value `1`, `codegen` determines that `myglobal` has type `double`. `codegen` declares the imported global variable `myglobal`. It does not define `myglobal` or generate code that initializes `myglobal`. `myfile.c` provides the code that defines and initializes `myglobal`.

To view the generated code for `myglobal`, click the `View report` link.

`myglobal` is declared as `extern` in the `Variable Declarations` section in `addglobal_imptr_data.h`.

```
/* Variable Declarations */
/* Declaration for custom storage class: ImportedExternPointer */
extern double *myglobal;
```

Input Arguments

`global_name` — Name of global variable

character vector

`global_name` is the name of a global variable, specified as a character vector. `global_name` must be a compile-time constant.

Example: `'myglobal'`

Data Types: `char`

`storage_class` — Name of storage class

`'ExportedGlobal'` | `'ExportedDefine'` | `'ImportedExtern'` | `'ImportedExternPointer'`

Storage class to assign to `global_var`. `storage_class` can have one of the following values.

Storage Class	Description
'ExportedGlobal'	• Defines the variable in the `Variable Definitions` section of the C file *entry_point_name*.c. • Declares the variable as an `extern` in the `Variable Declarations` section of the header file *entry_point_name*.h • Initializes the variable in the function *entry_point_name*_initialize.h.
'ExportedDefine'	Declares the variable with a `#define` directive in the `Exported data define` section of the header file *entry_point_name*.h.
'ImportedExtern'	Declares the variable as an `extern` in the `Variable Declarations` section of the header file *entry_point_name*_data.h. The external code must supply the variable definition.
'ImportedExternPointer'	Declares the variable as an `extern` pointer in the `Variable Declarations` section of the header file *entry_point_name*_data.h. The external code must define a valid pointer variable.

- If you do not assign a storage class to a global variable, except for the declaration location, the variable behaves like it has an `'ExportedGlobal'` storage class. For an `'ExportedGlobal'` storage class, the global variable is declared in the file *entry_point_name*.h. When the global variable does not have a storage class, the variable is declared in the file *entry_point_name*_data.h.

Data Types: `char`

Limitations

- After you assign a storage class to a global variable, you cannot assign a different storage class to that global variable.
- You cannot assign a storage class to a constant global variable.

See Also

`codegen`

Topics

"Control Declarations and Definitions of Global Variables in Code Generated from MATLAB Code"
"Storage Classes for Code Generation from MATLAB Code"

Introduced in R2015b

compare

Compare signal data

Syntax

```
[matchNames, matchFigures, mismatchNames, mismatchFigures] = cgv.CGV.compare(
data_set1, data_set2)
[matchNames, matchFigures, mismatchNames, mismatchFigures] = cgv.CGV.compare(
data_set1, data_set2, 'Plot', param_value)
[matchNames, matchFigures, mismatchNames, mismatchFigures] = cgv.CGV.compare(
data_set1, data_set2, 'Plot', 'none', 'Signals', signal_list,
'ToleranceFile', file_name)
```

Description

`[matchNames, matchFigures, mismatchNames, mismatchFigures] = cgv.CGV.compare(data_set1, data_set2)` compares data from two data sets which have common signal names between both executions. Possible outputs of the `cgv.CGV.compare` function are matched signal names, figure handles to the matched signal names, mismatched signal names, and figure handles to the mismatched signal names. By default, `cgv.CGV.compare` looks at the signals which have a common name between both executions.

`[matchNames, matchFigures, mismatchNames, mismatchFigures] = cgv.CGV.compare(data_set1, data_set2, 'Plot', param_value)` compares the signals and plots the signals according to `param_value`.

`[matchNames, matchFigures, mismatchNames, mismatchFigures] = cgv.CGV.compare(data_set1, data_set2, 'Plot', 'none', 'Signals', signal_list, 'ToleranceFile', file_name)` compares only the given signals and does not produce plots.

Input Arguments

`data_set1, data_set2`

Output data from a model. After running the model, use the `getOutputData` function to get the data. The `cgv.CGV.getOutputData` function returns a cell array of the output signal names.

`varargin`

Variable number of parameter name and value pairs.

varargin Parameters

You can specify the following argument properties for the `cgv.CGV.compare` function using parameter name and value argument pairs. These parameters are optional.

`Plot`(optional)

Designates which comparison data to plot. The value of this parameter must be one of the following:

- `'match'`: plot the comparison of the matched signals from the two data sets
- `'mismatch'`(default): plot the comparison of the mismatched signals from the two datasets
- `'none'`: do not produce a plot

`Signals`(optional)

A cell array of character vectors, where each vector is a signal name in the output data. Use `getSavedSignals` to view the list of available signal names in the output data. `signal_list` can contain an individual signal or multiple signals. The syntax for an individual signal name is:

```
signal_list = {'log_data.subsystem_name.Data(:,1)'}
```

The syntax for multiple signal names is:

```
signal_list = {'log_data.block_name.Data(:,1)',...
                                    'log_data.block_name.Data(:,2)',...
                      'log_data.block_name.Data(:,3)',...
                      'log_data.block_name.Data(:,4)'};
```

If a model component contains a space or newline character, MATLAB adds parentheses and a single quote to the name of the component. For example, if a section of the signal has a space, `'block name'`, MATLAB displays the signal name as:

```
log_data.('block name').Data(:,1)
```

To use the signal name as input to a CGV function, `'block name'` must have two single quotes. For example:

```
signal_list = {'log_data.(''block name'').Data(:,1)'}
```

If `Signals` is not present, the signals are compared.

`Tolerancefile`(optional)

Name for the file created by the `createToleranceFile` function. The file contains the signal names and the associated tolerance parameter name and value pair for comparing the data.

Output Arguments

Depending on the data and the parameters, the following output arguments might be empty.

`match_names`

Cell array of matching signal names.

`match_figures`

Array of figure handles for matching signals

`mismatch_names`

Cell array of mismatching signal names

`mismatch_figures`

Array of figure handles for mismatching signals

See Also

Topics
“Verify Numerical Equivalence with CGV”

configModel

Determine and change configuration parameter values

Syntax

`cfgObj.configModel()`

Description

`cfgObj.configModel()` determines the recommended values for the configuration parameters in the model. *cfgObj* is a handle to a `cgv.Config` object. The `ReportOnly` property of the object determines whether `configModel` changes the configuration parameter values.

See Also

Topics
"Manage Configuration Sets for a Model"
"Programmatic Code Generation Verification"

coder.MATLABCodeTemplate class

Package: `coder`

Represent code generation template for MATLAB Coder

Description

Create a coder.MATLABCodeTemplate object from a code generation template (CGT) file. You can use this file to customize the code generation output for MATLAB Coder™. If a CGT file is not provided, the coder.MATLABCodeTemplate object is created from the default template file *`matlabroot`*`/toolbox/coder/matlabcoder/templates/matlabcoder_default_template.cgt`.

Construction

`newObj = coder.MATLABCodeTemplate()` creates a `coder.MATLABCodeTemplate` object from the default code generation template (CGT) file *`matlabroot`*`/toolbox/coder/matlabcoder/templates/matlabcoder_default_template.cgt`.

`newObj = coder.MATLABCodeTemplate(CGTFile)` creates a `coder.MATLABCodeTemplate` object from the code generation template file `CGTFile`. If the file is not on the MATLAB path, specify a full path to the file.

Input Arguments

`CGTFile`

Name of code generation template file

Methods

emitSection	Emit comments for template section
getCurrentTokens	Get current tokens
getTokenValue	Get value of token
setTokenValue	Set value of token for code generation template

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

```
newObj = coder.MATLABCodeTemplate()
```

```
newObj =

  MATLABCodeTemplate with properties:

    CGTFile: 'matlabcoder_default_template.cgt'
```

```
newObj = coder.MATLABCodeTemplate('custom_matlabcoder_template.cgt')
```

```
newObj =

  MATLABCodeTemplate with properties:

    CGTFile: 'custom_matlabcoder_template.cgt'
```

See Also

coder.MATLABCodeTemplate.emitSection | coder.MATLABCodeTemplate.getCurrentTokens | coder.MATLABCodeTemplate.getTokenValue | coder.MATLABCodeTemplate.setTokenValue

Topics

"Generate Custom File and Function Banners for C/C++ Code"
"Code Generation Template Files for MATLAB Code"

copySetup

Create copy of `cgv.CGV` object

Syntax

`cgvObj2 = cgvObj1.copySetup()`

Description

`cgvObj2 = cgvObj1.copySetup()` creates a copy of a `cgv.CGV` on page 1-44 object, *`cgvObj1`*. The copied object, `cgvObj2`, has the same configuration as *`cgvObj1`*, but does not copy results of the execution.

Examples

Make a copy of a `cgv.CGV` object, set it to run in a different mode, then run and compare the objects in a `cgv.Batch` object.

```
cgvModel = 'rtwdemo_cgv';
cgvObj1 = cgv.CGV(cgvModel, 'connectivity', 'sim');
cgvObj1.run();
cgvObj2 = cgvObj1.copySetup()
cgvObj2.setMode('sil');
cgvObj2.run();
```

Tips

- You can use this method to make a copy of a `cgv.CGV` object and then modify the object to run in a different mode by calling `setMode`.
- If you have a `cgv.CGV` object, which reported errors or failed at execution, you can use this method to copy the object and rerun it. The copied object has the same configuration as the original object, therefore you might want to modify the location of the output files by calling `setOutputDir`. Otherwise, during execution, the copied `cgv.CGV` object overwrites the output files.

See Also

`run`

Topics

"Verify Numerical Equivalence with CGV"

copyConceptualArgsToImplementation

Copy conceptual argument specifications to implementation specifications of an entry for code replacement table entry

Syntax

`copyConceptualArgsToImplementation(hEntry)`

Description

`copyConceptualArgsToImplementation(hEntry)` provides a quick way to perform a shallow copy of conceptual arguments to matching implementation arguments.

The conceptual arguments and implementation arguments refer to the same argument instance. If you update an implementation argument, the corresponding conceptual argument is also updated.

Use this function when the conceptual arguments and the implementation arguments are the same for a code replacement table entry.

For arguments with an unsized type, such as `integer`, the code generator determines the size of the argument values based on hardware implementation configuration settings of the MATLAB code or model.

Examples

Copy Conceptual Argument to Implementation Arguments

This example shows how to use the `copyConceptualArgsToImplementation` function to copy conceptual argument specifications to matching implementation arguments for an addition operation.

```
hLib = RTW.TflTable;

% Create an entry for addition of built-in uint8 data type
op_entry = RTW.TflCOperationEntry;
op_entry.setTflCOperationEntryParameters( ...
    'Key',                      'RTW_OP_ADD', ...
    'Priority',                 90, ...
    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes',            {'RTW_ROUND_UNSPECIFIED'}, ...
    'ImplementationName',       'u8_add_u8_u8', ...
    'ImplementationHeaderFile', 'u8_add_u8_u8.h', ...
    'ImplementationSourceFile', 'u8_add_u8_u8.c' );

arg = hLib.getTflArgFromString('y1','uint8');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.addConceptualArg(arg);

arg = hLib.getTflArgFromString('u1','uint8');
op_entry.addConceptualArg(arg);
```

```
arg = hLib.getTflArgFromString('u2','uint8');
op_entry.addConceptualArg(arg);

op_entry.copyConceptualArgsToImplementation();

addEntry(hLib, op_entry);
```

Input Arguments

`hEntry` — Handle to a code replacement table entry

handle

The *`hEntry`* is a handle to a code replacement table entry previously returned by instantiating a code replacement table entry class, such as *`hEntry`* `= RTW.TflCFunctionEntry` or *`hEntry`* `= RTW.TflCOperationEntry`.

Example: `op_entry`

See Also

Topics

"Define Code Replacement Library Optimizations"
"Code You Can Replace from MATLAB Code"
"Code You Can Replace From Simulink Models"

Introduced in R2007b

createAndAddConceptualArg

Create conceptual argument from specified properties and add to conceptual arguments for code replacement table entry

Syntax

`arg = createAndAddConceptualArg(hEntry,argType, varargin)`

Description

`arg = createAndAddConceptualArg(hEntry,argType, varargin)` creates a conceptual argument from specified properties and adds the argument to the conceptual arguments for a code replacement table entry.

Examples

Specify Conceptual Output and Input Arguments

This example shows how to use the`createAndAddConceptualArg` function to specify conceptual output and input arguments for a code replacement operator entry.

For examples of fixed-point arguments that use relative scaling or relative slope/bias values, see “Net Slope Scaling Code Replacement” and “Equal Slope and Zero Net Bias Code Replacement”.

```
op_entry = RTW.TflCOperationEntry;
.
.
.
createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
    'Name',       'y1', ...
    'IOType',     'RTW_IO_OUTPUT', ...
    'IsSigned',   true, ...
    'WordLength', 32, ...
    'FractionLength', 0);

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric',...
    'Name',       'u1', ...
    'IOType',     'RTW_IO_INPUT',...
    'IsSigned',   true,...
    'WordLength', 32, ...
    'FractionLength', 0 );

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric',...
    'Name',       'u2', ...
    'IOType',     'RTW_IO_INPUT',...
    'IsSigned',   true,...
    'WordLength', 32, ...
    'FractionLength', 0 );
```

Specify Types for Conceptual Argument

These examples show some common type specifications using `createAndAddConceptualArg`.

```
hEntry = RTW.TflCOperationEntry;
.
.
.
% uint8:
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',           'u1', ...
    'IOType',         'RTW_IO_INPUT', ...
    'IsSigned',       false, ...
    'WordLength',     8, ...
    'FractionLength', 0 );

% single:
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',         'u1', ...
    'IOType',       'RTW_IO_INPUT', ...
    'DataTypeMode', 'single' );

% double:
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',         'y1', ...
    'IOType',       'RTW_IO_OUTPUT', ...
    'DataTypeMode', 'double' );

% boolean:
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',         'u1', ...
    'IOType',       'RTW_IO_INPUT', ...
    'DataTypeMode', 'boolean' );

% Fixed-point using binary-point-only scaling:
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',           'y1', ...
    'IOType',         'RTW_IO_OUTPUT', ...
    'CheckSlope',     true, ...
    'CheckBias',      true, ...
    'DataTypeMode',   'Fixed-point: binary point scaling', ...
    'IsSigned',       true, ...
    'WordLength',     32, ...
    'FractionLength', 28);

% Fixed-point using [slope bias] scaling:
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',           'y1', ...
    'IOType',         'RTW_IO_OUTPUT', ...
    'CheckSlope',     true, ...
    'CheckBias',      true, ...
    'DataTypeMode',   'Fixed-point: slope and bias scaling', ...
    'IsSigned',       true, ...
    'WordLength',     16, ...
```

```
    'Slope',          15, ...
    'Bias',           2);
```

Input Arguments

`hEntry` — Handle to a code replacement table entry
handle

The *`hEntry`* is a handle to a code replacement table entry previously returned by instantiating a code replacement table entry class, such as *`hEntry`* `= RTW.TflCFunctionEntry` or *`hEntry`* `= RTW.TflCOperationEntry`.

Example: `op_entry`

`argType` — Specifies the argument type to create
`'RTW.TflArgNumeric'` | `'RTW.TflArgMatrix'`

The *`argType`* is a character vector or string scalar that specifies the argument type to create. Use `'RTW.TflArgNumeric'` for numeric or `'RTW.TflArgMatrix'` for matrix.

Example: `'RTW.TflArgNumeric'`

`varargin` — Name-value pair arguments that specify the conceptual argument
name-value pair

Example: `'Name','y1'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Name','y1'`

`Name` — Specifies the argument name
character vector | string scalar

Example: `'Name','y1'`

`IOType` — Specifies the I/O type of the argument
`'RTW_IO_INPUT'` (default) | `'RTW_IO_OUTPUT'`

Use value `'RTW_IO_INPUT'` for input or value `'RTW_IO_OUTPUT'`.

Example: `'IOType','RTW_IO_INPUT'`

`IsSigned` — Indicates whether the argument is signed
`true` (default) | `false`

Boolean value that, when set to `true`, indicates that the argument is signed.

Example: `'IsSigned',true`

`WordLength` — Specifies the word length, in bits, of the argument
`16` (default) | integer

Integer specifying the word length, in bits, of the argument. The default is `16`.

Example: 'WordLength',16

CheckSlope — Selects whether to check that the slope value of the argument exactly matches the call-site slope value
true (default) | false

Boolean flag that, when set to true for a fixed-point argument, causes code replacement request processing to check that the slope value of the argument exactly matches the call-site slope value.

Specify true if you are matching a specific [slope bias] scaling combination or a specific binary-point-only scaling combination on fixed-point operator inputs and output. Specify false if you are matching relative scaling or relative slope and bias values across fixed-point operator inputs and output.

Example: 'CheckSlope',true

CheckBias — Selects whether to check that the bias value of the argument exactly matches the call-site bias value
true (default) | false

Boolean flag that, when set to true for a fixed-point argument, causes code replacement request processing to check that the bias value of the argument exactly matches the call-site bias value.

Specify true if you are matching a specific [slope bias] scaling combination or a specific binary-point-only scaling combination on fixed-point operator inputs and output. Specify false if you are matching relative scaling or relative slope and bias values across fixed-point operator inputs and output.

Example: 'CheckBias',true

DataTypeMode — Specifies the data type mode of the argument
'Fixed-point: binary point scaling' (default) | 'Fixed-point: slope and bias scaling' | 'boolean' | 'double' | 'single'

You can specify either DataType (with Scaling) or DataTypeMode, but do not specify both.

Example: 'DataTypeMode','Fixed-point: binary point scaling'

DataType — Specifies the data type of the argument
'Fixed' (default) | 'boolean' | 'double' | 'single'

Example: 'DataType','Fixed'

Scaling — Specifies the data type scaling of the argument
'BinaryPoint' (default) | 'SlopeBias'

Specify the data type scaling of the argument as 'BinaryPoint' for binary-point scaling or 'SlopeBias' for slope and bias scaling.

Example: 'Scaling','BinaryPoint'

Slope — Specifies the slope of the argument
1 (default) | floating-point value

If you are matching a specific [slope bias] scaling combination on fixed-point operator inputs and output, specify either this parameter or a combination of the SlopeAdjustmentFactor and FixedExponent parameters.

Example: 'Slope',1.0

SlopeAdjustmentFactor — Specifies the slope adjustment factor (F) part of the slope, $F2^E$, of the argument
1.0 (default) | floating-point value

If you are matching a specific [slope bias] scaling combination on fixed-point operator inputs and output, specify either the Slope parameter or a combination of this parameter and the FixedExponent parameter.

Example: 'SlopeAdjustmentFactor',1.0

FixedExponent — Specifies the fixed exponent (E) part of the slope, $F2^E$, of the argument
-15 (default) | integer value

If you are matching a specific [slope bias] scaling combination on fixed-point operator inputs and output, specify either the Slope parameter or a combination of this parameter and the SlopeAdjustmentFactor parameter.

Example: 'FixedExponent',-15

Bias — Specifies the bias of the argument
0.0 (default) | floating-point value

Specify this parameter if you are matching a specific [slope bias] scaling combination on fixed-point operator inputs and output.

Example: 'Bias',2.0

FractionLength — Specifies the fraction length for the argument
15 (default) | integer value

Specify this parameter if you are matching a specific binary-point-only scaling combination on fixed-point operator inputs and output.

Example: 'FractionLength',15

BaseType — Specifies the base data type for which a matrix argument is valid
character vector | string scalar

Example: 'BaseType','double'

DimRange — Specifies the dimensions for which a matrix argument is valid
matrix dimensions

You can also specify a range of dimensions specified in the format [Dim1Min Dim2Min ... DimNMin; Dim1Max Dim2Max ... DimNMax]. For example, [2 2; inf inf] means a two-dimensional matrix of size 2x2 or larger.

Example: 'DimRange',[2 2]

Output Arguments

arg — Handle to the created conceptual argument
handle

The *arg* is a handle to the created conceptual argument. Specifying the return argument in the `createAndAddConceptualArg` function call is optional.

See Also

Topics
"Define Code Replacement Library Optimizations"
"Code You Can Replace from MATLAB Code"
"Code You Can Replace From Simulink Models"

Introduced in R2007b

createAndAddImplementationArg

Create implementation argument from specified properties and add to implementation arguments for code replacement table entry

Syntax

`arg = createAndAddImplementationArg(hEntry,argType,varargin)`

Description

`arg = createAndAddImplementationArg(hEntry,argType,varargin)` creates an implementation argument from specified properties and adds the argument to the implementation arguments for a code replacement table entry.

Implementation arguments must describe fundamental numeric data types, such as `double`, `single`, `int32`, `int16`, `int8`, `uint32`, `uint16`, `uint8`, `boolean`, or `'logical'` (not fixed-point data types).

Examples

Specify Implementation Output and Input Arguments

This example shows how to use the`createAndAddImplementationArg` function with the `createAndSetCImplementationReturn` function to specify the output and input arguments for an operator implementation.

```
op_entry = RTW.TflCOperationEntry;
.
.
.
createAndSetCImplementationReturn(op_entry, 'RTW.TflArgNumeric', ...
    'Name',       'y1', ...
    'IOType',     'RTW_IO_OUTPUT', ...
    'IsSigned',   true, ...
    'WordLength', 32, ...
    'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
    'Name',       'u1', ...
    'IOType',     'RTW_IO_INPUT',...
    'IsSigned',   true,...
    'WordLength', 32, ...
    'FractionLength', 0 );

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
    'Name',       'u2', ...
    'IOType',     'RTW_IO_INPUT',...
    'IsSigned',   true,...
    'WordLength', 32, ...
    'FractionLength', 0 );
```

Specify Types for Implementation Argument

These examples show some common type specifications using `createAndAddImplementationArg`.

```
hEntry = RTW.TflCOperationEntry;
.
.
.
% uint8:
createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',           'u1', ...
    'IOType',         'RTW_IO_INPUT', ...
    'IsSigned',       false, ...
    'WordLength',     8, ...
    'FractionLength', 0 );

% single:
createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',         'u1', ...
    'IOType',       'RTW_IO_INPUT', ...
    'DataTypeMode', 'single' );

% double:
createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',         'u1', ...
    'IOType',       'RTW_IO_INPUT', ...
    'DataTypeMode', 'double' );

% boolean:
createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',         'u1', ...
    'IOType',       'RTW_IO_INPUT', ...
    'DataTypeMode', 'boolean' );
```

Input Arguments

`hEntry` — Handle to a code replacement table entry
handle

The *`hEntry`* is a handle to a code replacement table entry previously returned by instantiating a code replacement entry class, such as *`hEntry`* `= RTW.TflCFunctionEntry` or *`hEntry`* `= RTW.TflCOperationEntry`.

Example: `op_entry`

`argType` — Specifies the argument type to create
`'RTW.TflArgNumeric'` | character vector | string scalar

The *`argType`* is a character vector or string scalar that specifies the argument type to create. Use `'RTW.TflArgNumeric'` for numeric.

Example: `'RTW.TflArgNumeric'`

`varargin` — Name-value pairs that specify the implementation argument
name-value pairs

Example: `'Name','u1'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Name','u1'`

`Name` — Specifies the argument name
character vector | string scalar

Example: `'Name','u1'`

`IOType` — Specifies the I/O type of the argument
`'RTW_IO_INPUT'` | character vector | string scalar

Use `'RTW_IO_INPUT'` for input.

Example: `'IOType','RTW_IO_INPUT'`

`IsSigned` — Indicates whether the argument is signed
`true` (default) | `false`

Boolean value that, when set to `true`, indicates that the argument is signed.

Example: `'IsSigned',true`

`WordLength` — Specifies the word length, in bits, of the argument
`16` (default) | integer value

Example: `'WordLength',16`

`DataTypeMode` — Specifies the data type mode of the argument
`'Fixed-point: binary point scaling'` (default) | `'Fixed-point: slope and bias scaling'` | `'boolean'` | `'double'` | `'single'`

You can specify either `DataType` (with `Scaling`) or `DataTypeMode`, but do not specify both.

Example: `'DataTypeMode','Fixed-point: binary point scaling'`

`DataType` — Specifies the data type of the argument
`'Fixed'` (default) | `'boolean'` | `'double'` | `'single'`

Example: `'DataType','Fixed'`

`Scaling` — Specifies the data type scaling of the argument
`'BinaryPoint'` (default) | `'SlopeBias'`

Use `'BinaryPoint'` for binary-point scaling or `'SlopeBias'` for slope and bias scaling.

Example: `'Scaling','BinaryPoint'`

`Slope` — Specifies the slope of the argument
`1.0` (default) | floating-point value

You can optionally specify either this parameter or a combination of the `SlopeAdjustmentFactor` and `FixedExponent` parameters, but do not specify both.

Example: `'Slope',1.0`

`SlopeAdjustmentFactor` — Specifies the slope adjustment factor (F) part of the slope, $F2^E$, of the argument
`1.0` (default) | floating-point value

You can optionally specify either the `Slope` parameter or a combination of this parameter and the `FixedExponent` parameter, but do not specify both.

Example: `'SlopeAdjustmentFactor',1.0`

`FixedExponent` — Specifies the fixed exponent (E) part of the slope, $F2^E$, of the argument
`-15` (default) | integer value

You can optionally specify either the `Slope` parameter or a combination of this parameter and the `SlopeAdjustmentFactor` parameter, but do not specify both.

Example: `'FixedExponent',0`

`Bias` — Specifies the bias of the argument
`0.0` (default) | floating-point value

Example: `'Bias',0.0`

`FractionLength` — Specifies the fraction length of the argument
`15` (default) | integer value

Example: `'FractionLength',0`

`Value` — Specifies the initial value of the argument
`0` (default) | constant value

Use this parameter only to set the value of injected constant input arguments, such as arguments that pass fraction-length values or flag values, in an implementation function signature. Do not use it for standard generated input arguments, such as `u1u2`. You can supply a constant input argument that uses this parameter anywhere in the implementation function signature, except as the return argument.

You can inject constant input arguments into the implementation signature for code replacement table entries. If the argument values or the number of arguments required depends on compile-time information, you can use custom matching. For more information, see “Customize Match and Replacement Process”.

Example: `'Value',0`

Output Arguments

`arg` — Handle to the created implementation argument
handle

Specifying the return argument in the `createAndAddImplementationArg` function call is optional.

See Also

`createAndSetCImplementationReturn`

Topics
“Define Code Replacement Library Optimizations”

"Code You Can Replace from MATLAB Code"
"Code You Can Replace From Simulink Models"

Introduced in R2007b

createAndSetCImplementationReturn

Create implementation return argument from specified properties and add to implementation for code replacement table entry

Syntax

```
arg = createAndSetCImplementationReturn(hEntry,argType,varargin)
```

Description

`arg = createAndSetCImplementationReturn(hEntry,argType,varargin)` creates an implementation return argument from specified properties and adds the argument to the implementation for a code replacement table.

Implementation return arguments must describe fundamental numeric data types, such as `double`, `single`, `int32`, `int16`, `int8`, `uint32`, `uint16`, `uint8`, or `boolean` (not fixed-point data types).

Examples

Specify Operator Output and Input Arguments

This example shows how to use the `createAndSetCImplementationReturn` function with the `createAndAddImplementationArg` function to specify the output and input arguments for an operator implementation.

```
op_entry = RTW.TflCOperationEntry;
.
.
.
createAndSetCImplementationReturn(op_entry, 'RTW.TflArgNumeric', ...
    'Name',       'y1', ...
    'IOType',     'RTW_IO_OUTPUT', ...
    'IsSigned',   true, ...
    'WordLength', 32, ...
    'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
    'Name',       'u1', ...
    'IOType',     'RTW_IO_INPUT',...
    'IsSigned',   true,...
    'WordLength', 32, ...
    'FractionLength', 0 );

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
    'Name',       'u2', ...
    'IOType',     'RTW_IO_INPUT',...
    'IsSigned',   true,...
    'WordLength', 32, ...
    'FractionLength', 0 );
```

Specify Types for Operator Implementation

These examples show some common type specifications using `createAndSetCImplementationReturn`.

```
hEntry = RTW.TflCOperationEntry;
.
.
.
% uint8:
createAndSetCImplementationReturn(hEntry, 'RTW.TflArgNumeric', ...
    'Name',           'y1', ...
    'IOType',         'RTW_IO_OUTPUT', ...
    'IsSigned',       false, ...
    'WordLength',     8, ...
    'FractionLength', 0 );

% single:
createAndSetCImplementationReturn(hEntry, 'RTW.TflArgNumeric', ...
    'Name',         'y1', ...
    'IOType',       'RTW_IO_OUTPUT', ...
    'DataTypeMode', 'single' );

% double:
createAndSetCImplementationReturn(hEntry, 'RTW.TflArgNumeric', ...
    'Name',         'y1', ...
    'IOType',       'RTW_IO_OUTPUT', ...
    'DataTypeMode', 'double' );

% boolean:
createAndSetCImplementationReturn(hEntry, 'RTW.TflArgNumeric', ...
    'Name',         'y1', ...
    'IOType',       'RTW_IO_OUTPUT', ...
    'DataTypeMode', 'boolean' );
```

Input Arguments

`hEntry` — Handle to a code replacement table entry
handle

The *`hEntry`* is a handle to a code replacement table entry previously returned by instantiating a code replacement entry class, such as *`hEntry`* `= RTW.TflCFunctionEntry` or *`hEntry`* `= RTW.TflCOperationEntry`.

Example: `op_entry`

`argType` — Specifies the argument type to create
`'RTW.TflArgNumeric'` | character vector | string scalar

The *`argType`* is a character vector or string scalar that specifies the argument type to create. Use `'RTW.TflArgNumeric'` for numeric.

Example: `'RTW.TflArgNumeric'`

`varargin` — Name-value pairs that specify the implementation return argument
name-value pairs

Example: `'Name','y1'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Name','y1'`

`Name` — Specifies the argument name
character vector | string scalar

Example: `'Name','y1'`

`IOType` — Specifies the I/O type of the argument
`'RTW_IO_OUTPUT'` | character vector | string scalar

Use `'RTW_IO_OUTPUT'` for output.

Example: `'IOType','RTW_IO_OUTPUT'`

`IsSigned` — Indicates whether the argument is signed
`true` (default) | `false`

Boolean value that, when set to `true`, indicates that the argument is signed. The default is `true`.

Example: `'IsSigned',true`

`WordLength` — Specifies the word length, in bits, of the argument
`16` (default) | integer

Example: `'WordLength',16`

`DataTypeMode` — Specifies the data type mode of the argument
`'Fixed-point: binary point scaling'` (default) | `'Fixed-point: slope and bias scaling'` | `'boolean'` | `'double'` | `'single'`

You can specify either `DataType` (with `Scaling`) or `DataTypeMode`, but do not specify both.

Example: `'DataTypeMode','Fixed-point: binary point scaling'`

`DataType` — Specifies the data type of the argument
`'Fixed'` (default) | `'boolean'` | `'double'` | `'single'`

Example: `'DataType','Fixed'`

`Scaling` — Specifies the data type scaling of the argument
`'BinaryPoint'` (default) | `'SlopeBias'`

Use `'BinaryPoint'` for binary-point scaling or `'SlopeBias'` for slope and bias scaling.

Example: `'Scaling','BinaryPoint'`

`Slope` — Specifies the slope for a fixed-point argument
`1.0` (default) | floating-point value

You can optionally specify either this parameter or a combination of the `SlopeAdjustmentFactor` and `FixedExponent` parameters, but do not specify both.

Example: `'Slope',1.0`

SlopeAdjustmentFactor — Specifies the slope adjustment factor (F) part of the slope, $F2^E$, of the argument
1.0 (default) | floating-point value

You can optionally specify either the Slope parameter or a combination of this parameter and the FixedExponent parameter, but do not specify both.

Example: 'SlopeAdjustmentFactor',1.0

FixedExponent — Specifies the fixed exponent (E) part of the slope, $F2^E$, of the argument
-15 (default) | integer value

You can optionally specify either the Slope parameter or a combination of this parameter and the SlopeAdjustmentFactor parameter, but do not specify both.

Example: 'FixedExponent',0

Bias — Specifies the bias of the argument
0.0 (default) | floating-point value

Example: 'Bias',0.0

FractionLength — Specifies the fraction length of the argument
15 (default) | integer value

Example: 'FractionLength',0

Output Arguments

arg — Handle to the created implementation return argument
handle

Specifying the return argument in the createAndSetCImplementationReturn function call is optional.

See Also

createAndAddImplementationArg

Topics

"Define Code Replacement Library Optimizations"
"Code You Can Replace from MATLAB Code"
"Code You Can Replace From Simulink Models"

Introduced in R2007b

createCRLEntry

Create code replacement table entry from conceptual and implementation argument string specifications

Syntax

```
tableEntry = createCRLEntry(crTable,conceptualSpecification,
implementationSpecification)
```

Description

`tableEntry = createCRLEntry(crTable,conceptualSpecification, implementationSpecification)` returns a code replacement table entry. The entry maps a conceptual representation of a function or operator to an implementation representation. The `conceptualSpecification` argument is a character vector or string scalar that defines the name and conceptual arguments, familiar to the code generator, for the function or operator to replace. The `implementationSpecification` argument is a character vector or string scalar that defines the name and C/C++ implementation arguments for the replacement function.

This function does not support:

- C++ implementations
- Data alignment
- Operator replacement with net slope arguments
- Entry parameter specifications (for example, priority, algorithm, building information)
- Semaphore and mutex function replacements

In the syntax specifications, place a space before and after an operator symbol. For example, use `double u1 + double u2` instead of `double u1+double u2`. Also, asterisk (*), tilde (~), and semicolon (;) have the following meaning.

Symbol	Meaning
*	• Following a supported data type, such as `int32*`, pass by reference (pointer). If the conceptual arguments are not scalar, in the implementation specification, pass them by reference. • As part of a fixed-point data type definition, such as `fixdt(1,32,*)`, wildcard.
~	Based on the position of the symbol, slopes or bias must be the same across data types.
;	Separates dimension ranges. For example, `[1 10; 1 100]` specifies a vector with length from 10 through 100.

The following table shows syntax for the conceptual and implementation specifications based on:

- Whether you are creating an entry for a function or operator.
- The type or characterization of the code replacement.

Type of Replacement	Conceptual Syntax	Implementation Syntax
Function Code Replacement Syntax		
Typical	`double y1 = sin(double u1)`	`double y1 = mySin(double u1)`
Derive implementation argument data types from conceptual specification	`double y1 = sin(double u1)`	`y1 = mySin(u1)`
Derive implementation arguments and data types from conceptual specification	`double y1 = sin(double u1)`	`mySin`
Change data type	`single y1 = sin(single u1)`	`double y1 = mySin(double u1)`
Reorder arguments	`double y1 = atan2(double u1, double u2)`	`y1 = myAtan(u2, u1)`
Specify column vector arguments	`double y1 = sin(double u1[10])`	`double y1 = mySin(double* u1)`
Specify column vector arguments and dimension range	`double y1[1 100; 1 100] = sin(double u1[1 100; 1 100])`	`mySin(double* u1, double* y1)`
Remap return value as output argument	`double y1 = sin(double u1)`	`mySin(double u1, double* y1)`
Specify fixed-point data types	`fixdt(1,16,3) y1 = sin(fixdt(1,16,3) u1)`	`int16 y1 = mySin(int16 u1)`
Specify fixed-point data types and set `CheckSlope` to `false`, `CheckBias` to `true`, and `Bias` to 0	`fixdt(1,16,*) y1 = sin(fixdt(1,16,*) u1)`	`int16 y1 = mySin(int16 u1)`
Specify fixed-point data types and set `SlopesMustBeTheSame` to `true`, `CheckSlope` to `false`, `CheckBias` to `true`, and `Bias` to 0	`fixdt(1,16,~) y1 = sin(fixdt(1,16,~) u1)`	`int16 y1 = mySin(int16 u1)`

Type of Replacement	Conceptual Syntax	Implementation Syntax
Specify fixed-point data types and set SlopesMustBeTheSame to true, BiasMustBeTheSame to true, CheckSlope to false, and CheckBias to false	fixdt(1,16,~,~) y1 = sin(fixdt(1,16,~,~) u1)	int16 y1 = mySin(int16 u1)
Specify multiple output arguments	[double y1 double y2] = foo(double u1, double u2)	double y1 = myFoo(double u1, double u2, double* y2)
Operator Code Replacement Syntax		
Typical	int16 y1 = int16 u1 + int16 u2	int16 y1 = myAdd(int16 u1, int16 u2)
Specify fixed-point data types	fixdt(1,16,3) y1 = fixdt(1,16,3) u1 + fixdt(1,16,3) u2	int16 y1 = myAdd(int16 u1, int16 u2)
Specify fixed-point data types and set CheckSlope to false, CheckBias to true, and Bias to 0	fixdt(1,16,*) y1 = fixdt(1,16,*) u1 + fixdt(1,16,*) u2	int16 y1 = myAdd(int16 u1, int16 u2)
Specify fixed-point data types, wildcard, slopes must be the same, and zero bias	fixdt(1,16,~,0) y1 = fixdt(1,16,~,0) u1 + fixdt(1,16,~,0) u2	int16 y1 = myAdd(int16 u1, int16 u2)
Typecast	int16 y1 = int8 u1	int16 y1 = myCast(int8 u1)
Shift	int16 y1 = int16 u1 << int16 u2 int16 y1 = int16 u1 >> int16 u2 int16 y1 = int16 u1 .>> int16 u2	int16 y1 = myShiftLeft(int16 u1, int16 u2) int16 y1 = myShiftRightArithmetic(int16 u1, int16 u2) int16 y1 = myShiftRightLogical(int16 u1, int16 u2)
Specify relational operator	bool y1 = int16 u1 < int16 u2	bool y1 = myLessThan(int6 u1, int16 u2)
Specify multiplication and division	int32 y1 = int32 u1 * in32 u2 / in32 u3	int32 y1 = myMultDiv(int32 u1, int32 u2, int32 u3)
Specify matrix multiplication	double y1[10][10] = double u1[10][10] * double u2[10][10]	myMult(double* u1, double* u2, double* y1)

Type of Replacement	Conceptual Syntax	Implementation Syntax
Specify element-wise matrix multiplication	`double y1[10][10] = double u1[10][10] .* double u2[10][10]`	`myMult(double* u1, double* u2, double* y1)`
Specify matrix multiplication with transpose of an input argument	`double y1[10][10] = double u1[10][10]' * double u2[10][10]`	`myMult(double* u1, double* u2, double* y1)`
Specify matrix multiplication with Hermitian of an input argument	`cdouble y1[10][10] = cdouble u1[10][10]' * cdouble u2[10][10]` `cdouble y1[10][10] = cdouble u1[10][10] * cdouble u2[10][10]'`	`myMult(cdouble* u1, cdouble* u2, cdouble* y1)`
Specify left matrix division	`double y1[10][10] = double u1[10][10] \ double u2[10][10]`	`myLeftDiv(double* u1, double* u2, double* y1)`
Specify right matrix division	`double y1[10][10] = double u1[10][10] / double u2[10][10]`	`myRightDiv(double* u1, double* u2, double* y1)`

Examples

Replacement Entry for a Function

Create a table definition file that contains a function definition.

```
function crTable = crl_table_sinfcn()
```

Within the function body, create the code replacement table.

```
crTable = RTW.TflTable;
```

Create a table entry for the `sin` function.

```
tableEntry = createCRLEntry(crTable, ...
    'double y1 = sin(double u1)', ...
    'double y1 = mySin(double u1)');
```

Set entry parameters for the `sin` function. To generate the replacement code, specify that the code generator use the header and source files `mySin.h` and `mySin.c`.

```
setTflCFunctionEntryParameters(tableEntry, ...
    'ImplementationHeaderFile', 'mySin.h', ...
    'ImplementationSourceFile', 'mySin.c');
```

Add the entry to the table.

```
addEntry(crTable, tableEntry);
```

Replacement Entry for an Operator

Create a table definition file that contains a function definition.

```
function crTable = crl_table_addfcn()
```

Within the function body, create the code replacement table.

```
crTable = RTW.TflTable;
```

Create a table entry for the addition operator.

```
tableEntry = createCRLEntry(crTable, ...
    'int16 y1 = int16 u1 + int16 u2', ...
    'int16 y1 = myAdd(int16 u1, int16 u2)');
```

Set entry parameters such that the entry specifies a cast-after-sum addition. To generate the replacement code, specify that the code generator use the header and source files `myAdd.h` and `myAdd.c`.

```
setTflCOperationEntryParameters(tableEntry, ...
    'EntryInfoAlgorithm', 'RTW_CAST_AFTER_OP', ...
    'ImplementationHeaderFile', 'myAdd.h', ...
    'ImplementationSourceFile', 'myAdd.c'));
```

Add the entry to the table.

```
addEntry(crTable, tableEntry);
```

Replacement Entry for Fixed-Point Operator With Same Slope Across Types

Create a table definition file that contains a function definition.

```
function crTable = crl_table_intaddfcn()
```

Within the function body, create the code replacement table.

```
crTable = RTW.TflTable;
```

Create a table entry for a signed fixed-point addition operation requiring the same slope across types.

```
tableEntry = createCRLEntry(crTable, ...
    'fixdt(1,16,~,0) y1 = fixdt(1,16,~,0) u1 + fixdt(1,16,~,0) u2', ...
    'int16 y1 = myAdd(int16 u1, int16 u2)');
```

Set entry parameters. Set algorithm parameters for a cast-after-sum addition and saturation and rounding modes. To generate the replacement code, specify that the code generator use the header and source files `myIntAdd.h` and `myIntAdd.c`.

```
setTflCOperationtionEntryParameters(tableEntry, ...
    'EntryInfoAlgorithm', 'RTW_CAST_AFTER_OP', ...
    'SaturationMode', 'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingMode', 'RTW_ROUND_SIMPLEST', ...
    'ImplementationHeaderFile', 'myIntAdd.h', ...
    'ImplementationSourceFile', 'myIntAdd.c');
```

Add the entry to the table.

```
addEntry(crTable, tableEntry);
```

Replacement Entry That Assumes Implementation and Conceptual Argument Data Types Are the Same

Create a table definition file that contains a function definition.

```
function crTable = crl_table_sinfcn()
```

Within the function body, create the code replacement table.

```
crTable = RTW.TflTable;
```

Create a table entry for a `sin` function, where the implementation arguments are the same as the conceptual arguments.

```
tableEntry = createCRLEntry(crTable, ...
    'double y1 = sin(double u1)', ...
    'y1 = mySin(u1)');
```

Set entry parameters. To generate the replacement code, specify that the code generator use the header and source files `mySin.h` and `mySin.c`.

```
setTflCFunctionEntryParameters(tableEntry, ...
    'ImplementationHeaderFile', 'mySin.h', ...
    'ImplementationSourceFile', 'mySin.c');
```

Add the entry to the table.

```
addEntry(crTable, tableEntry);
```

Input Arguments

`crTable` — Code replacement table
object

Table that stores one or more code replacement entries, each representing a potential replacement for a function or operator. Each entry maps a conceptual representation of a function or operator to an implementation representation and priority.

`conceptualSpecification` — Conceptual specification
character vector | string scalar

Representation of the name or symbol and conceptual input and output arguments for a function or operator that the software replaces, specified as a character vector or string scalar. Conceptual arguments observe naming conventions ('y1', 'u1', 'u2', ...) and data types familiar to the code generator. Use the syntax table in "Description" on page 1-157 to determine the syntax to use for your conceptual argument specification.

Example: `'double y1 = sin(double u1)'`

Example: `'int16 y1 = int16 u1 + int16 u2'`

`implementationSpecification` — Implementation specification
character vector | string scalar

Representation of the name and implementation input and output arguments for a C or C++ replacement function, specified as a character vector or string scalar. Implementation arguments

observe C/C++ name and data type specifications. Use the syntax table in “Description” on page 1-157 to determine the syntax for your implementation argument specification.

Example: `'double y1 = my_sin(double u1)'`

Example: `'int16 y1 = myAdd(int16 u1, int16 u2)'`

Output Arguments

`tableEntry` — Code replacement table entry
object

Code replacement table entry that represents a potential code replacement for a function or operator, returned as an object. Maps the conceptual representation of a function or operator, `conceptualSpecification`, to the C/C++ implementation representation, `implementationSpecification`.

See Also

`RTW.TflTable` | `addEntry` | `setTflCFunctionEntryParameters`

Topics
“Define Code Replacement Library Optimizations”
“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”

Introduced in R2015a

createToleranceFile

Create file correlating tolerance information with signal names

Syntax

```
cgvObj.createToleranceFile(file_name , signal_list, tolerance_list)
```

Description

`cgvObj.createToleranceFile(file_name , signal_list, tolerance_list)` creates a MATLAB file, named `file_name`, containing the tolerance specification for each output signal name in `signal_list`. Each signal name in the `signal_list` corresponds to the same location of a parameter name and value pair in the `tolerance_list`.

Input Arguments

file_name

Name for the file containing the tolerance specification for each signal. Use this file as input to `cgv.CGV.compare` and `cgv.Batch.addTest`.

signal_list

A cell array of character vectors, where each vector is a signal name for data from the model. Use `getSavedSignals` to view the list of available signal names in the output data. `signal_list` can contain an individual signal or multiple signals. The syntax for an individual signal name is:

```
signal_list = {'log_data.subsystem_name.Data(:,1)'}
```

The syntax for multiple signal names is:

```
signal_list = {'log_data.block_name.Data(:,1)',...
'log_data.block_name.Data(:,2)',...
'log_data.block_name.Data(:,3)',...
'log_data.block_name.Data(:,4)'};
```

To specify a global tolerance for the signals, include the reserved signal name, `'global_tolerance'`, in `signal_list`. Assign a global tolerance value in the associated `tolerance_list`. If `signal_list` contains other signals, their associated tolerance value overrides the global tolerance value. In this example, the global tolerance is a relative tolerance of `0.02`.

```
signal_list = {'global_tolerance',...
'log_data.block_name.Data(:,1)',...
'log_data.block_name.Data(:,2)'};

tolerance_list = {{'relative', 0.02},...
    {'relative', 0.015},{'absolute', 0.05}};
```

Note If a model component contains a space or newline character, MATLAB adds parentheses and a single quote to the name of the component. For example, if the signal name has a space, `'block name'`, MATLAB displays the signal name as:

```
log_data.('block name').Data(:,1)
```

To use the signal name as input to a CGV function, `'block name'` must have two single quotes in the `signal_list`. For example:

```
signal_list = {'log_data.(''block name'').Data(:,1)'}
```

`tolerance_list`

Cell array of cell arrays. Each element of the outer cell array is a cell array containing a parameter name and value pair for the type of tolerance and its value. Possible parameter names are `'absolute'` | `'relative'` | `'function'`. There is a one-to-one mapping between each parameter name and value pair in the `tolerance_list` and a signal name in the `signal_list`. For example, a `tolerance_list` for a `signal_list` containing four signals might look like the following:

```
tolerance_list = {{'relative', 0.02},{'absolute', 0.06},...
    {'relative', 0.015},{'absolute', 0.05}};
```

See Also

Topics
"Verify Numerical Equivalence with CGV"

crossReleaseImport

Import generated model code from a previous release as SIL or PIL blocks

Syntax

```
blockHandle = crossReleaseImport(buildFolder,
configSetOrModel,'SimulationMode',mode)
blockHandle = crossReleaseImport(buildFolder,
configSetOrModel,'SimulationMode',mode,'ConfigParams',
additionalParameterList)
blockHandle = crossReleaseImport(buildFolder,
configSetOrModel,'SimulationMode',mode,'DataDictionary',dictionaryFile)
blockHandle = crossReleaseImport(buildFolder,
configSetOrModel,'SimulationMode',mode,'OriginalPaths',
originalPaths,'ReplacementPaths',replacementPaths)
blockHandle = crossReleaseImport(buildFolder,
configSetOrModel,'SimulationMode',mode,'SFunctionName',sFunctionName)
```

Description

`blockHandle = crossReleaseImport(buildFolder, configSetOrModel,'SimulationMode',mode)` imports previously generated model component code into the current release. The function imports the code as a cross-release block and returns the numeric handle of the block. The function displays the block in a new model window.

In an existing model, you can replace the model component with the cross-release block.

If you set `'SimulationMode'` to, for example, `'SIL'` or `'PIL'`, the function imports the code as a software-in-the-loop (SIL) or processor-in-the-loop (PIL) block. When you run a simulation or build the model, the model component uses generated code from the previous release.

To build a SIL or PIL block, the function by default uses the following parameters of the Simulink model specified by `configSetOrModel`:

- `SystemTargetFile`
- `Toolchain` or `TemplateMakefile`
- `ExistingSharedCode`
- `PortableWordSizes`
- `TargetLang`
- `TargetLangStandard`
- `TargetLibSuffix`
- `ModelReferenceNumInstancesAllowed`
- **Hardware Implementation** pane parameters

If you set `'SimulationMode'` to `'none'`, the function creates a Cross-Release Code Integration block, which:

- Supports generation of code that calls the imported code.
- Does not support normal, accelerator, or rapid accelerator mode simulations.
- Does not compile the imported code.

You can use the Cross-Release Code Integration block, for example, in workflows where compilation occurs on a different computer.

`blockHandle = crossReleaseImport(buildFolder, configSetOrModel,'SimulationMode',mode,'ConfigParams', additionalParameterList)` uses additional configuration parameters for building the SIL or PIL block.

`blockHandle = crossReleaseImport(buildFolder, configSetOrModel,'SimulationMode',mode,'DataDictionary',dictionaryFile)` imports generated code that uses data types specified by a data dictionary. If `configSetOrModel` is a model associated with a data dictionary, you do not have to specify the name-value pair. By default, the function identifies and uses the data dictionary when it imports the generated code. If you specify a name-value pair, the data dictionary that you specify takes precedence over the default data dictionary.

`blockHandle = crossReleaseImport(buildFolder, configSetOrModel,'SimulationMode',mode,'OriginalPaths', originalPaths,'ReplacementPaths',replacementPaths)` imports generated model code with relocated custom code or modified include paths. The paths specified by `replacementPaths` override the original custom code or include paths specified by `originalPaths` in a one-to-one manner. You cannot use `replacementPaths` to specify additional custom code or include paths.

`blockHandle = crossReleaseImport(buildFolder, configSetOrModel,'SimulationMode',mode,'SFunctionName',sFunctionName)` names the generated SIL or PIL block `sFunctionName_sil` or `sFunctionName_pil`. Use the `sFunctionName` argument if the default block name produces associated MATLAB identifiers that are longer than 63 characters.

Examples

Import Generated Code from Previous Release

This example shows how to import generated model code from a previous release.

Specify the location of the build folder.

```
buildFolder = fullfile(pwd,'R2015bWork', 'folderPathForP1_ert_rtw');
```

Import code for the integration model `Controller`.

```
crossReleaseImport(buildFolder,'Controller','SimulationMode','SIL');
```

The function displays a SIL block in a new Simulink editor window.

Input Arguments

`buildFolder` — Build folder
character vector

Build folder that contains generated model component code from a previous release.

`configSetOrModel` — Configuration object or model
`Simulink.ConfigSet` | character vector

A configuration set or Simulink model on the MATLAB path.

`mode` — Block mode
`'SIL'` | `'PIL'` | `{'SIL','PIL'}` | `'none'`

Simulation mode for block with imported code:

- `'SIL'` — Create SIL block.
- `'PIL'` — Create PIL block.
- `{'SIL','PIL'}` — Create SIL and PIL blocks.
- `'none'` — Create Cross-Release Code Integration block.

`additionalParameterList` — Additional parameters
cell array of character vectors

Additional parameters for building the SIL or PIL block.

`dictionaryFile` — Dictionary file
character vector

Data dictionary that specifies data types used by the generated code.

`originalPaths` — Original custom code folders or include paths
character vector | cell array of character vectors | string array

Folder or include paths for original custom code. Must have a one-to-one correspondence with `replacementPaths`.

`replacementPaths` — Replacement custom code folders or include paths
character vector | cell array of character vectors | string array

Folder or include paths for relocated custom code. Must have a one-to-one correspondence with `originalPaths`.

`sFunctionName` — Name for SIL or PIL block
character vector | cell array of character vectors | string array

Specify name for SIL or PIL block that contains generated code from previous release. If the default block name produces associated MATLAB identifiers that are longer than 63 characters, use this argument to specify a shorter block name.

Output Arguments

`blockHandle` — Numeric handle of a block
double|array of doubles

Numeric handle of a block. Returned as a double if `mode` is `'SIL'` or `'PIL'`. Returned as an array of doubles if `mode` is `{'SIL','PIL'}`.

See Also
`sharedCodeUpdate`

Topics
"Cross-Release Shared Utility Code Reuse"
"Cross-Release Code Integration"

Introduced in R2016b

display

Package: `coder.profile`

Generate message that describes how to open code execution profiling report

Syntax

`myExecutionProfile.display`

Description

`myExecutionProfile.display` generates a message that describes how you can open the code execution profiling report.

Examples

Display Code Execution Profiling Report

To generate a message that describes how to open the code execution profiling report, use the display function and the `myExecutionProfile` workspace variable.

```
myExecutionProfile.display
```

Input Arguments

`myExecutionProfile` — Variable specifies annotation
workspace variable

`myExecutionProfile` is a workspace variable, specified through the configuration parameter `CodeExecutionProfileVariable` and generated by a simulation.

Example: `myExecutionProfile`

See Also

`report`

Topics
"Code Execution Profiling with SIL and PIL"
"View and Compare Code Execution Times"

Introduced in R2011a

displayReport

Display results of comparing configuration parameter values

Syntax

```
cfgObj.displayReport()
```

Description

`cfgObj.displayReport()` displays the results at the MATLAB Command Window of comparing the configuration parameter values for the model with the values that the object recommends. *`cfgObj`* is a handle to a `cgv.Config` object.

See Also

Topics
"Verify Numerical Equivalence Between Two Modes of Execution of a Model"

coder.MATLABCodeTemplate.emitSection

Class: `coder.MATLABCodeTemplate`
Package: `coder`

Emit comments for template section

Syntax

```
sectionComments = emitSection(sectionName,isCPPComment)
```

Description

`sectionComments = emitSection(sectionName,isCPPComment)` emits comments for the code template section that `sectionName` specifies. If `isCPPComment` is `true`, `emitSection` uses C++ style comments. If `emitSection` is `false`, it uses C style comments. Use `emitSection` to preview banners before you generate code. Before invoking `emitSection` to emit the banner for a template section, you must set the values for all tokens used in that section.

Input Arguments

`sectionName` — Name of templates section

character vector

Name of template section specified as one of the following values:

- `'FileBanner'`
- `'FunctionBanner'`
- `'SharedUtilityBanner'`
- `'FileTrailer'`
- `'IncludeFilesBanner'`
- `'TypeDefinitionsBanner'`
- `'NamedConstantsBanner'`
- `'VariableDeclarationsBanner'`
- `'VariableDefinitionsBanner'`
- `'FunctionDeclarationsBanner'`
- `'FunctionDefinitionsBanner'`
- `'CustomSourceCodeBanner'`
- `'CustomHeaderCodeBanner'`

`isCPPComment` — C++ comment style flag

true | false

Specify `true` for C++ style comments. Specify `false` for C style comments.

Output Arguments

`sectionComments` — Comments for template section

character vector

Comments for the specified section, returned as a character vector.

Examples

Emit File Banner from Default Template

This example shows how to set the `FileName` token value and emit the default file banner.

Create a `coder.MATLABCodeTemplate` object from the default template.

```
newObj = coder.MATLABCodeTemplate
```

Set the `FileName` token value.

```
fileN = 'myfilename.c';
newObj.setTokenValue('FileName', fileN)
```

Emit the file banner.

```
newObj.emitSection('FileBanner', false)
```

The `emitSection` method generates the file banner replacing the `FileName` token with the file name that you specified. It replaces the `MATLABCoderVersion` token with the current MATLAB Coder version number. It replaces the `SourceGeneratedOn` token with the time stamp.

```
/*
 * File: myfilename.c
 *
 * MATLAB Coder version            : 2.7
 * C/C++ source code generated on  : 07-Apr-2014 17:43:32
 */
```

Emit Include Files Banner from Custom Template

This example shows how to create and modify a custom code generation template (CGT) file. It shows how to emit the include files section banner from the custom CGT file.

Create a local copy of the default CGT file for MATLAB Coder. Name it `myCGTFile.cgt`.

In your local copy of the CGT File, in the `IncludeFilesBanner` open tag, change the style to `"box"`.

```
<IncludeFilesBanner style="box">
Include Files
</IncludeFilesBanner>
```

Create a `MATLABCodeTemplate` object from your custom CGT file.

```
CGTFile = 'myCGTFile.cgt';
newObj= coder.MATLABCodeTemplate(CGTFile);
```

Emit the include files section banner using C++ style comments.

```
newObj.emitSection('IncludeFilesBanner', true)
```

The `emitSection` method generates the include files section banner using the box style with C++ style comments.

```
////////////////////////////////////////////////////////////////////////////////
// Include Files                                                              //
////////////////////////////////////////////////////////////////////////////////
```

See Also

`coder.MATLABCodeTemplate.getCurrentTokens` | `coder.MATLABCodeTemplate.getTokenValue` | `coder.MATLABCodeTemplate.setTokenValue`

Topics

"Generate Custom File and Function Banners for C/C++ Code"
"Code Generation Template Files for MATLAB Code"

enableCPP

Enable C++ support for function entry in code replacement table

Syntax

`enableCPP(hEntry)`

Description

`enableCPP(hEntry)` enables C++ support for a function entry in a code replacement table. This support allows you to specify a C++ namespace for the implementation function defined in the entry (see the `setNameSpace` function).

When you register a code replacement library containing C++ function entries, you must specify the value `{'C++'}` for the `LanguageConstraint` property of the code replacement registry entry. For more information, see “Register Code Replacement Library”.

Examples

Enable C++ Support for Function Entry

This example shows how to use the `enableCPP` function to enable C++ support. Then, the example calls the `setNameSpace` function to set the namespace for the `sin` implementation function to `std`.

```
fcn_entry = RTW.TflCFunctionEntry;
fcn_entry.setTflCFunctionEntryParameters( ...
    'Key',                      'sin', ...
    'Priority',                 100, ...
    'ImplementationName',       'sin', ...
    'ImplementationHeaderFile', 'cmath' );
fcn_entry.enableCPP();
fcn_entry.setNameSpace('std');
```

Input Arguments

`hEntry` — Handle to a code replacement function entry
handle

The *`hEntry`* is a handle to a code replacement function entry previously returned by *`hEntry`* `= RTW.TflCFunctionEntry` or *`hEntry = MyCustomFunctionEntry`*. The *`MyCustomFunctionEntry`* is a class derived from `RTW.TflCFunctionEntry`.

Example: `fcn_entry`

See Also

`registerCPPFunctionEntry` | `setNameSpace`

Topics
“Math Function Code Replacement”

"Define Code Replacement Library Optimizations"
"Code You Can Replace from MATLAB Code"
"Code You Can Replace From Simulink Models"

Introduced in R2010a

excludeCheck

Package: `rtw.codegenObjectives`

Exclude check from code generation objective

Syntax

`excludeCheck(objective, checkID)`

Description

`excludeCheck(objective, checkID)` excludes the specified check from the Code Generation Advisor when you specify the objective. When you select multiple objectives, if you specify an additional objective that includes this check as a higher priority objective, the Code Generation Advisor includes this check.

Examples

Create an Objective Based On an Existing Objective

Create a custom objective based on the `Traceability` objective.

Create a file `sl_customization.m` to contain a callback function that creates the custom objective.

```
function sl_customization(cm)
%SL_CUSTOMIZATION objective customization callback

objCustomizer = cm.ObjectiveCustomizer;
index = objCustomizer.addCallbackObjFcn(@addObjectives);
objCustomizer.callbackFcn{index}();

end
```

Create and configure the objective in the `addObjectives` function. Set the name of the objective and modify the list of checks, parameters, and values to verify. Then register the objective in the Code Generation Advisor.

```
function addObjectives

% Create the custom objective
obj = rtw.codegenObjectives.Objective('ex_trace', 'Traceability');
setObjectiveName(obj, 'Custom Traceability Example');

% Remove inherited parameters from the objective
removeInheritedParam(obj, 'MATLABFcnDesc');
removeInheritedParam(obj, 'MATLABSourceComments');

% Remove the inherited code instrumentation check
removeInheritedCheck(obj, 'mathworks.codegen.CodeInstrumentation');

% Modify the inherited parameter 'ConcertIfToSwitch' and set the value to 'on'
modifyInheritedParam(obj, 'ConvertIfToSwitch', 'on');

% Exclude the inherited check for the software environment
```

```
excludeInheritedCheck(obj, 'mathworks.codegen.SWEnvironmentSpec');

%Register the objective
register(obj);

end
```

Input Arguments

`objective` — Code generation objective

`rtw.codegenObjectives.Objective` object

Code generation objective, specified as a `rtw.codegenObjectives.Objective` object.

`checkID` — Identifier of check

character vector | string scalar

Identifier of check that you want to exclude, specified as a character vector or string scalar.

Example: `'mathworks.codegen.CodeInstrumentation'`

See Also

`Simulink.ModelAdvisor`

Topics

"Create Custom Code Generation Objectives"
`Simulink.ModelAdvisor`

Introduced in R2009a

getAlgorithmParameters

Examine algorithm parameter settings for lookup table function code replacement table entry

Syntax

```
algParams = getAlgorithmParameters(tableEntry)
```

Description

`algParams = getAlgorithmParameters(tableEntry)` returns the algorithm parameter settings for the lookup table function identified in the code replacement table entry `tableEntry`. If you call `getAlgorithmParameters` before using `setAlgorithmParameters`, `getAlgorithmParameters` lists the default parameter settings for the lookup table function.

Examples

Examine Default Parameter Settings for prelookup Table Entry

Create a code replacement table.

```
crTable = RTW.TflTable;
```

Create a table entry for a function.

```
tableEntry = RTW.TflCFunctionEntry;
```

Identify the table entry as an entry for the `prelookup` function.

```
setTflCFunctionEntryParameters(tableEntry, ...
    'Key', 'prelookup', ...
    'Priority', 100, ...
    'ImplementationName', 'myPrelookup');
```

Get the algorithm parameter settings for the `prelookup` function table entry.

```
algParams = getAlgorithmParameters(tableEntry)
```

```
algParams = 

  Prelookup with properties:

             ExtrapMethod: [1x1 coder.algorithm.parameter.ExtrapMethod]
                  RndMeth: [1x1 coder.algorithm.parameter.RndMeth]
        IndexSearchMethod: [1x1 coder.algorithm.parameter.IndexSearchMethod]
        UseLastBreakpoint: [1x1 coder.algorithm.parameter.UseLastBreakpoint]
    RemoveProtectionInput: [1x1 coder.algorithm.parameter.RemoveProtectionInput]
```

Examine the information for parameter `ExtrapMethod`.

```
algParams.ExtrapMethod
```

```
ans = 

  ExtrapMethod with properties:

       Name: 'ExtrapMethod'
    Options: {'Linear'  'Clip'}
```

```
    Primary: 1
      Value: {'Linear'}
```

Examine the information for parameter `RndMeth`.

```
algParams.RndMeth
```

```
ans = 

  RndMeth with properties:

       Name: 'RndMeth'
    Options: {1x7 cell}
    Primary: 0
      Value: {1x7 cell}
```

Examine the current `Value` setting.

```
algParams.RndMeth.Value
```

```
ans = 

  Columns 1 through 6

    'Ceiling'    'Convergent'    'Floor'    'Nearest'    'Round'    'Simplest'

  Column 7

    'Zero'
```

Examine the information for parameter `IndexSearchMethod`.

```
algParams.IndexSearchMethod
```

```
ans = 

  IndexSearchMethod with properties:

       Name: 'IndexSearchMethod'
    Options: {'Linear search'  'Binary search'  'Evenly spaced points'}
    Primary: 0
      Value: {'Binary search'  'Evenly spaced points'  'Linear search'}
```

Examine the information for parameter `UseLastBreakpoint`.

```
algParams.UseLastBreakpoint
```

```
ans = 

  UseLastBreakpoint with properties:

       Name: 'UseLastBreakpoint'
    Options: {'off'  'on'}
    Primary: 0
      Value: {'off'  'on'}
```

Examine the information for parameter `RemoveProtectionInput`.

```
algParams.RemoveProtectionInput
```

```
ans = 

  RemoveProtectionInput with properties:

       Name: 'RemoveProtectionInput'
    Options: {'off'  'on'}
    Primary: 0
      Value: {'off'  'on'}
```

Examine Modified Parameter Setting for `lookup2D` Table Entry

Create a code replacement table.

```
crTable = RTW.TflTable;
```

Create a table entry for a function.

```
tableEntry = RTW.TflCFunctionEntry;
```

Identify the table entry as an entry for the `lookup2D` function.

```
setTflCFunctionEntryParameters(tableEntry, ...
    'Key', 'lookup2D', ...
    'Priority', 100, ...
    'ImplementationName', 'myLookup2D');
```

Get the algorithm parameter settings for the `lookup2D` function table entry.

```
algParams = getAlgorithmParameters(tableEntry)
```

```
algParams = 

  Lookup with properties:

                InterpMethod: [1x1 coder.algorithm.parameter.InterpMethod]
                ExtrapMethod: [1x1 coder.algorithm.parameter.ExtrapMethod]
         UseRowMajorAlgorithm: [1x1 coder.algorithm.parameter.UseRowMajorAlgorithm]
                     RndMeth: [1x1 coder.algorithm.parameter.RndMeth]
           IndexSearchMethod: [1x1 coder.algorithm.parameter.IndexSearchMethod]
           UseLastTableValue: [1x1 coder.algorithm.parameter.UseLastTableValue]
       RemoveProtectionInput: [1x1 coder.algorithm.parameter.RemoveProtectionInput]
   SaturateOnIntegerOverflow: [1x1 coder.algorithm.parameter.SaturateOnIntegerOverflow]
     SupportTunableTableSize: [1x1 coder.algorithm.parameter.SupportTunableTableSize]
             BPPower2Spacing: [1x1 coder.algorithm.parameter.BPPower2Spacing]
```

Display the possible index search method settings.

```
algParams.IndexSearchMethod.Options
```

```
ans = 

    'Linear search'    'Binary search'    'Evenly spaced points'
```

Display the current index search method setting.

```
algParams.IndexSearchMethod.Value
```

```
ans = 

    'Linear search'    'Binary search'    'Evenly spaced points'
```

By default, the parameter is set to the same value set.

Set the index search method to binary search.

```
algParams.IndexSearchMethod = 'Binary search';
```

Verify the modified parameter setting.

```
algParams.IndexSearchMethod.Value
```

```
ans = 

    'Binary search'
```

Input Arguments

`tableEntry` — Code replacement table entry for a lookup table function

object

Code replacement table entry that you previously created and represents a potential code replacement for a lookup table function. The entry must identify the lookup table function for which you are calling `getAlgorithmParameters`.

1. Create the entry. For example, call the function `RTW.TflCFunctionEntry`.

```
tableEntry = RTW.TflCFunctionEntry;
```

2 Specify the name of the lookup table function for which you created the entry. Use the `Key` parameter in a call to `setTflCFunctionEntryParameters`. The following function call specifies the lookup table function `prelookup`.

```
setTflCFunctionEntryParameters(tableEntry, ...
   'Key', 'prelookup', ...
   'Priority', 100, ...
   'ImplementationName', 'myPrelookup');
```

Output Arguments

`algParams` — Algorithm parameter settings for a lookup table function
object

Algorithm parameter settings for the lookup table function identified with the `Key` parameter in `tableEntry`.

See Also

`RTW.TflCFunctionEntry` | `RTW.TflTable` | `addEntry` | `setAlgorithmParameters` | `setTflCFunctionEntryParameters`

Topics
"Lookup Table Function Code Replacement"
"Define Code Replacement Library Optimizations"
"Code You Can Replace from MATLAB Code"
"Code You Can Replace From Simulink Models"

Introduced in R2015a

getArgCategory

Get argument category for Simulink model port from model-specific C function prototype

Syntax

`category = getArgCategory(obj, portName)`

Description

`category = getArgCategory(obj, portName)` gets the category, `'Value'` or `'Pointer'`, of the argument corresponding to a specified Simulink model inport or outport from a specified model-specific C function prototype.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.getFunctionSpecification (modelName)`.
portName	Character vector specifying the name of an inport or outport in your Simulink model.

Output Arguments

category	Character vector specifying the argument category, `'Value'` or `'Pointer'`, for the specified Simulink model port.

Alternatives

Use the Configure C Step Function Interface dialog box to configure C step function arguments. See “Configure Name and Arguments for Individual Step Functions”.

See Also

Topics
“Configure C Code Generation for Model Entry-Point Functions”

getArgName

Get argument name for Simulink model port from model-specific C function prototype

Syntax

`argName = getArgName(obj, portName)`

Description

`argName = getArgName(obj, portName)` gets the argument name corresponding to a specified Simulink model inport or outport from a specified model-specific C function prototype.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.getFunctionSpecification (modelName)`.
portName	Character vector specifying the name of an inport or outport in your Simulink model.

Output Arguments

argName	Character vector specifying the argument name for the specified Simulink model port.

Alternatives

Use the Configure C Step Function Interface dialog box to configure C step function arguments. See "Configure Name and Arguments for Individual Step Functions".

See Also

Topics
"Configure C Code Generation for Model Entry-Point Functions"

getArgPosition

Get argument position for Simulink model port from model-specific C function prototype

Syntax

`position = getArgPosition(obj, portName)`

Description

`position = getArgPosition(obj, portName)` gets the position — 1 for first, 2 for second, etc. — of the argument corresponding to a specified Simulink model inport or outport from a specified model-specific C function prototype.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.getFunctionSpecification (modelName)`.
portName	Character vector specifying the name of an inport or outport in your Simulink model.

Output Arguments

position	Integer specifying the argument position — 1 for first, 2 for second, etc. — for the specified Simulink model port. Without an argument for the specified port, the function returns 0.

Alternatives

Use the Configure C Step Function Interface dialog box to configure C step function arguments. See “Configure Name and Arguments for Individual Step Functions”.

See Also

Topics
“Configure C Code Generation for Model Entry-Point Functions”

getArgQualifier

Get argument type qualifier for Simulink model port from model-specific C function prototype

Syntax

`qualifier = getArgQualifier(obj, portName)`

Description

`qualifier = getArgQualifier(obj, portName)` gets the type qualifier — `'none'`, `'const'`, `'const *'`, or `'const * const'`— of the argument corresponding to a specified Simulink model inport or outport from a specified model-specific C function prototype.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.getFunctionSpecification (modelName)`.
portName	Character vector specifying the name of an inport or outport in your Simulink model.

Output Arguments

qualifier	Character vector specifying the argument type qualifier — `'none'`, `'const'`, `'const *'`, or `'const * const'`— for the specified Simulink model port.

Alternatives

Use the Configure C Step Function Interface dialog box to configure C step function arguments. See “Configure Name and Arguments for Individual Step Functions”.

See Also

Topics
“Configure C Code Generation for Model Entry-Point Functions”

getCoderExecutionProfile

Package: `coder.profile.mc`

Extract execution-time profile for code generated from MATLAB function (MATLAB code generation)

Syntax

`myExecutionProfile = getCoderExecutionProfile('myMATLABFunction')`

Description

`myExecutionProfile = getCoderExecutionProfile('myMATLABFunction')` creates a workspace variable that contains the execution-time profile of the code generated from your MATLAB function. Run the command after the completion and termination of the SIL/PIL execution of your MATLAB function.

Examples

Get Coder Execution Profile

To get the coder execution profile for a MATLAB function, use the `getCoderExecutionProfile` function.

```
myExecutionProfile = getCoderExecutionProfile('myMATLABFunction');
```

Input Arguments

`myMATLABFunction` — Function whose generate code is profiled
function

The `myMATLABFunction` is the MATLAB function whose generate code is profiled for execution-time.

Example: `myMATLABFunction`

Output Arguments

`myExecutionProfile` — Workspace variable that contains the execution-time profile
workspace variable

The `myExecutionProfile` is a workspace variable that contains the execution-time profile of the code generated from your MATLAB function.

See Also

`Sections` | `TimerTicksPerSecond` | `report`

Topics
"Generate Execution Time Profile"

"Analyze Execution Time Data"

Introduced in R2014b

coder.MATLABCodeTemplate.getCurrentTokens

Class: `coder.MATLABCodeTemplate`
Package: `coder`

Get current tokens

Syntax

`currentTokens = getCurrentTokens()`

Description

`currentTokens = getCurrentTokens()` returns list of current tokens in the MATLABCodeTemplate object

Output Arguments

`currentTokens` — Current tokens
cell array of character vectors

A list of current tokens in the MATLABCodeTemplate object, returned as a cell array of character vectors.

Examples

Create a MATLABCodeTemplate object with the default template, then list its tokens.

```
newObj = coder.MATLABCodeTemplate;
% Creates a MATLABCodeTemplate object from the default template
newObj.getCurrentTokens()
% Returns a list of tokens for the template
```

See Also

`coder.MATLABCodeTemplate.emitSection` | `coder.MATLABCodeTemplate.getTokenValue` | `coder.MATLABCodeTemplate.setTokenValue`

Topics

"Generate Custom File and Function Banners for C/C++ Code"
"Code Generation Template Files for MATLAB Code"

getDefaultConf

Get default configuration information for model-specific C function prototype from Simulink model

Syntax

```
getDefaultConf(obj)
```

Description

`getDefaultConf(obj)` invokes the specified model-specific C function prototype to initialize the properties and the step function name of the function argument to a default configuration based on information from the ERT-based Simulink model to which it is attached. If you invoke the command again, only the properties of the function argument are reset to default values.

Before calling this function, you must call `attachToModel`, to attach the function prototype to a loaded model.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.ModelSpecificCPrototype`.

Alternatives

Use the **Get default** button on the Configure C Step Function Interface dialog box to get the default configuration. See “Configure Name and Arguments for Individual Step Functions”.

See Also

Topics
“Configure C Code Generation for Model Entry-Point Functions”

getFunctionName

Get function name from model-specific C function prototype

Syntax

`fcnName = getFunctionName(obj, fcnType)`

Description

`fcnName = getFunctionName(obj, fcnType)` gets the name of the step or initialize function described by the specified model-specific C function prototype.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.getFunctionSpecification(modelName)`.
fcnType	Optional character vector specifying which function name to get. Valid options are `'step'` and `'init'`. If *fcnType* is not specified, gets the step function name.

Output Arguments

fcnName	A character vector specifying the name of the function described by the specified model-specific C function prototype.

See Also

Topics

"Configure C Code Generation for Model Entry-Point Functions"

Name

Package: `coder.profile`

Get name of profiled code section

Syntax

`SectionName=NthSectionProfile.Name`

Description

`SectionName=NthSectionProfile.Name` returns the name that identifies the profiled code section. The software generates an identifier based on the model entity that corresponds to the profiled section of code.

Examples

Get Name of Profiled Code Section

To get the name that identifies the profiled code section, use the `Name` property of the `NthSectionProfile` object.

```
SectionName = NthSectionProfile.Name;
```

Input Arguments

`NthSectionProfile` — `coder.profile.ExecutionTimeSection` object
`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

`SectionName` — Profiled section name
section name

Name that identifies profiled code section.

See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `display` | `report`

Topics
"Code Execution Profiling with SIL and PIL"
"View and Compare Code Execution Times"
"Analyze Code Execution Data"

Introduced in R2012b

Name

Package: `coder.profile.mc`

Get name of profiled code section (MATLAB code generation)

Syntax

`SectionName = NthSectionProfile.Name`

Description

`SectionName = NthSectionProfile.Name` returns the name that identifies the profiled code section.

Examples

Get Name of Profiled Code Section

To get the name that identifies the profiled code section, use the `Name` property of the `NthSectionProfile` object.

```
SectionName = NthSectionProfile.Name;
```

Input Arguments

`NthSectionProfile` — ExecutionTimeSection object
`ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile.MaximumExecutionTimeInTicks`

Output Arguments

`SectionName` — Profiled section name
section name

The `SectionName` is the name that identifies profiled code section.

See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `getCoderExecutionProfile` | `report`

Topics
“Generate Execution Time Profile”
“Analyze Execution Time Data”

Introduced in R2012b

getNumArgs

Get number of function arguments from model-specific C function prototype

Syntax

`num = getNumArgs(obj)`

Description

`num = getNumArgs(obj)` gets the number of function arguments for the function described by the specified model-specific C function prototype.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.getFunctionSpecification(modelName)`.

Output Arguments

num	An integer specifying the number of function arguments.

Alternatives

Use the Configure C Step Function Interface dialog box to view C step function arguments. See “Configure Name and Arguments for Individual Step Functions”.

See Also

Topics
“Configure C Code Generation for Model Entry-Point Functions”

NumCalls

Package: `coder.profile`

Total number of calls to profiled code section

Syntax

`TotalNumCalls = NthSectionProfile.NumCalls`

Description

`TotalNumCalls = NthSectionProfile.NumCalls` returns the total number of calls to the profiled code section over the entire simulation.

Examples

Get Number of Calls to Profiled Section

To get the total number of calls to the profiled code section over the entire simulation, use the `NumCalls` property of the `NthSectionProfile` object.

```
TotalNumCalls = NthSectionProfile.NumCalls;
```

Input Arguments

`NthSectionProfile` — `coder.profile.ExecutionTimeSection` object
`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

`TotalNumCalls` — Total number of calls
integer

The total number of calls to the profiled section of code.

See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `display` | `report`

Topics
"Code Execution Profiling with SIL and PIL"
"View and Compare Code Execution Times"
"Analyze Code Execution Data"

Introduced in R2012b

NumCalls

Package: `coder.profile.mc`

Total number of calls to profiled code section (MATLAB code generation)

Syntax

`TotalNumCalls = NthSectionProfile.NumCalls`

Description

`TotalNumCalls = NthSectionProfile.NumCalls` returns the total number of calls to the profiled code section over the entire execution.

Examples

Get Number of Calls to Profiled Section

To get the total number of calls to the profiled code section over the entire simulation, use the `NumCalls` property of the `NthSectionProfile` object.

```
TotalNumCalls = NthSectionProfile.NumCalls;
```

Input Arguments

`NthSectionProfile` — `coder.profile.ExecutionTimeSection` object
`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

`TotalNumCalls` — Total number of calls
integer

The total number of calls to the profiled section of code.

See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `getCoderExecutionProfile` | `report`

Topics
"Generate Execution Time Profile"
"Analyze Execution Time Data"

Introduced in R2012b

getOutputData

Get output data

Syntax

`out = cgvObj.getOutputData(InputIndex)`

Description

`out = cgvObj.getOutputData(InputIndex)` is the method that you use to retrieve the output data that the object creates during execution of the model. *`out`* is the output data that the object returns. *`cgvObj`* is a handle to a `cgv.CGV` object. *`InputIndex`* is a unique numeric identifier that specifies which output data to retrieve. The *`InputIndex`* is associated with specific input data.

See Also

Topics

"Verify Numerical Equivalence with CGV"

getPreview

Get model-specific C function prototype code preview

Syntax

`preview = getPreview(obj, fcnType)`

Description

`preview = getPreview(obj, fcnType)` gets the model-specific C function prototype code preview.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.getFunctionSpecification(modelName)`.
fcnType	Optional. Character vector specifying which function to preview. Valid options are `'step'` and `'init'`. If *fcnType* is not specified, previews the step function.

Output Arguments

preview	Character vector specifying the function prototype for the step or initialization function.

Alternatives

Use the **C function prototype** field in the Configure C Step Function Interface dialog box to preview how your step function prototype is interpreted in generated code. See "Configure Name and Arguments for Individual Step Functions".

See Also

Topics
"Configure C Code Generation for Model Entry-Point Functions"

getReportData

Return results of comparing configuration parameter values

Syntax

rpt_data = *cfgObj*.getReportData()

Description

rpt_data = *cfgObj*.getReportData() compares the original configuration parameter values with the values that the object recommends. *cfgObj* is a handle to a cgv.Config object. Returns a cell array of character vectors with the model, parameter, previous value, and recommended or new value.

See Also

Topics
"Verify Numerical Equivalence with CGV"

getSavedSignals

Display list of signal names to command line

Syntax

`signal_list = cgvObj.getSavedSignals(simulation_data)`

Description

`signal_list = cgvObj.getSavedSignals(simulation_data)` returns a cell array, *signal_list*, of the output signal names of the data elements from the input data set, *simulation_data*. *simulation_data* is the output data stored in the CGV object, *cgvObj*, when you execute the model.

Tips

- After executing your model, use the `getOutputData` function to get the output data used as the input argument to the `cgvObj.getSavedSignals` function.
- Use names from the output signal list at the command line or as input arguments to other CGV functions, for example, `createToleranceFile`, `compare`, and `plot`.

See Also

Topics
"Verify Numerical Equivalence with CGV"

Number

Package: `coder.profile`

Get number that uniquely identifies profiled code section

Syntax

`SectionNumber = NthSectionProfile.Number`

Description

`SectionNumber = NthSectionProfile.Number` returns a number that uniquely identifies the profiled code section, for example, in the code execution profiling report.

Examples

Get Number of Profiled Code Section

To get the number that identifies the profiled code section, use the `Number` property of the `NthSectionProfile` object.

```
SectionNumber = NthSectionProfile.Number;
```

Input Arguments

`NthSectionProfile` — `coder.profile.ExecutionTimeSection` object
`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

`SectionNumber` — Number of profiled section
integer

The `SectionNumber` is the number of the profiled code section.

See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `display` | `report`

Topics
"Code Execution Profiling with SIL and PIL"
"View and Compare Code Execution Times"
"Analyze Code Execution Data"

Introduced in R2012b

Number

Package: `coder.profile.mc`

Get number that uniquely identifies profiled code section (MATLAB code generation)

Syntax

`SectionNumber = NthSectionProfile.Number`

Description

`SectionNumber = NthSectionProfile.Number` returns a number that uniquely identifies the profiled code section.

Examples

Get Number of Profiled Code Section

To get the number that identifies the profiled code section, use the `Number` property of the `NthSectionProfile` object.

```
SectionNumber = NthSectionProfile.Number;
```

Input Arguments

`NthSectionProfile` — `coder.profile.ExecutionTimeSection` object
`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

`SectionNumber` — Number of profiled section
integer

The `SectionNumber` is the number of the profiled code section.

See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `getCoderExecutionProfile` | `report`

Topics

"Generate Execution Time Profile"
"Analyze Execution Time Data"

Introduced in R2012b

pil_block_replace

Replace block in model with block from another model

Syntax

```
pil_block_replace(sourceModelBlock, destinationModelBlock)
pil_block_replace(sourceModelBlock, destinationModelBlock, 'isvisible')
```

Description

`pil_block_replace(sourceModelBlock, destinationModelBlock)` replaces a block in the destination model with a block from the source model. To preserve the original block from the destination model, in the source model, the function replaces `sourceModelBlock` with `destinationModelBlock`.

`pil_block_replace(sourceModelBlock, destinationModelBlock, 'isvisible')` highlights the replaced block in the destination model.

Examples

Replace Destination Block with Source Block

This example shows how to replace a block in a model with a block from another model.

Create a destination model that contains an Outport block, `destinationBlock`.

```
new_system('destModel')
open_system('destModel');
add_block('simulink/Sinks/Out1', 'destModel/destinationBlock')
```

Create a source model that contains a Scope block, `sourceBlock`.

```
new_system('srcModel')
open_system('srcModel');
add_block('simulink/Sinks/Scope', 'srcModel/sourceBlock')
```

Replace the Outport block in the destination model with the Scope block from the source model.

```
pil_block_replace('srcModel/sourceBlock','destModel/destinationBlock','isvisible')
```

Input Arguments

`sourceModelBlock` — Source block
character vector

Full path to the replacement block in the source model.

Example: 'srcModel/sourceBlock'

`destinationModelBlock` — Destination block
character vector

Full path to the block in the destination model, which the source block replaces.

Example: 'destModel/destinationBlock'

See Also

Topics
"Cross-Release Code Integration"

Introduced in R2006b

piltest

Verify custom target connectivity configuration for Simulink PIL simulation

Syntax

```
piltest(config)
piltest(config,'ConfigParams',additionalParameterList)
piltest(config,'TestPoint',testName)
```

Description

`piltest(config)` runs a suite of tests that verify your custom processor-in-the-loop (PIL) target connectivity configuration. In the tests, the function runs various normal, software-in-the-loop (SIL), and PIL simulations. The function compares results and produces errors if it detects differences between simulation modes. For the PIL simulations, the function extracts these parameters from `config`:

- `SystemTargetFile`
- `TargetHWDeviceType`
- `Toolchain`

In the current working folder, the function creates the `piltest` folder, which contains subfolders with test results.

`piltest(config,'ConfigParams',additionalParameterList)` extracts additional parameters from `config` for the PIL simulation.

`piltest(config,'TestPoint',testName)` runs a specific test from the test suite.

Examples

Verify Target Connectivity Configuration with piltest

This example uses `piltest` to verify a target connectivity configuration for PIL simulations on your development computer.

Create a target connectivity implementation in your current working folder.

```
% Make a local copy of the connectivity classes.
src_dir = ...
    fullfile(matlabroot,'toolbox','coder','simulinkcoder',...
             '+coder','+mypil');
if exist(fullfile('.','+mypil'),'dir')
    rmdir('+mypil','s')
end
mkdir +mypil
copyfile(fullfile(src_dir,'Launcher.m'), '+mypil');
copyfile(fullfile(src_dir,'TargetApplicationFramework.m'), '+mypil');
copyfile(fullfile(src_dir,'ConnectivityConfig.m'), '+mypil');
```

```
% Make the copied files writable.
fileattrib(fullfile('+mypil', '*'),'+w');

% Update the package name to reflect the new location of the files.
coder.mypil.Utils.UpdateClassName(...
    './+mypil/ConnectivityConfig.m',...
    'coder.mypil',...
    'mypil');
```

Register a target connectivity configuration using an `sl_customization.m` file. This example uses a supplied file.

```
sl_customization_path = fullfile(matlabroot,...
    'toolbox',...
    'rtw',...
    'rtwdemos',...
    'pil_demo');
addpath(sl_customization_path);
sl_refresh_customizations;
```

Specify the PIL simulation mode for the model.

```
close_system('rtwdemo_sil_topmodel')
open_system('rtwdemo_sil_topmodel')
set_param('rtwdemo_sil_topmodel','SimulationMode',...
          'processor-in-the-loop (pil)');
```

Specify the manufacturer and test hardware type. For example, PIL simulation on a 64-bit Windows® development computer requires:

```
set_param('rtwdemo_sil_topmodel','TargetHWDeviceType',...
          'Intel->x86-64 (Windows64)');
set_param('rtwdemo_sil_topmodel','TargetLongLongMode',true);
```

Run `piltest`.

```
piltest('rtwdemo_sil_topmodel', 'ConfigParam', {'ProdLongLongMode'} )
```

Input Arguments

`config` — Configuration set, configuration reference, or model

`ConfigSet` | `Simulink.ConfigSetRef` | character vector

A configuration set, configuration set reference, or Simulink model.

`additionalParameterList` — Additional parameters

cell array of character vectors

Extract additional parameters from `config` for PIL simulation.

`testName` — Specific test

`'all'` (default) | `'verifyPILBlock'` | `'verifyModelBlock'` | `'verifyTopModel'` | `'verifyExecutionOnTarget'` | `'verifyTopModelSILPILSwitching'` | `'verifyModelBlockSILPILSwitching'`

- `'verifyPILBlock'` — For normal mode results, run a simulation of a Simulink model with a subsystem. For PIL results, replace the subsystem with a PIL block and rerun the simulation. The function compares normal and PIL mode results. If the function detects differences, it produces an error.
- `'verifyModelBlock'` — For normal mode results, run simulations of a Simulink model with a Model block in normal mode.

  For PIL mode results, run simulation loops with the Model block in PIL mode. The function varies these settings:

  - Model block parameter **Code interface** — Set to `Top model` (standalone code interface) or `Model reference`.
  - **Configuration Parameters > Code Generation > Language** — Set to `C` or `C++`. For the `C++` case, the function sets **Code Generation > Interface > Code interface packaging** to `C++ class`.

  The function compares normal and PIL mode results. If the function detects differences, it produces an error.
- `'verifyTopModel'` — Run simulations of a Simulink top-model in normal and PIL modes. The function compares normal and PIL mode results. If the function detects differences, it produces an error.
- `'verifyExecutionOnTarget'` — Run simulations of a Simulink model with a Model block in normal and PIL modes. For each mode, the Model block uses standalone and model reference code interfaces. For PIL mode, the function introduces a deliberate mismatch. The function compares normal and PIL mode results. If it does not detect the deliberate mismatch, it produces an error.
- `'verifyTopModelSILPILSwitching'` — For a Simulink top model:

  - Verify that production code is not regenerated when the function switches between SIL and PIL simulation modes. The function compares timestamps of the production code in each mode.
  - Compares results from SIL and PIL mode simulations to results from a normal mode simulation.

  If the function detects differences in timestamps or simulation results, it produces an error.
- `'verifyModelBlockSILPILSwitching'` — For a Simulink Model block:

  - Verify that production code is not regenerated when the Model block simulation mode switches between SIL and PIL modes. The function compares timestamps of the production code in each mode.
  - Run simulation loops with the Model block in SIL and PIL modes. The function varies the **Code interface** Model block parameter, setting this parameter to `Top model` or `Model reference`. The function compares results from SIL and PIL mode simulations to results from a normal mode simulation.

  If the function detects differences in timestamps or simulation results, it produces an error.
- `'all'` — Run all tests from the test suite.

See Also

`ConfigSet` | `Simulink.ConfigSetRef`

Topics

"Create PIL Target Connectivity Configuration for Simulink"
"SIL and PIL Simulations"

Introduced in R2016b

piltest

Verify custom target connectivity configuration for MATLAB PIL execution

Syntax

```
piltest(config)
piltest(config,'ConfigParams',additionalParameterList)
piltest(config,'TestPoint',testName)
```

Description

`piltest(config)` runs tests that verify your custom processor-in-the-loop (PIL) target connectivity configuration. In the tests, the function runs the MATLAB function and performs PIL executions. The function compares results and produces errors if it detects differences. For PIL executions, the function extracts the `TargetHWDeviceType` and `Toolchain` settings from `config`.

In the current working folder, the function creates the `piltest` folder, which contains subfolders with test results.

`piltest(config,'ConfigParams',additionalParameterList)` extracts additional settings from `config` for the PIL execution.

`piltest(config,'TestPoint',testName)` runs the specified test.

Examples

Verify Target Connectivity Configuration with `piltest`

This example shows how you can use `piltest` to verify a target connectivity configuration for PIL execution.

Create a code generation configuration object for C/C++ static library generation.

```
cfg = coder.config('lib');
```

Create hardware configuration object, specify manufacturer and test hardware type, and assign handle to code generation object.

```
hwImpl = coder.HardwareImplementation;
hwImpl.TargetHWDeviceType = 'Atmel->AVR';
cfg.HardwareImplementation = hwImpl;
```

Specify the toolchain for code generation.

```
cfg.Toolchain = 'AVR tools for Arduino';
```

Run the function.

```
piltest(cfg)
```

Input Arguments

`config` — Configuration object
`coder.EmbeddedCodeConfig`

A configuration object that specifies code generation parameters.

`additionalParameterList` — Additional parameters
cell array of character vectors

Extract additional parameters from `config` for PIL execution.

`testName` — Specific test
`'all'` (default) | `'verifyPILConfig'`

- `'verifyPILConfig'` — For a given set of input values, the function:
  - Runs a MATLAB function on your development computer.
  - Performs PIL executions of generated MATLAB code on your target hardware with `config.TargetLang` set to `'C'` and `'C++'`.

  The function compares MATLAB function and PIL results. If the function detects differences, it produces an error.
- `'all'` — Run all tests.

See Also

Topics
"Create PIL Target Connectivity Configuration for MATLAB"
"PIL Execution of Code Generated for a Kalman Estimator"

Introduced in R2016b

Sections

Package: `coder.profile`

Get array of `coder.profile.ExecutionTimeSection` objects for profiled code sections

Syntax

```
NthSectionProfile=myExecutionProfile.Sections(N)
numberOfSections=length(myExecutionProfile.Sections)
```

Description

`NthSectionProfile=myExecutionProfile.Sections(N)` returns an `coder.profile.ExecutionTimeSection` object for the `N`th profiled code section.

`numberOfSections=length(myExecutionProfile.Sections)` returns the number of code sections for which profile data is available.

Examples

Get Nth Section Profile

Get the `coder.profile.ExecutionTimeSection` object for the `N`th profiled code section.

```
NthSectionProfile=myExecutionProfile.Sections(N);
```

Get Number of Profiled Code Sections

Get the number of code sections for which profile data is available.

```
numberOfSections=length(myExecutionProfile.Sections);
```

Input Arguments

`myExecutionProfile` — Workspace variable
workspace variable

`myExecutionProfile` is a workspace variable generated by a simulation.

Example: `myExecutionProfile`

`N` — Index of code section
index of code section

Index of code section for which profile data is required

Example: `N`

Output Arguments

`NthSectionProfile` — profile information object
profile information object

Object that contains profile information about the code section. You can use the following `coder.profile.ExecutionTimeSection` methods to retrieve the information:

- `Name` — Name of the code section.
- `Number` — Number of the code section.
- `NumCalls` — Number of calls to the code section.
- `TotalExecutionTimeInTicks` — Total number of timer ticks recorded for the code section over the entire simulation.
- `TurnaroundTimeInTicks` — Time between start and finish of the code section, in timer ticks.
- `TotalTurnaroundTimeInTicks` — Total number of timer ticks between start and finish of the code section, over the entire simulation.
- `MaximumExecutionTimeInTicks` — Maximum number of timer ticks for a single invocation of the code section.
- `MaximumExecutionTimeCallNum` — Number of call associated with the maximum number of timer ticks recorded for a single invocation of the code section.
- `MaximumTurnaroundTimeInTicks` — Maximum number of ticks between start and finish for a single invocation.
- `MaximumTurnaroundTimeCallNum` — Number of call associated with the maximum time between start and finish of a single invocation.
- `MaximumSelfTimeInTicks` — Maximum self time, in timer ticks.
- `SelfTimeInTicks` — Self time for the code section, in timer ticks.
- `TotalSelfTimeInTicks` — Total self time for the code section, over the entire simulation.
- `MaximumSelfTimeCallNum` — Call associated with maximum self time.
- `ExecutionTimeInTicks` — Vector of execution times.

`numberOfSections` — Number of code sections
number of code sections

The `numberOfSections` is the number of code sections with profile data.

See Also

`TimerTicksPerSecond` | `display` | `report`

Topics
"Code Execution Profiling with SIL and PIL"
"View and Compare Code Execution Times"
"Analyze Code Execution Data"

Introduced in R2012b

Sections

Package: `coder.profile.mc`

Get array of `coder.profile.ExecutionTimeSection` objects for profiled code sections (MATLAB code generation)

Syntax

```
NthSectionProfile = myExecutionProfile.Sections(N)
numberOfSections = length(myExecutionProfile.Sections)
```

Description

`NthSectionProfile = myExecutionProfile.Sections(N)` returns an `coder.profile.ExecutionTimeSection` object for the `N`th profiled code section.

`numberOfSections = length(myExecutionProfile.Sections)` returns the number of code sections for which profile data is available.

Examples

Get Nth Section Profile

Get the `coder.profile.ExecutionTimeSection` object for the `N`th profiled code section.

```
NthSectionProfile = myExecutionProfile.Sections(N);
```

Get Number of Profiled Code Sections

Get the number of code sections for which profile data is available.

```
numberOfSections = length(myExecutionProfile.Sections)
```

Input Arguments

`myExecutionProfile` — Workspace variable created by `getCoderExecutionProfile`
workspace variable

The `myExecutionProfile` is a workspace variable that you create by using the `getCoderExecutionProfile` function.

Example: `myExecutionProfile`

`N` — Index of code section
index of code section

Index of code section for which profile data is required.

Example: N

Output Arguments

`NthSectionProfile` — profile information object
profile information object

Object that contains profile information about the code section. You can use the following `coder.profile.ExecutionTimeSection` methods to retrieve the information:

- `Name` — Name of the code section.
- `Number` — Number of the code section.
- `NumCalls` — Number of calls to the code section.
- `TotalExecutionTimeInTicks` — Total number of timer ticks recorded for the code section over the entire simulation.
- `TurnaroundTimeInTicks` — Time between start and finish of the code section, in timer ticks.
- `TotalTurnaroundTimeInTicks` — Total number of timer ticks between start and finish of the code section, over the entire simulation.
- `MaximumExecutionTimeInTicks` — Maximum number of timer ticks for a single invocation of the code section.
- `MaximumExecutionTimeCallNum` — Number of call associated with the maximum number of timer ticks recorded for a single invocation of the code section.
- `MaximumTurnaroundTimeInTicks` — Maximum number of ticks between start and finish for a single invocation.
- `MaximumTurnaroundTimeCallNum` — Number of call associated with the maximum time between start and finish of a single invocation.
- `MaximumSelfTimeInTicks` — Maximum self time, in timer ticks.
- `SelfTimeInTicks` — Self time for the code section, in timer ticks.
- `TotalSelfTimeInTicks` — Total self time for the code section, over the entire simulation.
- `MaximumSelfTimeCallNum` — Call associated with maximum self time.
- `ExecutionTimeInTicks` — Vector of execution times.

`numberOfSections` — Number of code sections
number of code sections

Number of code sections with profile data

See Also

`TimerTicksPerSecond` | `getCoderExecutionProfile` | `report`

Topics
"Generate Execution Time Profile"
"Analyze Execution Time Data"

Introduced in R2012b

sharedCodeMATLABVersions

Manage MATLAB versions for cross-release code integration

Syntax

```
[registeredVersions, installationFolders] = sharedCodeMATLABVersions
sharedCodeMATLABVersions('Folder',versionInstallationFolder)
sharedCodeMATLABVersions('Remove', deregisterVersion)
```

Description

`[registeredVersions, installationFolders] = sharedCodeMATLABVersions` returns the available MATLAB versions and the installation folders.

`sharedCodeMATLABVersions('Folder',versionInstallationFolder)` registers a MATLAB version. The function specifies the folder where the MATLAB version is installed. The function checks that the folder corresponds to the `matlabroot` value for a valid installation, retrieves the MATLAB version number, and stores this information as a preference.

`sharedCodeMATLABVersions('Remove', deregisterVersion)` deregisters the MATLAB version and removes installation folder and version data.

Examples

Register Previous MATLAB Version for Cross-Release Code Integration

This code shows how you can register a previous release for your cross-release code integration workflow.

```
[registeredMATLABs, installationFolders] = sharedCodeMATLABVersions;
requiredVersion = 'R2017a';
typicalPath = 'C:\Program Files\MATLAB';

if isempty(registeredMATLABs) || ~any(strcmp(requiredVersion, registeredMATLABs))
  versionFolder = fullfile(typicalPath, requiredVersion);
  sharedCodeMATLABVersions('Folder', versionFolder);
end
```

Input Arguments

`versionInstallationFolder` — Installation folder location
character vector

Full path to the installation folder for the MATLAB version that you want to register.

Example: `'C:\Program Files\MATLAB\R2017a'`

`deregisterVersion` — Release version to deregister
character vector

MATLAB version that you want to deregister.

Example: `'R2017a'`

Output Arguments

`registeredVersions` — Registered release versions
cell array of character vectors

MATLAB release versions that are registered by the function.

`installationFolders` — Installation folder paths
cell array of character vectors

Installation folder locations for registered MATLAB versions.

See Also

`crossReleaseImport` | `sharedCodeUpdate`

Topics
“Cross-Release Shared Utility Code Reuse”
“Cross-Release Code Integration”

Introduced in R2017b

sharedCodeUpdate

Add new shared code source files to existing shared code folder

Syntax

```
sharedCodeUpdate(sourceFolder, destinationFolder)
sharedCodeUpdate(sourceFolder, destinationFolder, 'ExistingCodeSubfolder',
destinationSubfolder)
sharedCodeUpdate(buildFolder, destinationFolder)
sharedCodeUpdate(buildFolder, configurationSetOrModel)
sharedCodeUpdate(protectedModel, destinationFolder)
```

Description

`sharedCodeUpdate(sourceFolder, destinationFolder)` copies, for example, shared utility files from `sourceFolder` to a subfolder in `destinationFolder` provided that the files do not exist within `destinationFolder`. The function:

- Identifies files in both folders that have identical names but different content. The function does not overwrite these files in `destinationFolder`. In the Command Window, you see a `compare` link for each file. To examine differences by using the Comparison tool, click the link.
- Lists `sourceFolder` files that the function intends to copy and seeks confirmation. When you provide confirmation, the function copies the files to `destinationFolder`. By default, the destination of the copied files is a subfolder that corresponds to the release in which the files were created, for example, `R2015a` or `R2015b`.

`sharedCodeUpdate(sourceFolder, destinationFolder, 'ExistingCodeSubfolder', destinationSubfolder)` copies files to the subfolder that you specify.

`sharedCodeUpdate(buildFolder, destinationFolder)` copies shared code source files from the shared code location associated with `buildFolder`.

`sharedCodeUpdate(buildFolder, configurationSetOrModel)` copies shared code source files to the folder specified by the `'ExistingSharedCode'` parameter of a Simulink configuration set or model.

`sharedCodeUpdate(protectedModel, destinationFolder)` copies shared utility files for the protected model to the shared code folder.

Examples

Copy Shared Utility Files to Shared Code Folder

This example shows how to copy source files from a shared utilities folder to a shared code folder.

```
sourceFolder = fullfile(pwd,'R2015bWork','slprj','ert','_sharedutils');
existingSharedCodeFolder = fullfile(pwd,'SharedUtilCode');
sharedCodeUpdate(sourceFolder, existingSharedCodeFolder);
```

Copy Shared Utility Files to Subfolder

This example shows how to copy source files from a shared utilities folder to a specified subfolder in the shared code folder.

```
sourceFolder = fullfile(pwd,'R2015bWork','slprj','ert','_sharedutils');
existingSharedCodeFolder = fullfile(pwd, 'SharedUtilCode');
destinationSubfolder = 'mySub'
sharedCodeUpdate(sourceFolder, existingSharedCodeFolder,...
'ExistingCodeSubfolder', destinationSubfolder);
```

Copy Shared Utility Files From Relocated Code Folder

This example shows how to copy shared utility files from a relocated generated code folder to an existing shared code folder.

Specify path to shared code folder that you want to update.

```
pathToExistingSharedFolder = 'C:\mySharedCodeFolder';
```

Specify the full path to the relocated generated code folder `P1_ert_rtw`.

```
anchorFolder = 'C:\myWorkFolder';
relocatedCodeFolder = fullfile(anchorFolder, 'P1_ert_rtw');
```

Update the existing shared code folder.

```
sharedCodeUpdate(relocatedCodeFolder, pathToExistingSharedFolder);
```

Input Arguments

`sourceFolder` — Source folder
character vector

File path to folder with shared code files that you want to add to existing shared code folder.

`destinationFolder` — Existing shared code folder
character vector

File path to existing shared code folder.

`destinationSubfolder` — Destination subfolder
character vector

Destination subfolder in existing shared code folder.

`buildFolder` — Build folder
character vector

Path to a build folder that contains previously generated model code.

`configurationSetOrModel` — Configuration set or model
character vector

Simulink configuration set or model that uses an existing shared code folder specified by the `'ExistingSharedCode'` parameter.

`protectedModel` — Protected model file
character vector

File path for protected model. File name of protected model must have `.slxp` extension.

See Also

`crossReleaseImport`

Topics

"Cross-Release Shared Utility Code Reuse"
"Cross-Release Code Integration"
"Reference Protected Models from Third Parties"

Introduced in R2016b

getStatus

Return execution status

Syntax

```
status = cgvObj.getStatus()
status = cgvObj.getStatus(inputName)
```

Description

`status = cgvObj.getStatus()` returns the execution status of *`cgvObj`*. *`cgvObj`* is a handle to a `cgv.CGV` object.

`status = cgvObj.getStatus(inputName)` returns the status of a single execution for `inputName`.

Input Arguments

`inputName`

`inputName` is a unique numeric or character identifier associated with input data, which is added to the `cgv.CGV` object using `addInputData`.

Output Arguments

`status`

If `inputName` is provided, `status` is the result of the execution of input data associated with `inputName`.

Value	Description
`none`	Execution has not run.
`pending`	Execution is currently running.
`completed`	Execution ran to completion without errors and output data is available.
`passed`	Baseline data was provided. Execution ran to completion and comparison to the baseline data returned no differences.
`error`	Execution produced an error.
`failed`	Baseline data was provided. Execution ran to completion and comparison to the baseline data returned a difference.

If `inputName` is not provided, the following pseudocode describes the return status:

```
if (all executions return 'passed')
    status = 'passed'
```

```
else if (all executions return 'passed' or 'completed')
  status = 'completed'
else if (an execution returns 'error')
    status = 'error'
else if (an execution returns 'failed')
    status = 'failed'
else if (an execution returns 'none' or 'pending'
    status = 'none'
```

See Also

addBaseline | addInputData | run

Topics

"Verify Numerical Equivalence with CGV"

getTflArgFromString

Create code replacement argument based on specified name and built-in data type

Syntax

`arg = getTflArgFromString(hTable,name,datatype)`

Description

`arg = getTflArgFromString(hTable,name,datatype)` creates a code replacement argument that is based on a specified name and built-in or fixed-point data type.

The `IOType` property of the created argument defaults to `'RTW_IO_INPUT'`, indicating an input argument. For an output argument, change the `IOType` value to `'RTW_IO_OUTPUT'` by directly assigning the argument property.

This function does not support matrices. To create a matrix argument, use the argument class `RTW.TflArgMatrix` as shown in “Small Matrix Operation to Processor Code Replacement”, “Matrix Multiplication Operation to MathWorks BLAS Code Replacement”, and “Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement”.

Examples

Create and Add an Output Argument

This example shows how to use `getTflArgFromString` to create an `int16` output argument named `y1`. Then, the example adds the argument as a conceptual argument for a code replacement table entry.

```
hLib = RTW.TflTable;
op_entry = RTW.TflCOperationEntry;
.
.
.
arg = hLib.getTflArgFromString('y1', 'int16');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.addConceptualArg(arg);
```

Input Arguments

hTable — Handle to a code replacement table
handle

The *`hTable`* is a handle to a code replacement table previously returned by *`hTable`* `= RTW.TflTable`.

Example: `hLib`

name — Specifies the name to use for a code replacement argument
character vector | string scalar

Example: `'y1'`

`datatype` — Specifies a built-in data type or a fixed-point data type to use for the code replacement argument
`'integer'` | `'int8'` | `'int16'` | `'int32'` | `'long'` | `'long_long'` | `'uinteger'` | `'uint8'` | `'uint16'` | `'uint32'` | `'ulong'` | `'ulong_long'` | `'single'` | `'double'` | `'boolean'` | `'logical'`

You can specify fixed-point data types using the `fixdt` function from Fixed-Point Designer™ software; for example, `'fixdt(1,16,2)'`.

Example: `'integer'`

Output Arguments

`arg` — Handle to the created code replacement argument
handle

The *`arg`* is a handle to the created code replacement argument, which can be specified to the `addConceptualArg` function.

See Also

`addConceptualArg`

Topics
"Define Code Replacement Library Optimizations"
"Code You Can Replace from MATLAB Code"
"Code You Can Replace From Simulink Models"

Introduced in R2008a

getTflDWorkFromString

Create code replacement DWork argument for semaphore entry based on specified name and data type

Syntax

```
arg = getTflDWorkFromString(hTable,name,datatype)
```

Description

`arg = getTflDWorkFromString(hTable,name,datatype)` creates a code replacement DWork argument, based on a specified name and data type, for a semaphore entry in a code replacement table.

Examples

Create and Add a DWork Argument

This example shows how to use the `getTflDworkFromString` to create a `void*` argument named `d1`. Then, the example adds the argument as a DWork argument for a semaphore entry in a code replacement table.

```
hLib = RTW.TflTable;

% specify semaphore init function.
hEnt = RTW.TflCSemaphoreEntry;
hEnt.setTflCSemaphoreEntryParameters( ...
  'Key',                        'RTW_SEM_INIT', ...
  'Priority',                   30, ...
  'ImplementationName',         'mySemCreate', ...
  'ImplementationHeaderFile',   'mySem.h', ...
  'ImplementationSourceFile',   'mySem.c', ...
  'ImplementationHeaderPath',   LibPath, ...
  'ImplementationSourcePath',   LibPath, ...
  'GenCallback',                'RTW.copyFileToBuildDir', ...
  'SideEffects',                true);

% specify conceptual operands and result
arg = hLib.getTflArgFromString('y1', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
hEnt.addConceptualArg(arg);
arg = hLib.getTflArgFromString('u1', 'void');
hEnt.addConceptualArg(arg);

% specify replacement function signature
arg=hLib.getTflArgFromString('y1','void');
hEnt.Implementation.setReturn(arg);
arg.IOType = 'RTW_IO_OUTPUT';

% DWork Arg
arg = hLib.getTflDWorkFromString('d1','void*');
```

```
hEnt.addDWorkArg(arg);
addEntry(hLib, hEnt);
```

Input Arguments

hTable — Handle to a code replacement table
handle

The *hTable* is a handle to a code replacement table previously returned by *hTable* = RTW.TflTable.

Example: hLib

name — Specifies the name to use for the code replacement DWork argument
character vector | string scalar

Example: 'd1'

datatype — Specifies a data type to use for the code replacement DWork argument
character vector | string scalar

You must specify 'void*'.

Example: 'void*'

Output Arguments

arg — Handle to the created code replacement argument
arg

The *arg* is a handle to the created code replacement argument, which can be specified to the addDWorkArg function.

See Also

addDWorkArg

Topics
"Define Code Replacement Library Optimizations"

Introduced in R2013a

coder.hardware

Create hardware board configuration object for C/C++ code generation from MATLAB code

Description

The `coder.hardware` function creates a `coder.Hardware` object that contains hardware board parameters for C/C++ code generation from MATLAB code.

To use a `coder.Hardware` object for code generation, assign it to the `Hardware` property of a `coder.CodeConfig` or `coder.EmbeddedCodeConfig` object that you pass to `codegen`. Assigning a `coder.Hardware` object to the `Hardware` property customizes the associated `coder.HardwareImplementation` object and other configuration parameters for the particular hardware board.

Creation

Syntax

```
coder.hardware(boardname)
coder.hardware()
```

Description

`coder.hardware(boardname)` creates a `coder.Hardware` object for the specified hardware board. The board must be supported by an installed support package. To see a list of available boards, call `coder.hardware` without input parameters.

`coder.hardware()` returns a cell array of names of boards supported by installed support packages.

Input Arguments

`boardname` — hardware board name
character vector | string scalar

Hardware board name, specified as a character vector or a string scalar.

Example: `'Raspberry Pi'`

Example: `"Raspberry Pi"`

Properties

`Name` — Name of hardware board
character vector | string scalar

Name of hardware board, specified as a character vector or a string scalar. The `coder.hardware` function sets this property using the `boardname` argument.

`CPUClockRate` — Clock rate of hardware board
100 (default) | double scalar

Clock rate of hardware board, specified as a double scalar.

Examples

Generate Code for a Supported Hardware Board

Configure code generation for a Raspberry Pi board and generate code for a function `foo`.

```
hwlist = coder.hardware();
if ismember('Raspberry Pi',hwlist)
    hw = coder.hardware('Raspberry Pi');
    cfg = coder.config('lib');
    cfg.Hardware = hw;
    codegen foo -config cfg -report
end
```

Check Supported Hardware Boards

Before creating a `coder.Hardware` object for a hardware board, check that the board is supported by an installed support package.

List all boards for which a support package is installed.

```
hwlist = coder.hardware()
```

Test for an installed support package for a particular board.

```
hwlist = coder.hardware();
if ismember('Raspberry Pi',hwlist)
    hw = coder.hardware('Raspberry Pi');
end
```

Tips

- In addition to the `Name` and `CPUClockRate` properties, a `coder.Hardware` object has dynamic properties specific to the hardware board.
- To configure code generation parameters for processor-in-the-loop (PIL) execution on a supported hardware board, use `coder.hardware`. See “PIL Execution with ARM Cortex-A at the Command Line” and “PIL Execution with ARM Cortex-A by Using the MATLAB Coder App”. PIL execution requires Embedded Coder.

See Also
`codegen` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.HardwareImplementation`

Introduced in R2015b

RTW.TflBlasEntryGenerator

Package: RTW

Create code replacement table entry for a BLAS operation

Syntax

```
obj = RTW.TflBlasEntryGenerator
```

Description

`obj = RTW.TflBlasEntryGenerator` creates a handle, *`obj`*, to a code replacement table entry for a BLAS operator. The entry maps a conceptual representation of an operator to an implementation (replacement) representation.

Examples

Create Table Entry for BLAS Operator

This example shows how to create a code replacement table entry for a BLAS operator, `op_entry`.

```
hTable = RTW.TflTable;

arch = computer('arch');
compilerName = 'microsoft';
LibPath = fullfile('$(MATLAB_ROOT)', 'extern', ...
    'lib', arch, compilerName);

op_entry = RTW.TflBlasEntryGenerator;

libExt = 'lib';

setTflCOperationEntryParameters(op_entry, ...
    'Key',                       'RTW_OP_MUL', ...
    'Priority',                  100, ...
    'ImplementationName',        'dgemm32', ...
    'ImplementationHeaderFile', 'blascompat32_crl.h', ...
    'ImplementationHeaderPath', fullfile('$(MATLAB_ROOT)','extern','include'), ...
    'AdditionalLinkObjs',        {['libmwblascompat32.' libExt]}, ...
    'AdditionalLinkObjsPaths',  {LibPath}, ...
    'SideEffects',               true);
```

Output Arguments

obj — Handle to code replacement table entry for a BLAS operator
handle

The *`obj`* is a handle to the created code replacement table entry for a BLAS operator.

See Also

`RTW.TflCBlasEntryGenerator` | `RTW.TflCOperationEntry` | `RTW.TflTable`

Topics

"Define Code Replacement Library Optimizations"
"Matrix Multiplication Operation to MathWorks BLAS Code Replacement"
"Code You Can Replace from MATLAB Code"
"Code You Can Replace From Simulink Models"

Introduced in R2010a

RTW.TflCBlasEntryGenerator

Package: RTW

Create code replacement table entry for a CBLAS operation

Syntax

`obj = RTW.TflCBlasEntryGenerator`

Description

`obj = RTW.TflCBlasEntryGenerator` creates a handle, *`obj`*, to a code replacement table entry for a CBLAS operator. The entry maps a conceptual representation of an operator to an implementation (replacement) representation.

Examples

Create Table Entry for CBLAS Operator

This example shows how to create a code replacement table entry for a CBLAS operator, `hEnt`.

```
hTable = RTW.TflTable;

arch = computer('arch');
compilerName = 'my_compiler';
LibPath = fullfile('$(MATLAB_ROOT)', 'extern', ...
    'lib', arch, compilerName);

op_entry = RTW.TflCBlasEntryGenerator;

libExt = 'lib';

setTflCOperationEntryParameters(op_entry, ...
    'Key',                        'RTW_OP_MUL', ...
    'Priority',                   100, ...
    'ImplementationName',         'dgemm32', ...
    'ImplementationHeaderFile', 'my_cblas_compatible_crl.h', ...
    'ImplementationHeaderPath', fullfile('$(MATLAB_ROOT)','extern','include'), ...
    'AdditionalLinkObjs',         {['my_lib_cblas_compatible.' libExt]}, ...
    'AdditionalLinkObjsPaths',  {LibPath}, ...
    'SideEffects',                true);
```

Output Arguments

obj — Handle to code replacement table entry for a CBLAS operator
handle

The *`obj`* is a handle to the created code replacement table entry for a CBLAS operator.

See Also

`RTW.TflBlasEntryGenerator` | `RTW.TflCOperationEntry` | `RTW.TflTable`

Topics

"Define Code Replacement Library Optimizations"
"Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement"
"Code You Can Replace from MATLAB Code"
"Code You Can Replace From Simulink Models"

Introduced in R2010a

RTW.TflCFunctionEntry

Package: RTW

Create code replacement table entry for a function

Syntax

```
obj = RTW.TflCFunctionEntry
```

Description

`obj = RTW.TflCFunctionEntry` creates a handle, *`obj`*, to a code replacement table entry for a function. The entry maps a conceptual representation of a function to an implementation (replacement) representation.

Examples

Create Table Entry for Function

This example shows how to create a code replacement table entry for a function, `hEnt`.

```
hEnt = RTW.TflCFunctionEntry;
```

Output Arguments

obj — Handle to code replacement table entry for a function
handle

The *`obj`* is a handle to the created code replacement table entry for a function.

See Also

`RTW.TflCFunctionEntryML` | `RTW.TflTable`

Topics

"Define Code Replacement Library Optimizations"
"Math Function Code Replacement"
"Memory Function Code Replacement"
"Nonfinite Function Code Replacement"
"Lookup Table Function Code Replacement"
"Code You Can Replace from MATLAB Code"
"Code You Can Replace From Simulink Models"

Introduced in R2007b

RTW.TflCFunctionEntryML

Base class for custom code replacement table function entry

Syntax

`RTW.TflCFunctionEntryML`

Description

Derive a class from `RTW.TflCFunctionEntryML` to represent your custom function entry.

Examples

"Customize Match and Replacement Process"

See Also

`RTW.TflCFunctionEntry` | `RTW.TflTable`

Topics

"Define Code Replacement Library Optimizations"
"Customize Match and Replacement Process"
"Code You Can Replace from MATLAB Code"
"Code You Can Replace From Simulink Models"

RTW.TflCOperationEntry

Package: RTW

Create code replacement table entry for an operator

Syntax

```
obj = RTW.TflCOperationEntry
```

Description

`obj = RTW.TflCOperationEntry` creates a handle, *obj*, to a code replacement table entry for an operator. The entry maps a conceptual representation of an operator to an implementation (replacement) representation.

Examples

Create Table Entry for Operator

This example shows how to create a code replacement table entry for an operator, `hEnt`.

```
hEnt = RTW.TflCOperationEntry;
```

Output Arguments

obj — Handle to code replacement table entry for an operator
handle

The *obj* is a handle to the created code replacement table entry for an operator.

See Also

`RTW.TflCOperationEntryGenerator` | `RTW.TflCOperationEntryGenerator_NetSlope` | `RTW.TflCOperationEntryML` | `RTW.TflTable`

Topics
"Define Code Replacement Library Optimizations"
"Scalar Operator Code Replacement"
"Addition and Subtraction Operator Code Replacement"
"Small Matrix Operation to Processor Code Replacement"
"Code You Can Replace from MATLAB Code"
"Code You Can Replace From Simulink Models"

Introduced in R2007b

RTW.TflCOperationEntryGenerator

Package: `RTW`

Create code replacement table entry for a fixed-point addition or subtraction operation

Syntax

`obj = RTW.TflCOperationEntryGenerator`

Description

`obj = RTW.TflCOperationEntryGenerator` creates a handle, *`obj`*, to a code replacement table entry for a fixed-point addition or subtraction operation. The entry maps a conceptual representation of an operator to an implementation (replacement) representation.

Examples

Create Table Entry for Fixed-Point Add or Subtract Operation

This example shows how to create a code replacement table entry for a fixed-point addition or subtraction operation, `hEnt`.

```
hEnt = RTW.TflCOperationEntryGenerator;
```

Output Arguments

`obj` — Handle to code replacement table entry for a fixed-point addition or subtraction operation
handle

The *`obj`* is a handle to the created code replacement table entry for a fixed-point addition or subtraction operation.

See Also

`RTW.TflCOperationEntry` | `RTW.TflCOperationEntryGenerator_NetSlope` | `RTW.TflCOperationEntryML` | `RTW.TflTable`

Topics

"Define Code Replacement Library Optimizations"
"Fixed-Point Operator Code Replacement"
"Binary-Point-Only Scaling Code Replacement"
"Slope Bias Scaling Code Replacement"
"Code You Can Replace from MATLAB Code"
"Code You Can Replace From Simulink Models"

Introduced in R2008a

RTW.TflCOperationEntryGenerator_NetSlope

Package: RTW

Create code replacement table entry for a net slope fixed-point operation

Syntax

```
obj = RTW.TflCOperationEntryGenerator_NetSlope
```

Description

`obj = RTW.TflCOperationEntryGenerator_NetSlope` creates a handle, *obj*, to a code replacement table entry for a net slope fixed-point operation. The entry maps a conceptual representation of an operator to an implementation (replacement) representation.

Examples

Create Table Entry for Net Slope Fixed-Point Operation

This example shows how to create a code replacement table entry for a net slope fixed-point operation, `hEnt`.

```
hEnt = RTW.TflCOperationEntryGenerator_NetSlope;
```

Output Arguments

obj — Handle to code replacement table entry for a net slope fixed-point operation
handle

The *obj* is a handle to the created code replacement table entry for a net slope fixed-point operation.

See Also

RTW.TflCOperationEntry | RTW.TflCOperationEntryGenerator | RTW.TflCOperationEntryML

Topics
“Define Code Replacement Library Optimizations”
“Fixed-Point Operator Code Replacement”
“Net Slope Scaling Code Replacement”
“Equal Slope and Zero Net Bias Code Replacement”
“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”

Introduced in R2008b

RTW.TflCOperationEntryML

Base class for custom code replacement table operator entry

Syntax

`RTW.TflCOperationEntryML`

Description

Derive a class from `RTW.TflCOperationEntryML` to represent your custom operator entry.

Examples

"Customize Code Match and Replacement for Scalar Operations"

See Also

`RTW.TflCOperationEntry` | `RTW.TflTable`

Topics

"Define Code Replacement Library Optimizations"
"Customize Match and Replacement Process"
"Code You Can Replace from MATLAB Code"
"Code You Can Replace From Simulink Models"

RTW.TflCSemaphoreEntry

Package: RTW

Create code replacement table entry for a semaphore or mutex

Syntax

```
obj = RTW.TflCSemaphoreEntry
```

Description

`obj = RTW.TflCSemaphoreEntry` creates a handle, *`obj`*, to a code replacement table entry for a semaphore or mutex. The entry maps a conceptual representation of a semaphore or mutex to an implementation (replacement) representation.

Examples

Create Table Entry for Semaphore or Mutex

This example shows how to create a code replacement table entry for a semaphore or mutex, `hEnt`.

```
hEnt = RTW.TflCSemaphoreEntry;
```

Output Arguments

`obj` — Handle to code replacement table entry for a semaphore or mutex
handle

The *`obj`* is a handle to the created code replacement table entry for a semaphore or mutex.

See Also

Topics
"Define Code Replacement Library Optimizations"
"Code You Can Replace from MATLAB Code"
"Code You Can Replace From Simulink Models"

Introduced in R2010a

RTW.TflTable

Package: RTW

Create code replacement table

Syntax

`obj = RTW.TflTable`

Description

`obj = RTW.TflTable` creates a handle, *`obj`*, to a code replacement table.

Examples

Create a Code Replacement Table

This example shows how to create a code replacement table object, `hTable`.

```
hTable = RTW.TflTable;
```

Output Arguments

`obj` — Handle to code replacement table
handle

The *`obj`* is a handle to the created code replacement table.

See Also

`addEntry` | `createCRLEntry`

Topics

"Define Code Replacement Library Optimizations"
"Code You Can Replace from MATLAB Code"
"Code You Can Replace From Simulink Models"

Introduced in R2007b

schedule

Package: `coder.profile`

Visualize task scheduling

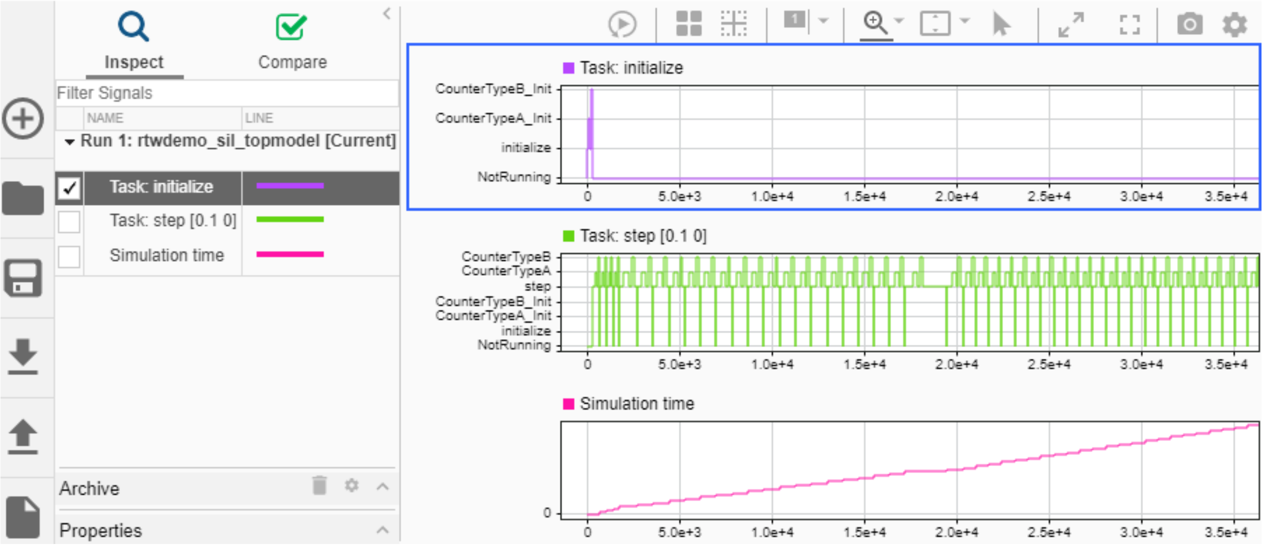
Syntax

```
schedule(executionProfile)
schedule(executionProfile,Name,Value)
```

Description

`schedule(executionProfile)`, using the Simulation Data Inspector, helps you to visualize how code was executed on the target hardware in the last software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation.

`schedule(executionProfile,Name,Value)` uses name-value arguments to control the display of function execution and simulation time.



Examples

Visualize Task Scheduling

Run a simulation with a model that is configured to generate a workspace variable with execution-time measurements.

```
rtwdemo_sil_topmodel;
set_param('rtwdemo_sil_topmodel',...
          'CodeExecutionProfiling', 'on');
set_param('rtwdemo_sil_topmodel',...
```

```
          'SimulationMode', 'software-in-the-loop (SIL)');
set_param('rtwdemo_sil_topmodel',...
          'CodeProfilingInstrumentation', 'Detailed');
set_param('rtwdemo_sil_topmodel',...
          'CodeProfilingSaveOptions', 'AllData');
simOut = sim('rtwdemo_sil_topmodel');
```

The simulation generates the variable `executionProfile` (default) in the object `simOut`.

At the end of the simulation, open a code execution report.

```
report(simOut.executionProfile)
```

Under **Profiled Sections of Code**, in the **Section** column, expand all nodes. You see profile information for six code sections. For example, the task `step [0.1 0]` and functions `CounterTypeA` and `CounterTypeB`.

2. Profiled Sections of Code

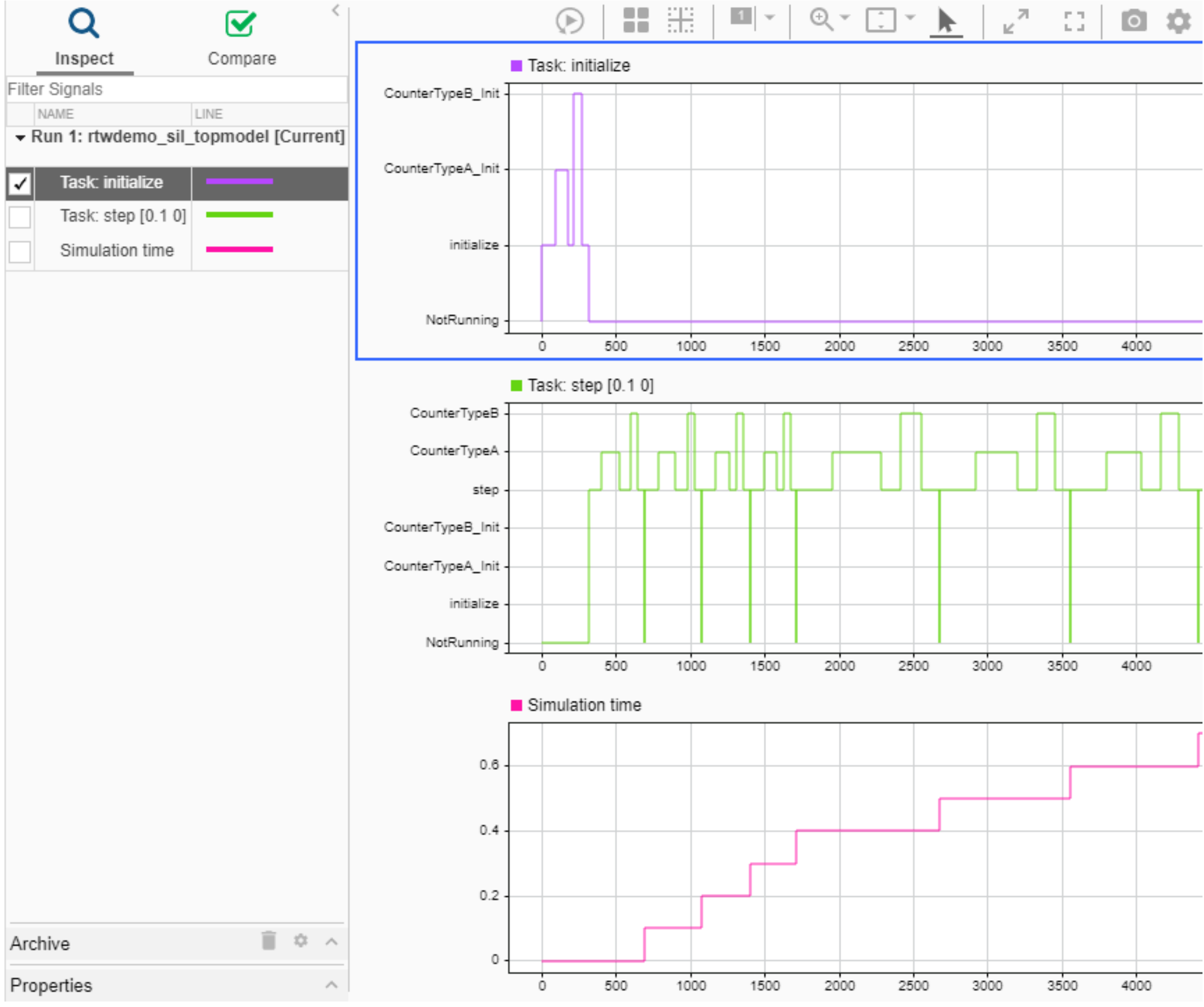
Section	Maximum Execution Time in ns	Average Execution Time in ns	Maximum Self Time in ns	Average Self Time in ns	Calls	
[-] initialize	241	241	148	148	1	
CounterTypeA	61	61	61	61	1	
CounterTypeB	33	33	33	33	1	
[-] step [0.1 0]	409	213	221	119	101	
CounterTypeA	109	58	109	58	101	
CounterTypeB	79	36	79	36	101	

To visualize how the tasks are scheduled and generated code is executed, run:

```
schedule(simOut.executionProfile)
```

Or, from the **SIL/PIL** tab, open the **Results** gallery. Under **Execution Profiling Results**, click **Generate Schedule**.

The Simulation Data Inspector displays task and simulation time plots.



In each task plot, the Y-axis lists tasks and functions called by each task. From the plots, you can infer the following information:

- The order in which tasks run. For example, `initialize` runs before `step`.

  If the model is multi-rate, you can see how Simulink schedules the different rates (a task for each rate).
- The time that is required to execute a task or a function, computed as the difference between the stop and start times. For example, observe that `CounterTypeB` takes less time to run than `CounterTypeA`. When a task is not running, the Y-axis value of the plot is `NotRunning`.
- The order in which functions run within a task. For example, in the initialize task, `counterTypeA_Init` function runs before `counterTypeB_Init` function. If function calls are nested, you can see the execution order of the functions.
- The last plot shows the simulation time when the tasks and functions are executed.

Input Arguments

`executionProfile` — Variable with profiling data
object

Variable specified by the **Workspace variable** configuration parameter, which contains the code execution profiling data. The SIL or PIL simulation generates the variable.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `schedule(simOut.executionProfile, 'ShowTasksOnly', false, 'StartSimTime', 0.1, 'StopSimTime', 5.7)`

`ShowTasksOnly` — Turn off function execution display
false (default) | true

Control display of function execution plots:

- `true` –– Display task execution only. Do not display function execution.
- `false` –– Display task and function execution.

`MaxNumPoints` — Maximum number of display points
integer

Specify maximum number of points to display.

`StartSimTime` — Simulation time at start of display
float

Specify simulation time at the start of the display.

`StopSimTime` — Simulation time at end of display
float

Specify simulation time at the end of the display.

See Also
`report` | `timeline`

Topics
“Code Execution Profiling with SIL and PIL”
“View and Compare Code Execution Times”

Introduced in R2021a

Time

Get simulation time for code section

Syntax

```
SimTime = NthSectionProfile.Time
```

Description

`SimTime = NthSectionProfile.Time` returns a simulation time vector that corresponds to the execution time measurements for the code section.

Examples

Get Simulation Time for Code Section

Run a simulation with a model that is configured to generate a workspace variable with execution time measurements.

```
rtwdemo_sil_topmodel;
set_param('rtwdemo_sil_topmodel',...
          'CodeExecutionProfiling', 'on');
set_param('rtwdemo_sil_topmodel',...
          'SimulationMode', 'software-in-the-loop (SIL)');
set_param('rtwdemo_sil_topmodel',...
          'CodeProfilingInstrumentation', 'on');
set_param('rtwdemo_sil_topmodel',...
          'CodeProfilingSaveOptions', 'AllData');
sim('rtwdemo_sil_topmodel');
```

The simulation generates the workspace variable `executionProfile` (default).

At the end of the simulation, get profile for the seventh code section.

```
seventhSectionProfile = executionProfile.Sections(7);
```

Get vector representing simulation time for code section.

```
simulationTimeVector = seventhSectionProfile.Time;
```

Input Arguments

`NthSectionProfile` — `coder.profile.ExecutionTimeSection`
object

Object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

`SimTime` — Simulation time
double

Simulation time, in seconds, for section of code. Returned as a vector.

See Also

`ExecutionTimeInSeconds` | `ExecutionTimeInTicks` | `Sections`

Topics
"Code Execution Profiling with SIL and PIL"
"Analyze Code Execution Data"

Introduced in R2013a

Time

Package: `coder.profile.mc`

Time over which code section execution time measurements are made (MATLAB code generation)

Syntax

`Time = NthSectionProfile.Time`

Description

`Time = NthSectionProfile.Time` returns a time vector corresponding to the period over which execution times are measured for the code section.

Examples

Get Time Vector for Code Section

Copy MATLAB code to your working folder.

```
src_dir = ...
    fullfile(docroot,'toolbox','coder','examples','kalman');

copyfile(fullfile(src_dir,'kalman01.m'), '.')
copyfile(fullfile(src_dir,'test01_ui.m'), '.')
copyfile(fullfile(src_dir,'plot_trajectory.m'), '.')
copyfile(fullfile(src_dir,'position.mat'), '.')
```

Set up and run a SIL execution.

```
config = coder.config('lib');
config.GenerateReport = true;

config.VerificationMode = 'SIL';
config.CodeExecutionProfiling = true;

codegen('-config', config, '-args', {zeros(2,1)}, 'kalman01');

coder.runTest('test01_ui', ['kalman01_sil.' mexext]);
```

At end of the execution, you see the following message.

```
To terminate execution: clear kalman01_sil
Execution profiling report available after termination.
```

Click the link `clear kalman01_sil`.

```
### Stopping SIL execution for 'kalman01'
    Execution profiling report: report(getCoderExecutionProfile('kalman01'))
```

Create a workspace variable that holds execution time data.

```
executionProfile=getCoderExecutionProfile('kalman01');
```

Get the profile for the second code section.

```
secondSectionProfile = executionProfile.Sections(2);
```

Get time vector for code section.

```
time = secondSectionProfile.Time;
```

Input Arguments

NthSectionProfile — coder.profile.ExecutionTimeSection
object

Object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

Time — Time
double

Time, in seconds, over which measurements are made for code section. Returned as a vector.

See Also

`ExecutionTimeInSeconds` | `ExecutionTimeInTicks` | `Sections` | `getCoderExecutionProfile`

Topics

"Generate Execution Time Profile"
"Analyze Execution Time Data"

Introduced in R2013a

timeline

Package: `coder.profile`

Display invocations of code sections over execution timeline

Syntax

```
timeline(executionProfile)
timeline(executionProfile, 'MaxResizeIncrement', numberOfPoints)
```

Description

`timeline(executionProfile)` displays invocations of each profiled code section over the execution timeline.

Note You can use the `schedule` function to visualize, through the Simulation Data Inspector, task scheduling and the order of function calls.

`timeline(executionProfile, 'MaxResizeIncrement', numberOfPoints)` specifies the maximum increment by which you:

- Increase the number of displayed points when you click the zoom-out tool.
- Move along the `timeline` plot when you sweep right or left with the pan tool.

Use this command when you want to review large `timeline` plots quickly.

Examples

Display Code Section Invocations

Run a simulation with a model that is configured to generate a workspace variable with execution-time measurements.

```
rtwdemo_sil_topmodel;
set_param('rtwdemo_sil_topmodel',...
          'CodeExecutionProfiling', 'on');
set_param('rtwdemo_sil_topmodel',...
          'SimulationMode', 'software-in-the-loop (SIL)');
set_param('rtwdemo_sil_topmodel',...
          'CodeProfilingInstrumentation', 'Detailed');
set_param('rtwdemo_sil_topmodel',...
          'CodeProfilingSaveOptions', 'AllData');
sim('rtwdemo_sil_topmodel');
```

The simulation generates the workspace variable `executionProfile` (default).

At the end of the simulation, open a code execution report.

```
report(executionProfile)
```

Under **Profiled Sections of Code**, in the **Model** column, expand all nodes. You see profile information for eight code sections. For example, the task `rtwdemo_sil_topmodel_step` and functions `CounterTypeA` and `CounterTypeB`.

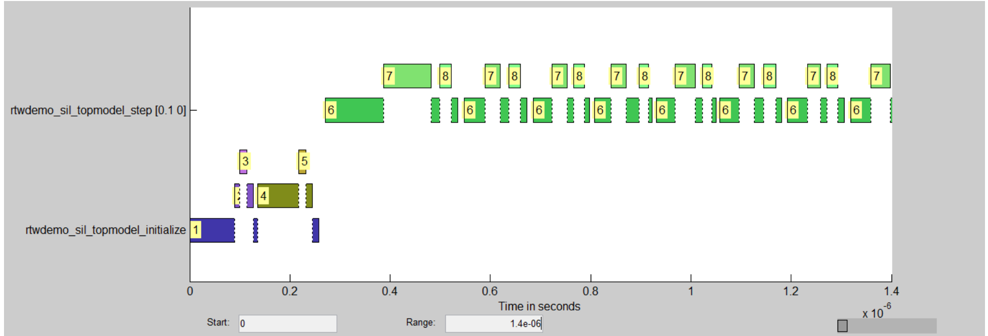
2. Profiled Sections of Code

Model	Maximum Execution Time	Average Execution Time	Maximum Self Time	Average Self Time	Calls	
[−] rtwdemo_sil_topmodel_initialize	257	257	111	111	1	
[−] CounterTypeA	38	38	23	23	1	
CounterTypeA	15	15	15	15	1	
[−] CounterTypeB	109	109	94	94	1	
CounterTypeB	15	15	15	15	1	
[−] rtwdemo_sil_topmodel_step [0.1 0]	265	121	147	61	101	
CounterTypeA	94	36	94	36	101	
CounterTypeB	37	24	37	24	101	

Display code section invocations.

```
timeline(executionProfile)
```

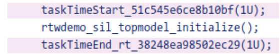
In the Execution Profile window, you see numbered horizontal bars that represent invocations of the code sections.



For example, the blue bars show when the first section, `rtwdemo_sil_topmodel_initialize`, is invoked.

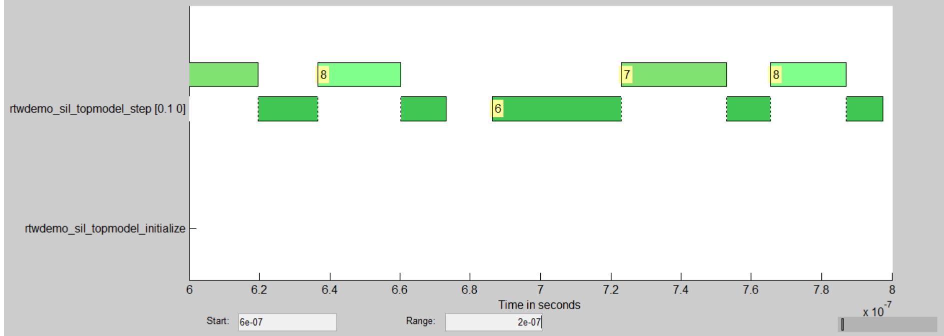
To see the first code section, in the first row of the Code Execution Profiling Report, click the icon .

The Code Generation Report displays the function call.



```
taskTimeStart_51c545e6ce8b10bf(1U);
rtwdemo_sil_topmodel_initialize();
taskTimeEnd_rt_38248ea98502ec29(1U);
```

To see what code sections are invoked over a specific time period, use the **Start** and **Range** fields of the Execution Profile window. For example, in the **Start** and **Range** fields, enter `6e-07` and `2e-07` respectively. Then press **Enter**.



Between 0.6 µs and 0.8 µs, you see that the task `rtwdemo_sil_topmodel_step` (code section 6) and the functions `CounterTypeA` (code section 7) and `CounterTypeB` (code section 8) are invoked.

On the bottom right of the Execution Profile window, the indicator shows what portion of the execution timeline is being displayed.

Input Arguments

`executionProfile` — `coder.profile.ExecutionTime`
object

When you run a simulation with code execution profiling, the software generates `executionProfile` as a workspace variable.

`numberOfPoints` — Number of points
20 (default) | integer

Maximum increment for zoom-out and pan tools.

See Also

`report` | `schedule`

Topics

"Code Execution Profiling with SIL and PIL"
"View and Compare Code Execution Times"

Introduced in R2013b

TimerTicksPerSecond

Package: `coder.profile`

Get and set number of timer ticks per second

Syntax

```
timerTicksPerSecVal = myExecutionProfile.TimerTicksPerSecond
myExecutionProfile.TimerTicksPerSecond = timerTicksPerSecVal
```

Description

`timerTicksPerSecVal = myExecutionProfile.TimerTicksPerSecond` returns the number of timer ticks per second. For example, if the timer runs at 1 MHz, then the number of ticks per second is 10^6. You can calculate the execution time in seconds using the formula *ExecutionTimeInSecs* = *ExecutionTimeInTicks*/*TimerTicksPerSecond*.

`myExecutionProfile.TimerTicksPerSecond = timerTicksPerSecVal` sets the number of timer ticks per second. Use this method if the “Create PIL Target Connectivity Configuration for Simulink” does not specify this value.

Examples

Get Timer Ticks per Second Value

To get the number of timer ticks per second, get the `TimerTicksPerSecond` property value from the `myExecutionProfile` workspace variable.

```
timerTicksPerSecVal = myExecutionProfile.TimerTicksPerSecond;
```

Set Timer Ticks per Second Value

To set the number of timer ticks per second, set the `TimerTicksPerSecond` property value in the `myExecutionProfile` workspace variable.

```
myExecutionProfile.TimerTicksPerSecond = timerTicksPerSecVal;
```

Input Arguments

`myExecutionProfile` — Workspace variable
workspace variable

`myExecutionProfile` is a workspace variable generated by a simulation.

Example: `myExecutionProfile`

`timerTicksPerSecVal` — Number of timer ticks
number of timer ticks

Number of timer ticks per second

Example: `timerTicksPerSecVal`

Output Arguments

`timerTicksPerSecVal` — Number of timer ticks
number of timer ticks

Number of timer ticks per second

See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `display` | `report`

Topics
"Code Execution Profiling with SIL and PIL"
"View and Compare Code Execution Times"
"Analyze Code Execution Data"

Introduced in R2012b

TimerTicksPerSecond

Package: `coder.profile.mc`

Get and set number of timer ticks per second (MATLAB code generation)

Syntax

```
timerTicksPerSecVal = myExecutionProfile.TimerTicksPerSecond
myExecutionProfile.TimerTicksPerSecond = timerTicksPerSecVal
```

Description

`timerTicksPerSecVal = myExecutionProfile.TimerTicksPerSecond` returns the number of timer ticks per second. For example, if the timer runs at 1 MHz, then the number of ticks per second is 10^6.

You can calculate the execution time in seconds using the formula *ExecutionTimeInSecs* = *ExecutionTimeInTicks*/*TimerTicksPerSecond*.

`myExecutionProfile.TimerTicksPerSecond = timerTicksPerSecVal` sets the number of timer ticks per second. Use this method if the target connectivity configuration does not specify this value.

Examples

Get Timer Ticks per Second Value

To get the number of timer ticks per second, get the `TimerTicksPerSecond` property value from the `myExecutionProfile` workspace variable.

```
timerTicksPerSecVal = myExecutionProfile.TimerTicksPerSecond;
```

Set Timer Ticks per Second Value

To set the number of timer ticks per second, set the `TimerTicksPerSecond` property value in the `myExecutionProfile` workspace variable.

```
myExecutionProfile.TimerTicksPerSecond = timerTicksPerSecVal;
```

Input Arguments

`myExecutionProfile` — Workspace variable that contains the execution-time profile
workspace variable

The `myExecutionProfile` is a workspace variable that contains the execution-time profile of the code generated from your MATLAB function.

`timerTicksPerSecVal` — Number of timer ticks
number of timer ticks

Number of timer ticks per second

Example: `timerTicksPerSecVal`

Output Arguments

`timerTicksPerSecVal` — Number of timer ticks
number of timer ticks

Number of timer ticks per second

See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `getCoderExecutionProfile` | `report`

Topics
"Generate Execution Time Profile"
"Analyze Execution Time Data"
"Create PIL Target Connectivity Configuration for MATLAB"

Introduced in R2012b

coder.MATLABCodeTemplate.getTokenValue

Class: `coder.MATLABCodeTemplate`
Package: `coder`

Get value of token

Syntax

`tokenValue = getTokenValue(tokenName)`

Description

`tokenValue = getTokenValue(tokenName)` returns the value of the specified token.

Input Arguments

`tokenName`

Name of token

Default: empty

Output Arguments

`tokenValue` — Token value
character vector

The current value of `tokenName`, returned as a character vector.

Examples

Create a MATLABCodeTemplate object with the default template, then get the value for a token.

```
newObj = coder.MATLABCodeTemplate;
% Creates a MATLABCodeTemplate object from the default template
newObj.getCurrentTokens()
% Get list of current tokens
newObj.getTokenValue('MATLABCoderVersion')
% Check value of a token
```

See Also

`coder.MATLABCodeTemplate.emitSection` | `coder.MATLABCodeTemplate.getCurrentTokens` | `coder.MATLABCodeTemplate.setTokenValue`

Topics

"Generate Custom File and Function Banners for C/C++ Code"
"Code Generation Template Files for MATLAB Code"

ExecutionTimeInSeconds

Get execution time in seconds for profiled section of code

Syntax

`ExecutionTimes = NthSectionProfile.ExecutionTimeInSeconds`

Description

`ExecutionTimes = NthSectionProfile.ExecutionTimeInSeconds` returns a vector of execution times, measured in seconds, for the profiled section of code. Each element of `ExecutionTimes` contains the difference between the timer reading at the start and the end of the section.

If you set the `CodeProfilingSaveOptions` parameter to `'SummaryOnly'`, `NthSectionProfile.ExecutionTimeInSeconds` returns an empty array. To change that parameter, open the Configuration Parameters dialog box by pressing **Ctrl+E**, open the **Verification** pane under **Code Generation**, and change the **Save options** parameter to `All data`.

Examples

Get Execution Times for Code Section

Run a simulation with a model that is configured to generate a workspace variable with execution time measurements.

```
rtwdemo_sil_topmodel;
set_param('rtwdemo_sil_topmodel', 'CodeExecutionProfiling', 'on');
set_param('rtwdemo_sil_topmodel', 'SimulationMode', 'software-in-the-loop (SIL)');
set_param('rtwdemo_sil_topmodel', 'CodeProfilingInstrumentation', 'on');
set_param('rtwdemo_sil_topmodel', 'CodeProfilingSaveOptions', 'AllData');
sim('rtwdemo_sil_topmodel');
```

The simulation generates the workspace variable `executionProfile` (default).

At the end of the simulation, get the profile for the seventh code section.

```
SeventhSectionProfile = executionProfile.Sections(7);
```

Get vector of execution times for the code section.

```
time_vector = SeventhSectionProfile.ExecutionTimeInSeconds;
```

Input Arguments

`NthSectionProfile` — `coder.profile.ExecutionTimeSection`
object

Object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

`ExecutionTimes` — Execution time measurements
double

Execution times, in seconds, for section of code. Returned as a vector.

See Also

`ExecutionTimeInTicks` | `Sections`

Topics
"Code Execution Profiling with SIL and PIL"
"Analyze Code Execution Data"

Introduced in R2013a

ExecutionTimeInSeconds

Package: `coder.profile.mc`

Get execution time in seconds for profiled section of code (MATLAB code generation)

Syntax

```
ExecutionTimes = NthSectionProfile.ExecutionTimeInSeconds
```

Description

`ExecutionTimes = NthSectionProfile.ExecutionTimeInSeconds` returns a vector of execution times, measured in seconds, for the profiled section of code. Each element of `ExecutionTimes` contains the difference between the timer reading at the start and the end of the section.

Examples

Get Execution Times for Code Section

Copy MATLAB code to your working folder.

```
src_dir = ...
    fullfile(docroot,'toolbox','coder','examples','kalman');

copyfile(fullfile(src_dir,'kalman01.m'), '.')
copyfile(fullfile(src_dir,'test01_ui.m'), '.')
copyfile(fullfile(src_dir,'plot_trajectory.m'), '.')
copyfile(fullfile(src_dir,'position.mat'), '.')
```

Set up and run a SIL execution.

```
config = coder.config('lib');
config.GenerateReport = true;

config.VerificationMode = 'SIL';
config.CodeExecutionProfiling = true;

codegen('-config', config, '-args', {zeros(2,1)}, 'kalman01');

coder.runTest('test01_ui', ['kalman01_sil.' mexext]);
```

At end of the execution, you see the following message.

```
To terminate execution: clear kalman01_sil
Execution profiling report available after termination.
```

Click the link `clear kalman01_sil`.

```
### Stopping SIL execution for 'kalman01'
    Execution profiling report: report(getCoderExecutionProfile('kalman01'))
```

Create a workspace variable that holds execution time data.

```
executionProfile=getCoderExecutionProfile('kalman01');
```

Get the profile for the second code section.

```
SecondSectionProfile = executionProfile.Sections(2);
```

Get vector of execution times for the code section.

```
time_vector = SecondSectionProfile.ExecutionTimeInSeconds;
```

Input Arguments

`NthSectionProfile` — `coder.profile.ExecutionTimeSection`
object

Object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

`ExecutionTimes` — Execution time measurements
double

Execution times, in seconds, for section of code. Returned as a vector.

See Also

`ExecutionTimeInTicks` | `Sections` | `getCoderExecutionProfile`

Topics

"Generate Execution Time Profile"
"Analyze Execution Time Data"

Introduced in R2013a

ExecutionTimeInTicks

Package: `coder.profile`

Get execution times in timer ticks for profiled section of code

Syntax

`ExecutionTimes = NthSectionProfile.ExecutionTimeInTicks`

Description

`ExecutionTimes = NthSectionProfile.ExecutionTimeInTicks` returns a vector of execution times, measured in timer ticks, for the profiled section of code. Each element of `ExecutionTimes` contains the difference between the timer reading at the start and the end of the section. The data type of the arrays is the same as the data type of the timer used on the target, which allows you to infer the maximum range of the timer measurements.

If you set the `CodeProfilingSaveOptions` parameter to `'SummaryOnly'`, *`NthSectionProfile`*`.ExecutionTimeInTicks` returns an empty array. To change that parameter, open the Configuration Parameters dialog box by pressing **Ctrl+E**, open the **Verification** pane under **Code Generation**, and change the **Save options** parameter to `All data`.

You can calculate the execution time in seconds using the formula
ExecutionTimeInSecs = ExecutionTimeInTicks/TimerTicksPerSecond

Examples

Get Execution Time in Ticks

To get a vector of execution times, measured in timer ticks, for the profiled section of code, use the `ExecutionTimeInTicks` property of the `NthSectionProfile` object.

```
ExecutionTimes = NthSectionProfile.ExecutionTimeInTicks;
```

Input Arguments

`NthSectionProfile` — `coder.profile.ExecutionTimeSection` object
`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

`ExecutionTimes` — Vector of execution times
vector of execution times

The `SelfExecutionTimes` is a vector of execution times, in timer ticks, for profiled section of code.

See Also

`MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `display` | `report`

Topics

"Code Execution Profiling with SIL and PIL"
"View and Compare Code Execution Times"
"Analyze Code Execution Data"

Introduced in R2012b

ExecutionTimeInTicks

Package: coder.profile.mc

Get execution times in timer ticks for profiled section of code (MATLAB code generation)

Syntax

```
ExecutionTimes = NthSectionProfile.ExecutionTimeInTicks
```

Description

ExecutionTimes = NthSectionProfile.ExecutionTimeInTicks returns a vector of execution times, measured in timer ticks, for the profiled section of code. Each element of ExecutionTimes contains the difference between the timer reading at the start and the end of the section. The data type of the arrays is the same as the data type of the timer used on the target, which allows you to infer the maximum range of the timer measurements.

Examples

Get Execution Time in Ticks

To get a vector of execution times, measured in timer ticks, for the profiled section of code, use the ExecutionTimeInTicks property of the NthSectionProfile object.

```
ExecutionTimes = NthSectionProfile.ExecutionTimeInTicks
```

Input Arguments

NthSectionProfile — coder.profile.ExecutionTimeSection object
coder.profile.ExecutionTimeSection object

The NthSectionProfile is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

Example: NthSectionProfile

Output Arguments

ExecutionTimes — Vector of execution times
vector of execution times

The SelfExecutionTimes is a vector of execution times, in timer ticks, for profiled section of code.

See Also

ExecutionTimeInSeconds | MaximumExecutionTimeCallNum | MaximumExecutionTimeInTicks | MaximumSelfTimeCallNum | MaximumSelfTimeInTicks | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeInTicks | Name | NumCalls |

`Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` |
`TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` |
`TurnaroundTimeInTicks` | `getCoderExecutionProfile` | `report`

Topics
"Generate Execution Time Profile"
"Analyze Execution Time Data"

Introduced in R2012b

MaximumExecutionTimeCallNum

Package: coder.profile

Get the call number at which maximum number of timer ticks occurred

Syntax

MaxTicksCallNum = NthSectionProfile.MaximumExecutionTimeCallNum

Description

MaxTicksCallNum = NthSectionProfile.MaximumExecutionTimeCallNum returns the call number at which the maximum number of timer ticks was recorded in a single invocation of the profiled code section during a simulation.

Examples

Get Maximum Execution Time Call Number

To get the call number at which the maximum number of timer ticks was recorded, use the MaximumExecutionTimeCallNum property of the NthSectionProfile object.

```
MaxTicksCallNum = NthSectionProfile.MaximumExecutionTimeCallNum;
```

Input Arguments

NthSectionProfile — coder.profile.ExecutionTimeSection object
coder.profile.ExecutionTimeSection object

The NthSectionProfile is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

Example: NthSectionProfile

Output Arguments

MaxTicksCallNum — Call number at which the maximum number of timer ticks occurred
integer

The MaxTicksCallNum is the call number at which the maximum number of timer ticks occurred for a single invocation of the profiled code section.

See Also

ExecutionTimeInTicks | ExecutionTimeInTicks | MaximumExecutionTimeInTicks | MaximumSelfTimeCallNum | MaximumSelfTimeInTicks | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeInTicks | Name | NumCalls | Number | Sections | SelfTimeInTicks | TimerTicksPerSecond | TotalExecutionTimeInTicks | TotalSelfTimeInTicks | TotalTurnaroundTimeInTicks | TurnaroundTimeInTicks | display | report

Topics
"Code Execution Profiling with SIL and PIL"
"View and Compare Code Execution Times"
"Analyze Code Execution Data"

Introduced in R2012b

MaximumExecutionTimeCallNum

Package: `coder.profile.mc`

Get the call number at which maximum number of timer ticks occurred (MATLAB code generation)

Syntax

`MaxTicksCallNum = NthSectionProfile.MaximumExecutionTimeCallNum`

Description

`MaxTicksCallNum = NthSectionProfile.MaximumExecutionTimeCallNum` returns the call number at which the maximum number of timer ticks was recorded in a single invocation of the profiled code section during an execution.

Examples

Get Maximum Execution Time Call Number

To get the call number at which the maximum number of timer ticks was recorded, use the `MaximumExecutionTimeCallNum` property of the `NthSectionProfile` object.

```
MaxTicksCallNum = NthSectionProfile.MaximumExecutionTimeCallNum;
```

Input Arguments

`NthSectionProfile` — `coder.profile.ExecutionTimeSection` object
`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

`MaxTicksCallNum` — Call number at which the maximum number of timer ticks occurred
integer

The `MaxTicksCallNum` is the call number at which the maximum number of timer ticks occurred for a single invocation of the profiled code section

See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` |

`TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `getCoderExecutionProfile` | `report`

Topics

"Generate Execution Time Profile"
"Analyze Execution Time Data"

Introduced in R2012b

MaximumExecutionTimeInTicks

Package: coder.profile

Get maximum number of timer ticks for single invocation of profiled code section

Syntax

MaxTicks = NthSectionProfile.MaximumExecutionTimeInTicks

Description

MaxTicks = NthSectionProfile.MaximumExecutionTimeInTicks returns the maximum number of timer ticks recorded in a single invocation of the profiled code section during a simulation.

Examples

Get Maximum Timer Ticks Recorded

Get the maximum number of timer ticks recorded in a single invocation of the profiled code section during a simulation.

```
MaxTicks = NthSectionProfile.MaximumExecutionTimeInTicks;
```

Input Arguments

NthSectionProfile — ExecutionTimeSection object
ExecutionTimeSection object

The NthSectionProfile is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

Example: NthSectionProfile.MaximumExecutionTimeInTicks

Output Arguments

MaxTicks — Maximum number of timer ticks
integer

Maximum number of timer ticks for single invocation of profiled code section

See Also

ExecutionTimeInTicks | ExecutionTimeInTicks | MaximumExecutionTimeCallNum | MaximumSelfTimeCallNum | MaximumSelfTimeInTicks | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeInTicks | MaximumTurnaroundTimeInTicks | Name | NumCalls | Number | Sections | SelfTimeInTicks | TimerTicksPerSecond | TotalExecutionTimeInTicks | TotalSelfTimeInTicks | TotalTurnaroundTimeInTicks | TotalTurnaroundTimeInTicks | TurnaroundTimeInTicks | TurnaroundTimeInTicks | display | report

Topics
"Code Execution Profiling with SIL and PIL"
"View and Compare Code Execution Times"
"Analyze Code Execution Data"

Introduced in R2012b

MaximumExecutionTimeInTicks

Package: `coder.profile.mc`

Get maximum number of timer ticks for single invocation of profiled code section (MATLAB code generation)

Syntax

`MaxTicks = NthSectionProfile.MaximumExecutionTimeInTicks`

Description

`MaxTicks = NthSectionProfile.MaximumExecutionTimeInTicks` returns the maximum number of timer ticks recorded in a single invocation of the profiled code section during an execution.

Examples

Get Maximum Timer Ticks Recorded

Get the maximum number of timer ticks recorded in a single invocation of the profiled code section.

```
MaxTicks = NthSectionProfile.MaximumExecutionTimeInTicks;
```

Input Arguments

`NthSectionProfile` — `ExecutionTimeSection` object
`ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile.MaximumExecutionTimeInTicks`

Output Arguments

`MaxTicks` — Maximum timer ticks single invocation
integer

The `MaxTicks` is the maximum number of timer ticks for single invocation of profiled code section.

See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `getCoderExecutionProfile` | `report`

Topics
"Generate Execution Time Profile"
"Analyze Execution Time Data"

Introduced in R2012b

TotalExecutionTimeInTicks

Package: `coder.profile`

Get total number of timer ticks recorded for profiled code section

Syntax

```
TotalTicks = NthSectionProfile.TotalExecutionTimeInTicks
```

Description

`TotalTicks = NthSectionProfile.TotalExecutionTimeInTicks` returns the total number of timer ticks recorded for the profiled code section over the entire simulation.

Examples

Get Total Execution Time in Ticks

To get a value for execution time, measured in timer ticks, for the profiled section of code, use the `TotalTicks` property of the `NthSectionProfile` object.

```
TotalTicks = NthSectionProfile.TotalTicks;
```

Input Arguments

`NthSectionProfile` — `coder.profile.ExecutionTimeSection` object
`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

`TotalTicks` — Total number of timer ticks
total number of timer ticks

The `TotalTicks` is the total number of timer ticks for profiled code section.

See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `display` | `report`

Topics
"Code Execution Profiling with SIL and PIL"
"View and Compare Code Execution Times"
"Analyze Code Execution Data"

Introduced in R2012b

TotalExecutionTimeInTicks

Package: coder.profile.mc

Get total number of timer ticks recorded for profiled code section (MATLAB code generation)

Syntax

TotalTicks = NthSectionProfile.TotalExecutionTimeInTicks

Description

TotalTicks = NthSectionProfile.TotalExecutionTimeInTicks returns the total number of timer ticks recorded for the profiled code section over the entire execution.

Examples

Get Total Execution Time in Ticks

To get a value for execution time, measured in timer ticks, for the profiled section of code, use the TotalTicks property of the NthSectionProfile object.

```
TotalTicks = NthSectionProfile.TotalExecutionTimeInTicks;
```

Input Arguments

NthSectionProfile — coder.profile.ExecutionTimeSection object
coder.profile.ExecutionTimeSection object

The NthSectionProfile is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

Example: NthSectionProfile

Output Arguments

TotalTicks — Total number of timer ticks
total number of timer ticks

The TotalTicks is the total number of timer ticks for profiled code section.

See Also

ExecutionTimeInTicks | MaximumExecutionTimeCallNum | MaximumExecutionTimeInTicks | MaximumSelfTimeCallNum | MaximumSelfTimeInTicks | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeInTicks | Name | NumCalls | Number | Sections | SelfTimeInTicks | TimerTicksPerSecond | TotalSelfTimeInTicks | TotalTurnaroundTimeInTicks | TurnaroundTimeInTicks | getCoderExecutionProfile | report

Topics
"Generate Execution Time Profile"
"Analyze Execution Time Data"

Introduced in R2012b

SelfTimeInTicks

Package: `coder.profile`

Get number of timer ticks recorded for profiled code section, excluding time spent in child functions

Syntax

`SelfTicks = NthSectionProfile.SelfTimeInTicks`

Description

`SelfTicks = NthSectionProfile.SelfTimeInTicks` returns the number of timer ticks recorded for the profiled code section. However, this number excludes the time spent in calls to child functions.

Examples

Get Execution Self Time in Ticks

To get the number of timer ticks recorded for the profiled code section, use the `SelfTimeInTicks` property of the `NthSectionProfile` object.

```
SelfTicks = NthSectionProfile.SelfTimeInTicks;
```

Input Arguments

`NthSectionProfile` — `coder.profile.ExecutionTimeSection` object
`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

`SelfTicks` — Number of timer ticks for profiled code section
integer

The `SelfTicks` is the number of timer ticks for profiled code section, excluding periods in child functions.

See Also

`ExecutionTimeInTicks` | `ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` |

`TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `display` | `report`

Topics
"Code Execution Profiling with SIL and PIL"
"View and Compare Code Execution Times"
"Analyze Code Execution Data"

Introduced in R2012b

SelfTimeInTicks

Package: `coder.profile.mc`

Get number of timer ticks recorded for profiled code section, excluding time spent in child functions (MATLAB code generation)

Syntax

`SelfTicks = NthSectionProfile.SelfTimeInTicks`

Description

`SelfTicks = NthSectionProfile.SelfTimeInTicks` returns the number of timer ticks recorded for the profiled code section. However, this number excludes the time spent in calls to child functions.

Examples

Get Execution Self Time in Ticks

To get the number of timer ticks recorded for the profiled code section, use the `SelfTimeInTicks` property of the `NthSectionProfile` object.

```
SelfTicks = NthSectionProfile.SelfTimeInTicks;
```

Input Arguments

`NthSectionProfile` — `coder.profile.ExecutionTimeSection` object
`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

`SelfTicks` — Number of timer ticks for profiled code section
integer

The `SelfTicks` is the number of timer ticks for profiled code section, excluding periods in child functions

See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` |

`TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `getCoderExecutionProfile` | `report`

Topics
"Generate Execution Time Profile"
"Analyze Execution Time Data"

Introduced in R2012b

MaximumSelfTimeCallNum

Get the call number at which the maximum number of timer ticks occurred, excluding time spent in child functions

Syntax

MaxSelfTicksCallNum = *NthSectionProfile*.MaxSelfTimeCallNum

Description

MaxSelfTicksCallNum = *NthSectionProfile*.MaxSelfTimeCallNum returns the call number at which the maximum number of self-time ticks occurred for the profiled code section.

NthSectionProfile is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

Output Arguments

MaxSelfTicksCallNum

Call number at which the maximum number of self-time ticks occurred for profiled code section

See Also

ExecutionTimeInTicks | ExecutionTimeInTicks | MaximumExecutionTimeCallNum | MaximumExecutionTimeInTicks | MaximumSelfTimeInTicks | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeInTicks | Name | NumCalls | Number | Sections | SelfTimeInTicks | TimerTicksPerSecond | TotalExecutionTimeInTicks | TotalSelfTimeInTicks | TotalTurnaroundTimeInTicks | TurnaroundTimeInTicks | display | report

Topics

"Code Execution Profiling with SIL and PIL"
"Code Execution Profiling with SIL and PIL"
"View and Compare Code Execution Times"

Introduced in R2012b

MaximumSelfTimeCallNum

Package: `coder.profile.mc`

Get the call number at which the maximum number of timer ticks occurred, excluding time spent in child functions (MATLAB code generation)

Syntax

```
MaxSelfTicksCallNum = NthSectionProfile.MaximumSelfTimeCallNum
```

Description

`MaxSelfTicksCallNum = NthSectionProfile.MaximumSelfTimeCallNum` returns the call number at which the maximum number of self-time ticks occurred for the profiled code section.

Examples

Get Maximum Execution Time Call Number

To get the call number at which the maximum number of timer ticks was recorded, use the `MaximumExecutionTimeCallNum` property of the `NthSectionProfile` object.

```
MaxSelfTicksCallNum = NthSectionProfile.MaxSelfTimeCallNum;
```

Input Arguments

`NthSectionProfile` — `coder.profile.ExecutionTimeSection` object
`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

`MaxSelfTicksCallNum` — Call number of maximum number of self-time ticks
integer

The `MaxSelfTicksCallNum` is the call number at which the maximum number of self-time ticks occurred for profiled code section.

See Also

`TotalTurnaroundTimeInTicks` | `getCoderExecutionProfile`

Topics

"Generate Execution Time Profile"

"Analyze Execution Time Data"

Introduced in R2012b

MaximumSelfTimeInTicks

Package: `coder.profile`

Get the maximum number of timer ticks recorded for profiled code section, excluding time spent in child functions

Syntax

```
MaxSelfTicks = NthSectionProfile.MaximumSelfTimeInTicks
```

Description

`MaxSelfTicks = NthSectionProfile.MaximumSelfTimeInTicks` returns the maximum number of timer ticks recorded for the profiled code section. This number excludes the time spent in calls to child functions.

Examples

Get Maximum Execution Self Time in Ticks

To get the maximum number of timer ticks recorded for the profiled code section, use the `MaximumSelfTimeInTicks` property of the `NthSectionProfile` object.

```
MaxSelfTicks = NthSectionProfile.MaximumSelfTimeInTicks;
```

Input Arguments

`NthSectionProfile` — `coder.profile.ExecutionTimeSection` object
`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

`MaxSelfTicks` — Maximum number of timer ticks
integer

The `MaxSelfTicks` is the maximum number of timer ticks for profiled code section, excluding periods in child functions.

See Also

`ExecutionTimeInTicks` | `ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` |

`Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `display` | `report`

Topics

"Code Execution Profiling with SIL and PIL"
"View and Compare Code Execution Times"
"Analyze Code Execution Data"

Introduced in R2012b

MaximumSelfTimeInTicks

Package: `coder.profile.mc`

Get the maximum number of timer ticks recorded for profiled code section, excluding time spent in child functions (MATLAB code generation)

Syntax

`MaxSelfTicks = NthSectionProfile.MaximumSelfTimeInTicks`

Description

`MaxSelfTicks = NthSectionProfile.MaximumSelfTimeInTicks` returns the maximum number of timer ticks recorded for the profiled code section. This number excludes the time spent in calls to child functions.

Examples

Get Maximum Execution Self Time in Ticks

To get the maximum number of timer ticks recorded for the profiled code section, use the `MaximumSelfTimeInTicks` property of the `NthSectionProfile` object.

```
MaxSelfTicks = NthSectionProfile.MaximumSelfTimeInTicks;
```

Input Arguments

`NthSectionProfile` — `coder.profile.ExecutionTimeSection` object
`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

`MaxSelfTicks` — Maximum number of timer ticks
integer

The `MaxSelfTicks` is the maximum number of timer ticks for profiled code section, excluding periods in child functions.

See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` |

`TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `getCoderExecutionProfile` | `report`

Topics

"Generate Execution Time Profile"
"Analyze Execution Time Data"

Introduced in R2012b

TotalSelfTimeInTicks

Package: `coder.profile`

Get total number of timer ticks recorded for profiled code section, excluding time spent in child functions

Syntax

```
TotalSelfTicks = NthSectionProfile.TotalSelfTimeInTicks
```

Description

`TotalSelfTicks = NthSectionProfile.TotalSelfTimeInTicks` returns the total number of timer ticks recorded for the profiled code section over the entire simulation. However, this number excludes the time spent in calls to child functions.

Examples

Get Execution Self Time in Ticks

To get a value for execution time, measured in timer ticks, for the profiled section of code, use the `TotalSelfTimeInTicks` property of the `NthSectionProfile` object.

```
TotalSelfTicks = NthSectionProfile.TotalSelfTimeInTicks;
```

Input Arguments

`NthSectionProfile` — `coder.profile.ExecutionTimeSection` object
`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

`TotalSelfTicks` — Value of execution time
value of execution times

The `TotalSelfTicks` is the total number of timer ticks for profiled code section, excluding periods in child functions.

See Also

`ExecutionTimeInTicks` | `ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` |

`Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `display` | `report`

Topics
"Code Execution Profiling with SIL and PIL"
"View and Compare Code Execution Times"
"Analyze Code Execution Data"

Introduced in R2012b

TotalSelfTimeInTicks

Package: `coder.profile.mc`

Get total number of timer ticks recorded for profiled code section, excluding time spent in child functions (MATLAB code generation)

Syntax

`TotalSelfTicks = NthSectionProfile.TotalSelfTimeInTicks`

Description

`TotalSelfTicks = NthSectionProfile.TotalSelfTimeInTicks` returns the total number of timer ticks recorded for the profiled code section over the entire execution. However, this number excludes the time spent in calls to child functions.

Examples

Get Execution Self Time in Ticks

To get a value for execution time, measured in timer ticks, for the profiled section of code, use the `TotalSelfTimeInTicks` property of the `NthSectionProfile` object.

```
TotalSelfTicks = NthSectionProfile.TotalSelfTimeInTicks;
```

Input Arguments

`NthSectionProfile` — `coder.profile.ExecutionTimeSection` object
`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

`TotalSelfTicks` — Value of execution time
value of execution times

The `TotalSelfTicks` is the total number of timer ticks for profiled code section, excluding periods in child functions.

See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` |

`TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `getCoderExecutionProfile` | `report`

Topics
"Generate Execution Time Profile"
"Analyze Execution Time Data"

Introduced in R2012b

MaximumTurnaroundTimeInTicks

Package: `coder.profile`

Get maximum number of timer ticks between start and finish of a single invocation of profiled code section

Syntax

`MaxTurnaroundTicks = NthSectionProfile.MaximumTurnaroundTimeInTicks`

Description

`MaxTurnaroundTicks = NthSectionProfile.MaximumTurnaroundTimeInTicks` returns the maximum number of timer ticks recorded between the start and finish of a single invocation of the profiled code section during a simulation. Unless the code is pre-empted, this is the same as the maximum execution time.

Examples

Get Maximum Turnaround Time in Ticks

To get the call number in which the maximum number of timer ticks was recorded, use the `MaximumTurnaroundTimeInTicks` property of the `NthSectionProfile` object.

```
MaxTurnaroundTicks = NthSectionProfile.MaximumTurnaroundTimeInTicks;
```

Input Arguments

`NthSectionProfile` — `coder.profile.ExecutionTimeSection` object
`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

`MaxTurnaroundTicks` — Maximum number of timer ticks
integer

The `MaxTurnaroundTicks` is the maximum number of timer ticks between start and finish of a single invocation of profiled code section.

See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` |

`Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `display` | `report`

Topics
"Code Execution Profiling with SIL and PIL"
"View and Compare Code Execution Times"
"Analyze Code Execution Data"

Introduced in R2012b

MaximumTurnaroundTimeInTicks

Package: `coder.profile.mc`

Get maximum number of timer ticks between start and finish of a single invocation of profiled code section (MATLAB code generation)

Syntax

`MaxTurnaroundTicks = NthSectionProfile.MaximumTurnaroundTimeInTicks`

Description

`MaxTurnaroundTicks = NthSectionProfile.MaximumTurnaroundTimeInTicks` returns the maximum number of timer ticks recorded between the start and finish of a single invocation of the profiled code section during a execution. Unless the code is pre-empted, this is the same as the maximum execution time.

Examples

Get Maximum Turnaround Time in Ticks

To get the call number in which the maximum number of timer ticks was recorded, use the `MaximumTurnaroundTimeInTicks` property of the `NthSectionProfile` object.

```
MaxTurnaroundTicks = NthSectionProfile.MaximumTurnaroundTimeInTicks;
```

Input Arguments

`NthSectionProfile` — `coder.profile.ExecutionTimeSection` object
`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

`MaxTurnaroundTicks` — Maximum number of timer ticks
integer

The `MaxTurnaroundTicks` is the maximum number of timer ticks between start and finish of a single invocation of profiled code section.

See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` |

`Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `getCoderExecutionProfile` | `report`

Topics
"Generate Execution Time Profile"
"Analyze Execution Time Data"

Introduced in R2012b

MaximumTurnaroundTimeCallNum

Package: `coder.profile`

Get call number of the maximum number of timer ticks between start and finish of a single invocation of profiled code section

Syntax

`MaxTurnaroundTicksCallNum = NthSectionProfile.MaximumTurnaroundTimeCallNum`

Description

`MaxTurnaroundTicksCallNum = NthSectionProfile.MaximumTurnaroundTimeCallNum` returns the call number in which the maximum number of timer ticks was recorded between start and finish of a single invocation of the profiled code section during a simulation. Unless the code is pre-empted, this is the same as the maximum execution time.

Examples

Get Maximum Turnaround Time Call Number

To get the call number in which the maximum number of timer ticks was recorded, use the `MaximumTurnaroundTimeCallNum` property of the `NthSectionProfile` object.

```
MaxTurnaroundTicksCallNum = NthSectionProfile.MaximumTurnaroundTimeCallNum;
```

Input Arguments

`NthSectionProfile` — `coder.profile.ExecutionTimeSection` object
`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

`MaxTurnaroundTicksCallNum` — Call number of the maximum number of timer ticks
integer

The `MaxTurnaroundTicksCallNum` is the call number of the maximum number of timer ticks between start and finish of a single invocation of profiled code section.

See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeInTicks` |

`Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `display` | `report`

Topics

"Code Execution Profiling with SIL and PIL"
"View and Compare Code Execution Times"
"Analyze Code Execution Data"

Introduced in R2012b

MaximumTurnaroundTimeCallNum

Package: coder.profile.mc

Get call number for the code section invocation with the maximum number of timer ticks between the start and the finish (MATLAB code generation)

Syntax

MaxTurnaroundTicksCallNum = NthSectionProfile.MaximumTurnaroundTimeCallNum

Description

MaxTurnaroundTicksCallNum = NthSectionProfile.MaximumTurnaroundTimeCallNum returns the call number in which the maximum number of timer ticks is recorded between the start and the finish of an invocation of the profiled code section. Unless the code is pre-empted, this is the same as the maximum execution time.

Examples

Get Maximum Turnaround Time Call Number

To get the call number in which the maximum number of timer ticks was recorded, use the MaximumTurnaroundTimeCallNum property of the NthSectionProfile object.

```
MaxTurnaroundTicksCallNum = NthSectionProfile.MaximumTurnaroundTimeCallNum;
```

Input Arguments

NthSectionProfile — coder.profile.ExecutionTimeSection object
coder.profile.ExecutionTimeSection object

The NthSectionProfile is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

Example: NthSectionProfile

Output Arguments

MaxTurnaroundTicksCallNum — Call number of the maximum number of timer ticks
integer

The MaxTurnaroundTicksCallNum is the call number of the maximum number of timer ticks between start and finish of a single invocation of profiled code section.

See Also

ExecutionTimeInTicks | MaximumExecutionTimeCallNum | MaximumExecutionTimeInTicks | MaximumSelfTimeCallNum | MaximumSelfTimeInTicks | MaximumTurnaroundTimeInTicks |

`Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `getCoderExecutionProfile` | `report`

Topics
"Generate Execution Time Profile"
"Analyze Execution Time Data"

Introduced in R2012b

TotalTurnaroundTimeInTicks

Package: `coder.profile`

Get total number of timer ticks between start and finish of the profiled code section over the entire simulation

Syntax

`TotalTurnaroundTicks = NthSectionProfile.TotalTurnaroundTimeInTicks`

Description

`TotalTurnaroundTicks = NthSectionProfile.TotalTurnaroundTimeInTicks` returns the total number of timer ticks recorded between the start and finish of the profiled code section over the entire simulation. Unless the code is pre-empted, this is the same as the total execution time.

Examples

Get Total Turnaround Time in Ticks

To get the total number of timer ticks recorded for turnaround time, use the `TotalTurnaroundTimeInTicks` property of the `NthSectionProfile` object.

```
TotalTurnaroundTicks = NthSectionProfile.TotalTurnaroundTimeInTicks;
```

Input Arguments

`NthSectionProfile` — `coder.profile.ExecutionTimeSection` object
`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

`TotalTurnaroundTicks` — Total number of timer ticks between start and finish
integer

The `TotalTurnaroundTicks` is the total number of timer ticks between start and finish of the profiled code section over the entire simulation.

See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` |

`TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TurnaroundTimeInTicks` | `display` | `report`

Topics

"Code Execution Profiling with SIL and PIL"
"View and Compare Code Execution Times"
"Analyze Code Execution Data"

Introduced in R2012b

TotalTurnaroundTimeInTicks

Package: `coder.profile.mc`

Get total number of timer ticks between start and finish of the profiled code section over the entire execution. (MATLAB code generation)

Syntax

`TotalTurnaroundTicks = NthSectionProfile.TotalTurnaroundTimeInTicks`

Description

`TotalTurnaroundTicks = NthSectionProfile.TotalTurnaroundTimeInTicks` returns the total number of timer ticks recorded between the start and finish of the profiled code section over the entire execution. Unless the code is pre-empted, this is the same as the total execution time.

Examples

Get Total Turnaround Time in Ticks

To get the total number of timer ticks recorded for turnaround time, use the `TotalTurnaroundTimeInTicks` property of the `NthSectionProfile` object.

```
TotalTurnaroundTicks = NthSectionProfile.TotalTurnaroundTimeInTicks;
```

Input Arguments

`NthSectionProfile` — `coder.profile.ExecutionTimeSection` object
`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

`TotalTurnaroundTicks` — Total number of timer ticks between start and finish
integer

The `TotalTurnaroundTicks` is the total number of timer ticks between start and finish of the profiled code section over the entire simulation.

See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` |

`TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TurnaroundTimeInTicks` | `getCoderExecutionProfile` | `report`

Topics
"Generate Execution Time Profile"
"Analyze Execution Time Data"

Introduced in R2012b

TurnaroundTimeInTicks

Package: `coder.profile`

Get number of timer ticks between start and finish of the profiled code section

Syntax

`TurnaroundTicks = NthSectionProfile.TurnaroundTimeInTicks`

Description

`TurnaroundTicks = NthSectionProfile.TurnaroundTimeInTicks` returns the number of timer ticks recorded between the start and finish of the profiled code section. Unless the code is pre-empted, this is the same as the execution time.

Examples

Get Turnaround Time in Ticks

To get the number of timer ticks recorded for turnaround time, use the `TurnaroundTimeInTicks` property of the `NthSectionProfile` object.

```
TurnaroundTicks = NthSectionProfile.TurnaroundTimeInTicks;
```

Input Arguments

`NthSectionProfile` — `coder.profile.ExecutionTimeSection` object
`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

`TurnaroundTicks` — Number of timer ticks between start and finish
integer

The `TurnaroundTicks` is the number of timer ticks between start and finish of the profiled code section.

See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `display` | `report`

Topics
"Code Execution Profiling with SIL and PIL"
"View and Compare Code Execution Times"
"Analyze Code Execution Data"

Introduced in R2012b

TurnaroundTimeInTicks

Package: coder.profile.mc

Get number of timer ticks between start and finish of the profiled code section (MATLAB code generation)

Syntax

TurnaroundTicks = NthSectionProfile.TurnaroundTimeInTicks

Description

TurnaroundTicks = NthSectionProfile.TurnaroundTimeInTicks returns the number of timer ticks recorded between the start and finish of the profiled code section. Unless the code is pre-empted, this is the same as the execution time.

Examples

Get Turnaround Time in Ticks

To get the number of timer ticks recorded for turnaround time, use the TurnaroundTimeInTicks property of the NthSectionProfile object.

```
TurnaroundTicks = NthSectionProfile.TurnaroundTimeInTicks;
```

Input Arguments

NthSectionProfile — coder.profile.ExecutionTimeSection object
coder.profile.ExecutionTimeSection object

The NthSectionProfile is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

Example: NthSectionProfile

Output Arguments

TurnaroundTicks — Number of timer ticks between start and finish
integer

The TurnaroundTicks is the number of timer ticks between start and finish of the profiled code section

See Also

ExecutionTimeInTicks | MaximumExecutionTimeCallNum | MaximumExecutionTimeInTicks | MaximumSelfTimeCallNum | MaximumSelfTimeInTicks | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeInTicks | Name | NumCalls | Number | Sections | SelfTimeInTicks |

`TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `getCoderExecutionProfile` | `report`

Topics
"Generate Execution Time Profile"
"Analyze Execution Time Data"

Introduced in R2012b

modifyInheritedParam

Package: rtw.codegenObjectives

Modify inherited parameter values in code generation objective

Syntax

modifyInheritedParam(objective, param, value)

Description

modifyInheritedParam(objective, param, value) changes the value of the specified inherited parameter that the Code Generation Advisor verifies in **Check model configuration settings against code generation objectives**. Use modifyInheritedParam to change the value of a parameter in an objective that you created from an existing objective.

Examples

Create an Objective Based On an Existing Objective

Create a custom objective based on the Traceability objective.

Create a file sl_customization.m to contain a callback function that creates the custom objective.

```
function sl_customization(cm)
%SL_CUSTOMIZATION objective customization callback

objCustomizer = cm.ObjectiveCustomizer;
index = objCustomizer.addCallbackObjFcn(@addObjectives);
objCustomizer.callbackFcn{index}();

end
```

Create and configure the objective in the addObjectives function. Set the name of the objective and modify the list of checks, parameters, and values to verify. Then register the objective in the Code Generation Advisor.

```
function addObjectives

% Create the custom objective
obj = rtw.codegenObjectives.Objective('ex_trace', 'Traceability');
setObjectiveName(obj, 'Custom Traceability Example');

% Remove inherited parameters from the objective
removeInheritedParam(obj, 'MATLABFcnDesc');
removeInheritedParam(obj, 'MATLABSourceComments');

% Remove the inherited code instrumentation check
removeInheritedCheck(obj, 'mathworks.codegen.CodeInstrumentation');

% Modify the inherited parameter 'ConcertIfToSwitch' and set the value to 'on'
modifyInheritedParam(obj, 'ConvertIfToSwitch', 'on');

% Exclude the inherited check for the software environment
```

```
excludeInheritedCheck(obj, 'mathworks.codegen.SWEnvironmentSpec');

%Register the objective
register(obj);

end
```

Input Arguments

`objective` — Code generation objective

`rtw.codegenObjectives.Objective` object

Code generation objective, specified as a `rtw.codegenObjectives.Objective` object.

`param` — Name of inherited parameter

character vector | string scalar

Name of inherited parameter to modify, specified as a character vector or string scalar.

`value` — Parameter value

character vector | string scalar

Parameter value to verify in the Code Generation Advisor, specified as a character vector or string scalar.

See Also

`get_param`

Topics

"Create Custom Code Generation Objectives"

Introduced in R2009a

plot

Create plot for signal or multiple signals

Syntax

```
[signal_names, signal_figures] = cgv.CGV.plot(data_set)
[signal_names, signal_figures] = cgv.CGV.plot(data_set,'Signals',
signal_list)
```

Description

`[signal_names, signal_figures] = cgv.CGV.plot(data_set)` create a plot for each signal in the `data_set`.

`[signal_names, signal_figures] = cgv.CGV.plot(data_set,'Signals', signal_list)` create a plot for each signal in the value of `'Signals'` and return the names and figure handles for the given signal names.

Input Arguments

data_set

Output data from a model. After running the model, use the `getOutputData` function to get the data. The `cgv.CGV.getOutputData` function returns a cell array of the output signal names.

'Signals', signal_list

Parameter/value argument pair specifying the signal or signals to plot. The value for this parameter can be an individual signal name, or a cell array of character vectors, where each character vector is a signal name in the `data_set`. Use `getSavedSignals` to view the list of available signal names in the `data_set`. The syntax for an individual signal name is:

```
signal_list = {'log_data.subsystem_name.Data(:,1)'}
```

The syntax for a list of signal names is:

```
signal_list = {'log_data.block_name.Data(:,1)',...
               'log_data.block_name.Data(:,2)',...
               'log_data.block_name.Data(:,3)',...
               'log_data.block_name.Data(:,4)'};
```

If a component of your model contains a space or newline character, MATLAB adds parentheses and a single quote to the name of the component. For example, if a section of the signal has a space, `'block name'`, MATLAB displays the signal name as:

```
log_data.('block name').Data(:,1)
```

To use the signal name as input to a CGV function, `'block name'` must have two single quotes. For example:

```
signal_list = {'log_data.(''block name'').Data(:,1)'}
```

Output Arguments

Depending on the data, one or more of the following parameters might be empty:

`signal_names`

Cell array of signal names

`signal_figures`

Array of figure handles for signals

See Also

Topics
"Verify Numerical Equivalence with CGV"

register

Package: `rtw.codegenObjectives`

Register objective in Code Generation Advisor

Syntax

`register(objective)`

Description

`register(objective)` registers *`obj`* add the specified objective to the end of the list of available objectives that you can use with the Code Generation Advisor.

Examples

Create a Custom Code Generation Objective

Create a custom objective named `Reduced RAM Example` that runs checks and verifies parameter values to confirm that the model is configured to reduce the RAM used by the generated code.

Create a file `sl_customization.m` to contain a callback function that creates the custom objective.

```
function sl_customization(cm)
%SL_CUSTOMIZATION objective customization callback

objCustomizer = cm.ObjectiveCustomizer;
index = objCustomizer.addCallbackObjFcn(@addObjectives);
objCustomizer.callbackFcn{index}();

end
```

Create and configure the objective in the `addObjectives` function. Set the name of the objective and add checks and parameters to verify. Then register the objective in the Code Generation Advisor.

```
function addObjectives

% Create the custom objective
obj = rtw.codegenObjectives.Objective('ex_ram_1');
setObjectiveName(obj, 'Reduce RAM Example');

% Add parameters to the objective
addParam(obj, 'InlineParams', 'on');
addParam(obj, 'BooleanDataType', 'on');
addParam(obj, 'OptimizeBlockIOStorage', 'on');
addParam(obj, 'EnhancedBackFolding', 'on');
addParam(obj, 'BooleansAsBitfields', 'on');

% Add additional checks to the objective
% The Code Generation Advisor automatically includes 'Check model
% configuration settings against code generation objectives' in every
% objective.
addCheck(obj, 'Identify unconnected lines, input ports, and output ports');
addCheck(obj, 'Check model and local libraries for updates');
```

```
%Register the objective
register(obj);

end
```

Input Arguments

`objective` — Code generation objective
`rtw.codegenObjectives.Objective` object

Code generation objective, specified as a `rtw.codegenObjectives.Objective` object.

See Also

Topics
"Create Custom Code Generation Objectives"
"Registering Customizations"

Introduced in R2009a

registerCFunctionEntry

Create function entry based on specified parameters and register in code replacement table

Syntax

```
entry = registerCFunctionEntry(hTable,priority,numInputs,functionName,
inputType,implementationName,outputType,headerFile,genCallback,genFileName)
```

Description

`entry = registerCFunctionEntry(hTable,priority,numInputs,functionName, inputType,implementationName,outputType,headerFile,genCallback,genFileName)` provides a quick way to create and register a code replacement function entry.

This function can be used only if your function entry meets the following conditions:

- The input arguments are of the same type.
- The input argument names and the return argument name follow the default Simulink naming convention:
  - For input argument names, `u1`, `u2`, ..., `u`*`n`*
  - For return argument, `y1`

Examples

Create C Function Entry in Table

This example shows how to use the `registerCFunctionEntry` function to create a C function entry for `sqrt` in a code replacement table.

```
hLib = RTW.TflTable;

hLib.registerCFunctionEntry(100, 1, 'sqrt', 'double', 'sqrt', ...
                            'double', '<math.h>', '', '');
```

Input Arguments

`hTable` — Handle to a code replacement table
handle

The *`hTable`* is a handle to a code replacement table previously returned by *`hTable`* `= RTW.TflTable`.

Example: `hLib`

`priority` — Specifies the search priority of the function entry
integer 0..100

The *`priority`* specifies the search priority of the function entry, relative to other entries of the same function name and conceptual argument list within this table. Highest priority is 0, and lowest

priority is 100. If the table provides two implementations for a function, the implementation with the higher priority shadows the one with the lower priority.

Example: `100`

`numInputs` — Specifies the number of input arguments
positive integer

Example: `1`

`functionName` — Specifies the name of the function to replace
character vector | string scalar

The *`functionName`* specifies the name of the function to replace. The name must match a function name listed in "Code You Can Replace" in "What Is Code Replacement Customization?" (MATLAB code) or "What Is Code Replacement Customization?" (Simulink models).

Example: `'sqrt'`

`inputType` — Specifies the data type of the input arguments
character vector | string scalar

This function requires that the input arguments are of the same type.

Example: `'double'`

`implementationName` — Specifies the name of the implementation
character vector | string scalar

The *`implementationName`* specifies the name of the implementation. For example, if *`functionName`* is `'sqrt'`, *`implementationName`* can be `'sqrt'` or a different name.

Example: `'sqrt'`

`outputType` — Specifies the data type of the return argument
character vector | string scalar

Example: `'double'`

`headerFile` — Specifies the header file that declares the implementation function
character vector | string scalar

Example: `'<math.h>'`

`genCallback` — Specifies callback that follows code generation
`''` | `'RTW.copyFileToBuildDir'`

The *`genCallback`* specifies a callback that follows code generation. If you specify `'RTW.copyFileToBuildDir'`, and if this function entry is matched and used, the code generator calls function `RTW.copyFileToBuildDir` after code generation. This callback function copies additional header, source, or object files that you have specified for this function entry to the build folder.

Example: `''`

`genFileName` — Specifies `''`
`''` | character vector | string scalar

This argument is reserved for MathWorks developers.

Example: `''`

Output Arguments

`entry` — Handle to the created code replacement function entry
handle

The *`entry`* is a handle to the created code replacement function entry. Specifying the return argument in the `registerCFunctionEntry` function call is optional.

See Also

`registerCPromotableMacroEntry`

Topics

"Define Code Replacement Library Optimizations"

Introduced in R2007b

registerCPPFunctionEntry

Create C++ function entry based on specified parameters and register in code replacement table

Syntax

Description

provides a quick way to create and register a code replacement C++ function entry.

This function can be used only if your C++ function entry meets the following conditions:

- The input arguments are of the same type.
- The input argument names and the return argument name follow the default Simulink naming convention:
  - For input argument names, `u1`, `u2`, ..., `u`*`n`*
  - For return argument, `y1`

When you register a code replacement library containing C++ function entries, you must specify the value `{'C++'}` for the `LanguageConstraint` property of the library registry entry. For more information, see "Register Code Replacement Library".

Examples

Create C++ Function Entry in Table

This example shows how to use the `registerCPPFunctionEntry` function to create a C++ function entry for `sin` in a code replacement table.

```
hLib = RTW.TflTable;

hLib.registerCPPFunctionEntry(100, 1, 'sin', 'single', 'sin', ...
                              'single', 'cmath', '', '', 'std');
```

Input Arguments

`hTable` — Handle to a code replacement table
handle

The *`hTable`* is a handle to a code replacement table previously returned by *`hTable`* `= RTW.TflTable`.

Example: `hLib`

`priority` — Specifies the search priority for the function entry
integer 0..100

The *`priority`* specifies the search priority for the function entry, relative to other entries of the same function name and conceptual argument list within this table. Highest priority is 0, and lowest

priority is 100. If the table provides two implementations for a function, the implementation with the higher priority shadows the one with the lower priority.

Example: `100`

`numInputs` — Specifies the number of input arguments
positive integer

Example: `1`

`functionName` — Specifies the name of the function to replace
character vector | string scalar

The *`functionName`* specifies the name of the function to replace. The name must match a function listed in “Code You Can Replace from MATLAB Code” or “Code You Can Replace From Simulink Models”.

Example: `'sin'`

`inputType` — Specifies the data type of the input arguments
character vector | string scalar

This function requires that the input arguments are of the same type.

Example: `'double'`

`implementationName` — Specifies the name of the implementation
character vector | string scalar

The *`implementationName`* specifies the name of the implementation. For example, if *`functionName`* is `'sqrt'`, *`implementationName`* can be `'sqrt'` or a different name.

Example: `'sqrt'`

`outputType` — Specifies the data type of the return argument
character vector | string scalar

Example: `'double'`

`headerFile` — Specifies the header file that declares the implementation function
character vector | string scalar

Example: `'<math.h>'`

`genCallback` — Specifies callback that follows code generation
`''` | `'RTW.copyFileToBuildDir'`

The *`genCallback`* specifies a callback that follows code generation. If you specify `'RTW.copyFileToBuildDir'`, and if this function entry is matched and used, the code generator calls function `RTW.copyFileToBuildDir` after code generation. This callback function copies additional header, source, or object files that you have specified for this function entry to the build folder.

Example: `''`

`genFileName` — Specifies `''`
`''`

This argument is reserved for MathWorks developers.

Example: `''`

`nameSpace` — Specifies the C++ namespace in which the implementation function is defined
character vector | string scalar

The *`nameSpace`* specifies the C++ namespace in which the implementation function is defined. If this function entry is matched, the software emits the namespace in the generated function code (for example, `std::sin(tfl_cpp_U.In1)`). If you specify `''`, the software does not emit a namespace designation in the generated code.

Example: `'std'`

Output Arguments

`entry` — Handle to the created C++ function entry
handle

The *`entry`* is a handle to the created C++ function entry. Specifying the return argument in the `registerCPPFunctionEntry` function call is optional.

See Also

`enableCPP` | `setNameSpace`

Topics
"Define Code Replacement Library Optimizations"

Introduced in R2010a

registerCPromotableMacroEntry

Create promotable code replacement macro entry based on specified parameters and register in code replacement table (for `abs` function replacement only)

Syntax

```
entry = registerCPromotableMacroEntry(hTable,priority,numInputs,functionName,
inputType,implementationName,outputType,headerFile,genCallback,genFileName)
```

Description

`entry = registerCPromotableMacroEntry(hTable,priority,numInputs,functionName,`
`inputType,implementationName,outputType,headerFile,genCallback,genFileName)`
creates a promotable macro entry based on specified parameters and registers the entry in the code replacement table. A promotable macro entry promotes the output data type based on the target word size.

This function provides a quick way to create and register a promotable macro entry. This function can be used only if your code replacement function entry meets the following conditions:

- The input arguments are of the same type.
- The input argument names and the return argument name follow the default Simulink naming convention:
  - For input argument names, `u1`, `u2`, ..., `u`*n*
  - For return argument, `y1`

Use this function only for `abs` function replacement. For other functions supported for replacement, use `registerCFunctionEntry`.

Examples

Create Promotable Macro Entry in Table

This example shows how to use the `registerCPromotableMacroEntry` function to create a promotable macro entry for `abs` in a code replacement table.

```
hLib = RTW.TflTable;

hLib.registerCPromotableMacroEntry(100, 1, 'abs', ...
    'double', 'abs_prime', ...
    'double', '<math_prime.h>', '', '');
```

Input Arguments

`hTable` — Handle to a code replacement table
handle

The *hTable* is a handle to a code replacement table previously returned by *hTable* = RTW.TflTable.

Example: hLib

priority — Specifies the search priority for the function entry
integer 0..100

The *priority* specifies the search priority for the function entry, relative to other entries of the same function name and conceptual argument list within this table. Highest priority is 0, and lowest priority is 100. If the table provides two implementations for a function, the implementation with the higher priority shadows the one with the lower priority.

Example: 100

numInputs — Specifies the number of input arguments
positive integer

Example: 1

functionName — Specifies the name of the function to replace
character vector | string scalar

The *functionName* specifies the name of the function to be replaced. Specify 'abs'. Use this function only for abs function replacement.

Example: 'abs'

inputType — Specifies the data type of the input arguments
character vector | string scalar

This function requires that the input arguments are of the same type.

Example: 'double'

implementationName — Specifies the name of the implementation
character vector | string scalar

The *implementationName* specifies the name of the implementation. For example, assuming *functionName* is 'abs', *implementationName* can be 'abs' or a different name of your choosing.

Example: 'abs'

outputType — Specifies the data type of the return argument
character vector | string scalar

Example: 'double'

headerFile — Specifies the header file that declares the implementation function
character vector | string scalar

Example: '<math.h>'

genCallback — Specifies callback that follows code generation
'' | 'RTW.copyFileToBuildDir'

The *genCallback* specifies a callback that follows code generation. If you specify 'RTW.copyFileToBuildDir', and if this function entry is matched and used, the code generator

calls function `RTW.copyFileToBuildDir` after code generation. This callback function copies additional header, source, or object files that you have specified for this function entry to the build folder.

Example: `''`

`genFileName` — Specifies `''`
`''` | character vector | string scalar

This argument is reserved for MathWorks developers.

Example: `''`

Output Arguments

`entry` — Handle to the created promotable macro entry
handle

The *`entry`* is a handle to the created promotable macro entry. Specifying the return argument in the `registerCPromotableMacroEntry` function call is optional.

See Also

`registerCFunctionEntry`

Topics
"Define Code Replacement Library Optimizations"

Introduced in R2007b

removeInheritedCheck

Package: `rtw.codegenObjectives`

Remove inherited check from objective

Syntax

`removeInheritedCheck(objective, checkID)`

Description

`removeInheritedCheck(objective, checkID)` removes the specified check that the objective inherited from the objective that you used to create it. Use this method when you create a new objective from an existing objective. If you select multiple objectives in the Code Generation advisor, if another selected objective includes this check, the advisor includes the check.

Examples

Create an Objective Based On an Existing Objective

Create a custom objective based on the `Traceability` objective.

Create a file `sl_customization.m` to contain a callback function that creates the custom objective.

```
function sl_customization(cm)
%SL_CUSTOMIZATION objective customization callback

objCustomizer = cm.ObjectiveCustomizer;
index = objCustomizer.addCallbackObjFcn(@addObjectives);
objCustomizer.callbackFcn{index}();

end
```

Create and configure the objective in the `addObjectives` function. Set the name of the objective and modify the list of checks, parameters, and values to verify. Then register the objective in the Code Generation Advisor.

```
function addObjectives

% Create the custom objective
obj = rtw.codegenObjectives.Objective('ex_trace',, 'Traceability');
setObjectiveName(obj, 'Custom Traceability Example');

% Remove inherited parameters from the objective
removeInheritedParam(obj, 'MATLABFcnDesc');
removeInheritedParam(obj, 'MATLABSourceComments');

% Remove the inherited code instrumentation check
removeInheritedCheck(obj, 'mathworks.codegen.CodeInstrumentation');

% Modify the inherited parameter 'ConcertIfToSwitch' and set the value to 'on'
modifyInheritedParam(obj, 'ConvertIfToSwitch', 'on');

% Exclude the inherited check for the software environment
```

```
excludeInheritedCheck(obj, 'mathworks.codegen.SWEnvironmentSpec');

%Register the objective
register(obj);

end
```

Input Arguments

objective — Code generation objective

rtw.codegenObjectives.Objective object

Code generation objective, specified as a `rtw.codegenObjectives.Objective` object.

checkID — Identifier of check

character vector | string scalar

Identifier of check that you want to remove, specified as a character vector or string scalar.

Example: 'mathworks.codegen.CodeInstrumentation'

See Also

Simulink.ModelAdvisor

Topics

"Create Custom Code Generation Objectives"
Simulink.ModelAdvisor

Introduced in R2009a

removeInheritedParam

Package: `rtw.codegenObjectives`

Remove inherited parameter from objective

Syntax

`removeInheritedParam(objective, param)`

Description

`removeInheritedParam(objective, param)` removes the specified inherited parameter from the objective. Use this method when you create a new objective from an existing objective. When you select multiple objectives in the Code Generation advisor, if another selected objective includes the parameter, the advisor reviews the parameter value using **Check model configuration settings against code generation objectives**.

Examples

Create an Objective Based On an Existing Objective

Create a custom objective based on the `Traceability` objective.

Create a file `sl_customization.m` to contain a callback function that creates the custom objective.

```
function sl_customization(cm)
%SL_CUSTOMIZATION objective customization callback

objCustomizer = cm.ObjectiveCustomizer;
index = objCustomizer.addCallbackObjFcn(@addObjectives);
objCustomizer.callbackFcn{index}();

end
```

Create and configure the objective in the `addObjectives` function. Set the name of the objective and modify the list of checks, parameters, and values to verify. Then register the objective in the Code Generation Advisor.

```
function addObjectives

% Create the custom objective
obj = rtw.codegenObjectives.Objective('ex_trace', 'Traceability');
setObjectiveName(obj, 'Custom Traceability Example');

% Remove inherited parameters from the objective
removeInheritedParam(obj, 'MATLABFcnDesc');
removeInheritedParam(obj, 'MATLABSourceComments');

% Remove the inherited code instrumentation check
removeInheritedCheck(obj, 'mathworks.codegen.CodeInstrumentation');

% Modify the inherited parameter 'ConcertIfToSwitch' and set the value to 'on'
modifyInheritedParam(obj, 'ConvertIfToSwitch', 'on');
```

```
% Exclude the inherited check for the software environment
excludeInheritedCheck(obj, 'mathworks.codegen.SWEnvironmentSpec');

%Register the objective
register(obj);

end
```

Input Arguments

`objective` — Code generation objective
`rtw.codegenObjectives.Objective` object

Code generation objective, specified as a `rtw.codegenObjectives.Objective` object.

`param` — Name of inherited parameter
character vector | string scalar

Name of inherited parameter to remove, specified as a character vector or string scalar.

See Also

`get_param`

Topics
"Create Custom Code Generation Objectives"

Introduced in R2009a

report

Package: `coder.profile`

Open code execution profiling report and specify display of time measurements

Syntax

```
report(myExecutionProfile)
report(myExecutionProfile, name-value pair)
```

Description

`report(myExecutionProfile)` opens the code execution profiling report using default display options.

`report(myExecutionProfile, name-value pair)` opens the report with display options specified by the name-value character vector pairs.

Examples

Create Report by Using Options

To create a report that displays time in microseconds (10^{-6} seconds) with a precision of three decimal places, select options by using name-value pairs.

```
report(myExecutionProfile, ...
    'Units','Seconds', ...
    'ScaleFactor','1e-06', ...
    'NumericFormat','%0.3f')
```

Create Report Comparing Against Baseline

To create a report that compares execution-time performance against a baseline, use the `baseline` option.

```
report(myExecutionProfile, ...
    'baseline', executionProfileBaseline)
```

Input Arguments

`myExecutionProfile` — Variable specifies annotation
workspace variable

`myExecutionProfile` is a workspace variable, specified through the configuration parameter `CodeExecutionProfileVariable` and generated by a simulation.

Example: `myExecutionProfile`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Units','Seconds'`

`Units` — Time measurement unit

`'Seconds'` (default for SIL simulation on Windows) | `'Ticks'` (default for SIL simulation on non-Windows and PIL simulation unless overridden to `'Seconds'` by number of ticks per second in the target connectivity configuration)

Time measurements displayed in seconds or timer ticks.

Example: `'Units','Seconds'`

`ScaleFactor` — Scale factor for displayed measurements

`'1e-9'` (default) | character vector representation of a number that is a power of 10

To display measurements in microseconds, use a scale factor name-value pair `'ScaleFactor'`, `'1e-6'`. The value must be a character vector representation of a number that is a power of 10. For example, `'1'`, `'1e-6'`, or `'1e-9'`. Default value is `'1e-9'`. To specify the scale factor, you must also specify `'Units'`, `'Seconds'`.

Example: `'ScaleFactor','1e-9'`

`NumericFormat` — Numeric format for displayed measurements

`'%0.0f'` (default) | decimal convention utilized by the ANSI C fallback for ANSI C function `sprintf`

Use the *decimal* convention utilized by the ANSI® C function `sprintf`, for example, `'%1.2f'`. Default is `'%0.0f'`. To specify the numeric format, you must also specify `'Units'`, `'Seconds'`.

Example: `'NumericFormat','%0.0f'`

`baseline` — Baseline for performance comparison

`executionProfileBaseline`

The `executionProfileBaseline` is the workspace variable that contains baseline execution-time metrics.

Example: `'baseline',executionProfileBaseline`

See Also

`annotate` | `display`

Topics

“Code Execution Profiling with SIL and PIL”
“View and Compare Code Execution Times”
“Code Execution Profiling Report”
“Compare Code Execution-Time Performance Against Baseline”

Introduced in R2011b

report

Package: `coder.profile.mc`

Open code execution profiling report and specify display of time measurements (MATLAB code generation)

Syntax

```
report(myExecutionProfile)
report(myExecutionProfile, name-value pair)
```

Description

`report(myExecutionProfile)` opens the code execution profiling report using default display options.

`report(myExecutionProfile, name-value pair)` opens the report with display options specified by the name-value character vector pairs.

Examples

Create Report by Using Options

To create a report that displays time in microseconds (10^{-6} seconds) with a precision of three decimal places, select options by using name-value pairs.

```
report(myExecutionProfile, ...
    'Units','Seconds', ...
    'ScaleFactor','1e-06', ...
    'NumericFormat','%0.3f')
```

Create Report Comparing Against Baseline

To create a report that compares execution-time performance against a baseline, use the `baseline` option.

```
report(myExecutionProfile, ...
    'baseline', executionProfileBaseline)
```

Input Arguments

`myExecutionProfile` — Workspace variable created by `getCoderExecutionProfile`
workspace variable

The `myExecutionProfile` is a workspace variable that you create by using the `getCoderExecutionProfile` function.

Example: `myExecutionProfile`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Units','Seconds'`

`Units` — Time measurement unit
`'Seconds'` (default for SIL simulation on Windows) | `'Ticks'` (default for SIL simulation on non-Windows and PIL simulation unless overridden to `'Seconds'` by number of ticks per second in the target connectivity configuration)

Time measurements displayed in seconds or timer ticks.

Example: `'Units','Seconds'`

`ScaleFactor` — Scale factor for displayed measurements
`'1e-9'` (default) | character vector representation of a number that is a power of 10

To display measurements in microseconds, use a scale factor name-value pair `'ScaleFactor'`, `'1e-6'`. The value must be a character vector representation of a number that is a power of 10. For example, `'1'`, `'1e-6'`, or `'1e-9'`. Default value is `'1e-9'`. To specify the scale factor, you must also specify `'Units'`, `'Seconds'`.

Example: `'ScaleFactor','1e-9'`

`NumericFormat` — Numeric format for displayed measurements
`'%0.0f'` (default) | decimal convention utilized by the ANSI C fallback for ANSI C function `sprintf`

Use the *decimal* convention utilized by the ANSI C function `sprintf`, for example, `'%1.2f'`. Default is `'%0.0f'`. To specify the numeric format, you must also specify `'Units'`, `'Seconds'`.

Example: `'NumericFormat','%0.0f'`

`baseline` — Baseline for performance comparison
`executionProfileBaseline`

The `executionProfileBaseline` is the workspace variable that contains baseline execution-time metrics.

Example: `'baseline',executionProfileBaseline`

See Also

`Sections` | `TimerTicksPerSecond` | `getCoderExecutionProfile`

Topics

"Generate Execution Time Profile"
"Analyze Execution Time Data"

Introduced in R2011b

rtIOStreamClose

Shut down communications channel

Syntax

`errFlg = rtIOStreamClose(streamID)`

Description

`errFlg = rtIOStreamClose(streamID)` shuts down the communications channel and cleans up associated resources.

Examples

Close Communications Channel

This code from `rtiostreamtest.c` detects errors when closing the communications channel.

```
static int closeServer(void)
{
    const int errorOccurred = rtIOStreamClose(streamID);
    if (errorOccurred == RTIOSTREAM_ERROR)
    {
        return errorOccurred;
    }
    return RTIOSTREAM_NO_ERROR;
}
```

Input Arguments

`streamID` — Stream handle
scalar integer

Handle to the stream returned by a previous call to `rtIOStreamOpen`.

Output Arguments

`errFlg` — Error flag
0 | -1

If the function runs without errors, it returns zero. Otherwise, it returns -1.

The `rtiostream.h` file defines these macros:

```
#define RTIOSTREAM_ERROR (-1)
#define RTIOSTREAM_NO_ERROR (0)
```

See Also

`rtIOStreamOpen` | `rtIOStreamSend` | `rtIOStreamRecv` | `rtiostream_wrapper`

Topics

"Create PIL Target Connectivity Configuration for MATLAB"
"Create PIL Target Connectivity Configuration for Simulink"
"Create a Target Communication Channel for Processor-In-The-Loop (PIL) Simulation"
"Configure Processor-In-The-Loop (PIL) for a Custom Target"

Introduced in R2009a

rtIOStreamOpen

Initialize communications channel

Syntax

`streamID = rtIOStreamOpen(argCount, argValues)`

Description

`streamID = rtIOStreamOpen(argCount, argValues)` initializes a communication stream to allow the exchange of data between the development computer and target processor.

Examples

Initialize Communications Channel

This code from `rtiostreamtest.c` initializes a communication stream and checks for errors.

```
static int openServer(int rtArgc, void * rtArgv [])
{
    streamID = rtIOStreamOpen(rtArgc, rtArgv);
    if (streamID == RTIOSTREAM_ERROR)
    {
        return streamID;
    }
    return RTIOSTREAM_NO_ERROR;
}
```

Input Arguments

`argCount` — Argument count
scalar integer

Number of elements in `argValues` array.

`argValues` — Driver parameters
array of pointers to character vectors

Parameters for the communications driver.

Output Arguments

`streamID` — Stream handle
positive integer | -1

If the function initializes a communication stream, it returns a positive integer that represents the stream handle. Otherwise, it returns -1, which indicates an error.

The `rtiostream.h` file defines this macro:

```
#define RTIOSTREAM_ERROR (-1)
```

See Also

rtIOStreamSend | rtIOStreamRecv | rtIOStreamClose | rtiostream_wrapper

Topics

"Create PIL Target Connectivity Configuration for MATLAB"
"Create PIL Target Connectivity Configuration for Simulink"
"Create a Target Communication Channel for Processor-In-The-Loop (PIL) Simulation"
"Configure Processor-In-The-Loop (PIL) for a Custom Target"

Introduced in R2009a

rtIOStreamRecv

Receive data through communication channel

Syntax

`errFlg = rtIOStreamRecv(streamID, dest, size, receivedDataSize)`

Description

`errFlg = rtIOStreamRecv(streamID, dest, size, receivedDataSize)` receives data through a communication channel.

Examples

Send and Receive Data from Processor

This code from `rtiostreamtest.c` shows how to send and receive data from a target processor.

```
static void blockingIO(int send, unsigned long numMemUnits)
{
    size_t sizeToTransfer = (size_t) numMemUnits;
    size_t sizeTransferred;
    IOUnit_T * ioPtr = (IOUnit_T *) &buff[0];
    int status;

    if (numMemUnits > BUFFER_SIZE)
    {
        AckCode = stat_notEnoughSpace;
        AckArg0 = BUFFER_SIZE;
        return;
    }

#ifdef HOST_WORD_ADDRESSABLE_TESTING
   /* map to bytes */
   sizeToTransfer *= MEM_UNIT_BYTES;
#endif

   while (sizeToTransfer > 0) {
      sizeTransferred = 0;
      /* Do the low level call */
      status = send ?
         rtIOStreamSend(streamID, ioPtr, sizeToTransfer, &sizeTransferred) :
         rtIOStreamRecv(streamID, ioPtr, sizeToTransfer, &sizeTransferred);

      if (status != RTIOSTREAM_NO_ERROR) {
         if (AckCode == stat_OK) {
            AckCode = stat_RTIOSTREAM_ERROR;
            AckArg0 = data_counter;
         }
         return;
      }
      else {
         sizeToTransfer -= sizeTransferred;
         ioPtr += sizeTransferred;
      }
   }
}
```

Input Arguments

`streamID` — Stream handle
scalar integer

Handle to the stream returned by a previous call to `rtIOStreamOpen`.

`dest` — Data destination
void pointer

Pointer to the start of the buffer for received data.

`size` — Size of data to copy
size_t

Size of data to copy into the destination buffer. For byte-addressable architectures, size is measured in bytes. Some DSP architectures are not byte-addressable. In these cases, size is measured in number of WORDs, where `sizeof(WORD) == 1`.

`receivedDataSize` — Size of data received
size_t

Number of units of data received and copied into the buffer `dest`. If no data is copied, value is zero.

Output Arguments

`errFlg` — Error flag
0 | -1

If the function runs without errors, it returns zero. Otherwise, it returns -1.

The `rtiostream.h` file defines these macros:

```
#define RTIOSTREAM_ERROR (-1)
#define RTIOSTREAM_NO_ERROR (0)
```

See Also

`rtIOStreamSend` | `rtIOStreamOpen` | `rtIOStreamClose` | `rtiostream_wrapper`

Topics
"Create PIL Target Connectivity Configuration for MATLAB"
"Create PIL Target Connectivity Configuration for Simulink"
"Create a Target Communication Channel for Processor-In-The-Loop (PIL) Simulation"
"Configure Processor-In-The-Loop (PIL) for a Custom Target"

Introduced in R2009a

rtIOStreamSend

Send data through communication channel

Syntax

`errFlag = rtIOStreamSend(streamID, src, size, sizeSent)`

Description

`errFlag = rtIOStreamSend(streamID, src, size, sizeSent)` sends data through a communication stream.

The API for `rtIOStream` functions is independent of the physical layer across which you send the data, for example, RS232, Ethernet, or Controller Area Network (CAN). The choice of physical layer affects the achievable data rates for communication between your development computer and target processor.

For a processor-in-the-loop (PIL) application, there is no minimum data rate requirement. The higher the data rate is, the faster the simulation runs.

A communications device driver can require additional hardware-specific or channel-specific configuration parameters. For example:

- A CAN channel can require the specification of the CAN node that is used.
- A TCP/IP channel can require the configuration of a port or static IP address.
- A CAN channel can require the specification of the CAN message ID and priority.

When you implement the `rtIOStream` driver functions, provide this configuration data, for example, by hard-coding the data or by supplying arguments to `rtIOStreamOpen`.

Examples

Send and Receive Data from Processor

This code from `rtiostreamtest.c` shows how to send and receive data from a target processor.

```
static void blockingIO(int send, unsigned long numMemUnits)
{
    size_t sizeToTransfer = (size_t) numMemUnits;
    size_t sizeTransferred;
    IOUnit_T * ioPtr = (IOUnit_T *) &buff[0];
    int status;

    if (numMemUnits > BUFFER_SIZE)
    {
        AckCode = stat_notEnoughSpace;
        AckArg0 = BUFFER_SIZE;
        return;
    }

#ifdef HOST_WORD_ADDRESSABLE_TESTING
   /* map to bytes */
   sizeToTransfer *= MEM_UNIT_BYTES;
#endif
```

```
    while (sizeToTransfer > 0) {
       sizeTransferred = 0;
       /* Do the low level call */
       status = send ?
          rtIOStreamSend(streamID, ioPtr, sizeToTransfer, &sizeTransferred) :
          rtIOStreamRecv(streamID, ioPtr, sizeToTransfer, &sizeTransferred);

       if (status != RTIOSTREAM_NO_ERROR) {
          if (AckCode == stat_OK) {
             AckCode = stat_RTIOSTREAM_ERROR;
             AckArg0 = data_counter;
          }
          return;
       }
       else {
          sizeToTransfer -= sizeTransferred;
          ioPtr += sizeTransferred;
       }
    }
 }
```

Input Arguments

`streamID` — Stream handle

scalar integer

Handle to the stream returned by a previous call to `rtIOStreamOpen`.

`src` — Data source

constant void pointer

Pointer to the start of the buffer that contains a data array for transmission.

`size` — Size of data to transmit

size_t

Size of data to transmit from the source buffer. For byte-addressable architectures, size is measured in bytes. Some DSP architectures are not byte-addressable. In these cases, size is measured in number of WORDs, where `sizeof(WORD) == 1`.

Output Arguments

`errFlag` — Error flag

0 | -1

If the function runs without errors, it returns zero. Otherwise, it returns -1.

The `rtiostream.h` file defines these macros:

```
#define RTIOSTREAM_ERROR (-1)
#define RTIOSTREAM_NO_ERROR (0)
```

`sizeSent` — Size of transmitted data

size_t pointer

Size of transmitted data, which is less than or equal to `size`. If data is not transmitted, value is zero.

See Also

`rtIOStreamOpen` | `rtIOStreamClose` | `rtIOStreamRecv` | `rtiostream_wrapper`

Topics

"Create PIL Target Connectivity Configuration for MATLAB"
"Create PIL Target Connectivity Configuration for Simulink"
"Create a Target Communication Channel for Processor-In-The-Loop (PIL) Simulation"
"Configure Processor-In-The-Loop (PIL) for a Custom Target"

Introduced in R2009a

rtiostream_wrapper

Test `rtiostream` shared library functions in MATLAB

Syntax

```
streamID = rtiostream_wrapper(sharedLib, 'open')
[errFlag,transmittedDataSize] = rtiostream_wrapper(sharedLib,'send',streamID,
data,dataSize)
[errFlag,receivedData,receivedDataSize] = rtiostream_wrapper(
sharedLib,'recv',streamID,dataSize)
streamID = rtiostream_wrapper( ___ ,Name,Value)
errFlag = rtiostream_wrapper(sharedLib,'close',streamID)
rtiostream_wrapper(sharedLib,'unloadlibrary')
```

Description

`streamID = rtiostream_wrapper(sharedLib, 'open')` opens an `rtiostream` communication channel or stream through a shared library.

`[errFlag,transmittedDataSize] = rtiostream_wrapper(sharedLib,'send',streamID, data,dataSize)`transmits data from a workspace variable through the open communication channel or stream.

`[errFlag,receivedData,receivedDataSize] = rtiostream_wrapper( sharedLib,'recv',streamID,dataSize)` receives workspace variable data from the open communication channel or stream.

`streamID = rtiostream_wrapper( ___ ,Name,Value)` specifies additional options using one or more name-value pair arguments. These arguments are implementation-dependent, that is, they are specific to the shared library that you use.

`errFlag = rtiostream_wrapper(sharedLib,'close',streamID)` closes the `rtiostream` communication channel or stream.

`rtiostream_wrapper(sharedLib,'unloadlibrary')` unloads the shared library, clearing persistent data.

Examples

Open Communication Channels

These examples use the supplied TCP/IP and serial communication drivers to open communication channels.

Open `rtiostream stationA` as a TCP/IP server:

```
stationA = rtiostream_wrapper('libmwrtiostreamtcpip.dll','open',...
                              '-client', '0',...
                              '-blocking', '0',...
                              '-port', port_number);
```

Opens `rtiostream StationB` as a TCP/IP client:

```
stationB = rtiostream_wrapper('libmwrtiostreamtcpip.dll','open',...
                              '-client','1',...
                              '-blocking', '0',...
                              '-port', port_number,...
                              '-hostname','localhost');
```

If you use the supplied development computer driver for serial communications (as an alternative to the drivers for TCP/IP), specify the bit rate when you open a channel with a specific port. For example, open channel `stationA` with port `COM1` and bit rate of 9600:

```
stationA = rtiostream_wrapper('libmwrtiostreamserial.dll','open',...
                              '-port','COM1',...
                              '-baud','9600');
```

Input Arguments

`sharedLib` — Shared library

character vector

Shared library that implements required `rtIOStream` functions, `rtIOStreamOpen`, `rtIOStreamSend`, `rtIOStreamRecv`, and `rtIOStreamClose`. Must be on system path. Specify one of these values:

- *`libTCPIP`* — For TCP/IP communication. Value depends on your operating system, for example, `'libmwrtiostreamtcpip.dll'`
- *`libSerial`* — For serial communication, for example, `'libmwrtiostreamserial.dll'`.

`streamID` — Stream handle

scalar integer

Handle to `rtiostream` communication stream.

`data` — Workspace variable

array

Array that contains data for transmission.

`dataSize` — Size of data to transmit or receive

scalar integer

Size of workspace variable data to transmit or receive, in bytes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: '-hostname','localhost'

TCP/IP Communication

`-client` — Stream type

0 | 1

Open `rtiostream` as TCP/IP server or client:

- 0 — TCP/IP server
- 1 — TCP/IP client

-port — Port number
scalar integer

Port for TCP/IP communication.

-hostname — Development computer
character vector

Identifier for your development computer, for example, `'localhost'`.

-blocking — Call behavior
0 | 1

Call behavior when receiving data:

- 0 — Polling mode. If data is available, call returns with data. If data is not available, call returns without waiting.
- 1 — Blocking mode. If data is available, call returns with data. If data is not available, call waits for data. Use `recv_timeout_secs` to specify the waiting period.

Default is 0 unless the preprocessor macro `define VXWORKS` exists. In this case, the default is 1.

-recv_timeout_secs — Waiting period
positive integer | 0 | -1 | -2 | -3

Waiting period of call that receives data:

- *X*, a positive integer — Wait for *X* seconds.
- 0 — No waiting period.
- -1 — Wait indefinitely.
- -2 — Wait for default period.
- -3 — Wait 10 ms.

Default for client connections is to wait 1 second. Default for server connections is to wait indefinitely.

Serial Communication

-port — Port number
scalar integer

COM port for serial communication.

-baud — Bit rate
scalar integer

Bit rate for serial communication port.

Output Arguments

`streamID` — Stream handle
scalar integer

If the function opens the communications stream, it returns a handle to the stream. Otherwise, it returns -1.

`errFlag` — Error flag
0 | -1

If the function runs without errors, it returns zero. Otherwise, it returns -1.

`transmittedDataSize` — Size of transmitted data
scalar integer

Size of data transmitted through communication stream. Can be less than `dataSize`, the number of bytes specified for transmission.

`receivedData` — Workspace variable received
array

Array that contains received data.

`receivedDataSize` — Size of received data
scalar integer

Number of bytes received from communication stream. Can be less than `dataSize`, the number of bytes specified for transmission.

See Also

`rtIOStreamOpen` | `rtIOStreamSend` | `rtIOStreamRecv` | `rtIOStreamClose`

Topics

"Create PIL Target Connectivity Configuration for MATLAB"
"Create PIL Target Connectivity Configuration for Simulink"
"Create a Target Communication Channel for Processor-In-The-Loop (PIL) Simulation"
"Configure Processor-In-The-Loop (PIL) for a Custom Target"

Introduced in R2008b

rtw.codegenObjectives.Objective

Custom code generation objective

Description

Use a code generation objective to specify a set of checks and parameters for the Code Generation Advisor to verify.

Creation

Syntax

```
rtw.codegenObjectives.Objective(newObjID)
rtw.codegenObjectives.Objective(newObjID, baseObjID)
```

Description

`rtw.codegenObjectives.Objective(newObjID)` creates a new code generation objective that has the specified identifier.

`rtw.codegenObjectives.Objective(newObjID, baseObjID)` creates a code generation objective based on the specified existing code generation objective. The existing objective must be registered in the Code Generation Advisor.

Input Arguments

`newObjID` — Identifier of new objective
character vector | string scalar

Identifier of new objective, specified as a character vector or string scalar.

`baseObjID` — Identifier of existing objective
character vector | string scalar

Identifier of existing objective that the new objective is based on, specified as a character vector or string scalar.

Object Functions

setObjectiveName	Specify objective name
addCheck	Add check to code generation objective
addParam	Add parameters to code generation objective
excludeCheck	Exclude check from code generation objective
modifyInheritedParam	Modify inherited parameter values in code generation objective
removeInheritedCheck	Remove inherited check from objective
removeInheritedParam	Remove inherited parameter from objective
register	Register objective in Code Generation Advisor

Examples

Create a Custom Code Generation Objective

Create a custom objective named `Reduced RAM Example` that runs checks and verifies parameter values to confirm that the model is configured to reduce the RAM used by the generated code.

Create a file `sl_customization.m` to contain a callback function that creates the custom objective.

```
function sl_customization(cm)
%SL_CUSTOMIZATION objective customization callback

objCustomizer = cm.ObjectiveCustomizer;
index = objCustomizer.addCallbackObjFcn(@addObjectives);
objCustomizer.callbackFcn{index}();

end
```

Create and configure the objective in the `addObjectives` function. Set the name of the objective and add checks and parameters to verify. Then register the objective in the Code Generation Advisor.

```
function addObjectives

% Create the custom objective
obj = rtw.codegenObjectives.Objective('ex_ram_1');
setObjectiveName(obj, 'Reduce RAM Example');

% Add parameters to the objective
addParam(obj, 'InlineParams', 'on');
addParam(obj, 'BooleanDataType', 'on');
addParam(obj, 'OptimizeBlockIOStorage', 'on');
addParam(obj, 'EnhancedBackFolding', 'on');
addParam(obj, 'BooleansAsBitfields', 'on');

% Add additional checks to the objective
% The Code Generation Advisor automatically includes 'Check model
% configuration settings against code generation objectives' in every
% objective.
addCheck(obj, 'Identify unconnected lines, input ports, and output ports');
addCheck(obj, 'Check model and local libraries for updates');

%Register the objective
register(obj);

end
```

See Also

Topics
"Create Custom Code Generation Objectives"

Introduced in R2009a

rtw.connectivity.ComponentArgs

Provide parameters for each target connectivity component

Description

An `rtw.connectivity.ComponentArgs` object provides functions for getting information about the source component and the target application.

Creation

Description

`compArgs = rtw.connectivity.ComponentArgs (componentPath, componentCodePath, componentCodeName, applicationCodePath)` returns a handle to an `rtw.connectivity.ComponentArgs` object.

Object Functions

Function	Description
`getComponentPath`	*`cmpPath`* `=` *`compArgs`*`.getComponentPath` returns the system path of the source component. For example: • For MATLAB, the path of the function that is under test. • For Simulink, the path of the referenced model that is under test.
`getComponentCodePath`	*`cmpCodePath`* `=` *`compArgs`*`.getComponentCodePath` returns the code generation folder path associated with the source component. For example: • For MATLAB, the code generation folder of the MATLAB function that is under test. • For Simulink, the code generation folder of the referenced model that is under test.
`getComponentCodeName`	*`cmpCodeName`* `=` *`compArgs`*`.getComponentCodeName` returns the component name used for code generation.
`getApplicationCodePath`	*`appCodePath`* `=` *`compArgs`*`.getApplicationCodePath` returns the folder path associated with the target application, for example, the path associated with the PIL application.
`getParam`	*`settingOrParameterValue`* `=` *`compArgs`*`.getParam(`*`settingOrParameterName`*`)` returns: • For MATLAB, the value of the specific MATLAB Coder setting for the generated code. • For Simulink, the value of the specific model configuration parameter for the generated code. The function does not load the model.

Examples

Using `rtw.connectivity.ComponentArgs` in PIL Connectivity

For an example that shows how to use this object in setting up PIL connectivity, see:

- For MATLAB, “Processor-in-the-Loop Execution From Command Line”
- For Simulink, “Configure Processor-In-The-Loop (PIL) for a Custom Target”

See Also

`rtw.connectivity.Config`

Topics

“Create PIL Target Connectivity Configuration for MATLAB”
“Create PIL Target Connectivity Configuration for Simulink”

Introduced in R2008b

rtw.connectivity.Config

Define connectivity implementation that comprises builder, launcher, and communicator components

Description

The `rtw.connectivity.Config` class specifies the actions required for running a processor-in-the-loop (PIL) simulation.

Creation

Description

`rtw.connectivity.Config(componentArgs, builder, launcher, communicator)` creates an `rtw.connectivity.Config` object with these arguments:

- *`componentArgs`* – `rtw.connectivity.ComponentArgs` object
- *`builder`* – `rtw.connectivity.Builder` object, for example, `rtw.connectivity.MakefileBuilder` object.
- *`launcher`* – `rtw.connectivity.Launcher` object
- *`communicator`* – `rtw.connectivity.Communicator`, for example, `rtw.connectivity.RtIOStreamHostCommunicator` object.

To define a connectivity implementation:

1. Create a subclass of `rtw.connectivity.Config` that creates instances of your connectivity component classes:

   - `rtw.connectivity.MakefileBuilder`
   - `rtw.connectivity.Launcher`
   - `rtw.connectivity.RtIOStreamHostCommunicator`

2. Define the constructor for your subclass:

   ```
   function this = myConfig(componentArgs)
   ```

   When the software creates an instance of your subclass of `rtw.connectivity.Config`, it provides an instance of the `rtw.connectivity.ComponentArgs` class as the only constructor argument. If you want to test your subclass of `rtw.connectivity.Config` manually, you can create an `rtw.connectivity.ComponentArgs` object to pass as a constructor argument.

3. After instantiating the builder, launcher and communicator objects in your subclass, call the constructor of the superclass `rtw.connectivity.Config` to define your complete target connectivity configuration. For example:

   ```
   this@rtw.connectivity.Config(componentArgs,...
   builder, launcher, communicator);
   ```

4 Optionally, for execution-time profiling, use the `setTimer` method to register your hardware timer. For example, if you specified the timer in a code replacement table, insert the following line:

```
this.setTimer('myCrlTable')
```

`myCrlTable` is the name of the code replacement table, which must be in a location on the MATLAB search path.

You can also estimate and remove instrumentation overheads from execution-time measurements. For example:

```
this.activateOverheadFiltering(true);
this.runOverheadBenchmark(true);
this.setOverheadBenchmarkSteps(50);
```

Register your subclass name, for example, `myPIL.ConnectivityConfig` by using the class `rtw.connectivity.ConfigRegistry`. The PIL infrastructure instantiates your subclass as required. The `rtwTargetInfo.m` file (for MATLAB) or `sl_customization.m` mechanism (for Simulink) specifies a suitable connectivity configuration for use with a particular PIL component (and its configuration set). The subclass can also perform additional validation on construction. For example, you can use the component path returned by the `getComponentPath` method of the `componentArgs` constructor argument to query and validate parameters associated with the PIL component under test.

Examples

Using `rtw.connectivity.Config` in PIL Connectivity

For an example that shows how to use this object in setting up PIL connectivity, see:

- For MATLAB, “Processor-in-the-Loop Execution From Command Line”
- For Simulink, “Configure Processor-In-The-Loop (PIL) for a Custom Target”

See Also

`rtw.connectivity.MakefileBuilder` | `rtw.connectivity.Launcher` | `rtw.connectivity.RtIOStreamHostCommunicator` | `rtw.connectivity.ComponentArgs`

Topics

“Create PIL Target Connectivity Configuration for MATLAB”
“Create PIL Target Connectivity Configuration for Simulink”
“Specify Hardware Timer”
“Specify Hardware Timer”
“Remove Instrumentation Overheads from Execution Time Measurements”
“SIL and PIL Limitations”

Introduced in R2008b

rtw.connectivity.ConfigRegistry

Register connectivity configuration

Description

Register your connectivity configuration with MATLAB or Simulink.

Creation

Description

`config = rtw.connectivity.ConfigRegistry` returns a handle to an `rtw.connectivity.ConfigRegistry` object.

To create this class:

- For MATLAB, use an `rtwTargetInfo.m` file, which you must place on the MATLAB search path. In the `rtwTargetInfo.m` file, a call to `registerTargetInfo` registers the connectivity configuration.
- For Simulink, use an `sl_customization.m` file, which you must place on the MATLAB search path. When Simulink starts, it reads the file, and registers your connectivity configuration through a call to `registerTargetInfo` in the file.

Through the first two properties of this class, you can specify for your connectivity configuration:

- A unique name.
- An associated connectivity implementation class, which is a subclass of `rtw.connectivity.Config`.

Through the remaining properties, you can define:

- For MATLAB, the code that is compatible with the connectivity implementation class.
- For Simulink, the set of models that are compatible with the connectivity implementation class.

A comparison of the union of these properties against the MATLAB Coder configuration settings or Simulink model parameters determines compatibility. For example with Simulink, whether the `SystemTargetFile`, `TemplateMakefile`, and `HardwareBoard` properties jointly match the corresponding model parameters.

Properties

`ConfigName` — Name
character vector

Unique name for configuration.

`ConfigClass` — Class name
character vector

Full class name of the connectivity implementation that you want to register.

`SystemTargetFile` — System target files (Simulink)

cell array of character vectors | {}

For Simulink, system target files that support the `rtw.connectivity.ConfigRegistry` object you create. A comparison of this cell array against the `SystemTargetFile` configuration parameter of the model determines whether the created object is valid for use. An empty cell array matches any system target file.

`TemplateMakefile` — Template makefiles (Simulink)

cell array of character vectors | {}

For Simulink, template makefiles that support the `rtw.connectivity.ConfigRegistry` object you create. A comparison of this cell array against the `TemplateMakefile` configuration parameter of the model determines whether the created object is valid for use. An empty cell array matches any template makefile and non-makefile target (`GenerateMakefile`: off).

If you use a toolchain to build the generated code, do not specify the `TemplateMakefile` configuration parameter. Instead, specify the `Toolchain` configuration parameter.

`Toolchain` — Toolchains

cell array of character vectors | {}

Toolchains that support the `rtw.connectivity.ConfigRegistry` object you create:

- For MATLAB, a comparison of this cell array against the MATLAB Coder `Toolchain` configuration setting determines whether the created object is valid for use.
- For Simulink, a comparison of this cell array against the `Toolchain` configuration parameter of the model determines whether the created object is valid for use. If you do not use a toolchain to build the generated code, do not specify the `Toolchain` configuration parameter. Instead, specify the `TemplateMakefile` configuration parameter.

An empty cell array matches any toolchain.

`HardwareBoard` — Hardware boards

cell array of character vectors | {}

Hardware boards that support the `rtw.connectivity.ConfigRegistry` object you create:

- For MATLAB, a comparison of this cell array against the MATLAB Coder `HardwareBoard` configuration setting determines whether the created object is valid for use.
- For Simulink, a comparison of this cell array against the `HardwareBoard` configuration parameter of the model determines whether the created object is valid for use.

An empty cell array matches any hardware board.

`TargetHWDeviceType` — Hardware device types

cell array of character vectors | {}

Hardware device types that support the `rtw.connectivity.ConfigRegistry` object you create:

- For MATLAB, a comparison of this cell array against the MATLAB Coder `TargetHWDeviceType` configuration setting determines whether the created object is valid for use.

- For Simulink, a comparison of this cell array against the `TargetHWDeviceType` configuration parameter of the model determines whether the created object is valid for use.

An empty cell array matches any hardware device type.

Examples

Create `rtwTargetInfo.m` File

This code is an example `rtwTargetInfo.m` file. Use the function syntax exactly as shown.

```
function rtwTargetInfo(tr)
% Register PIL connectivity config: mypil.ConnectivityConfig

tr.registerTargetInfo(@loc_createConfig);

% local function
function config = loc_createConfig

% Create object for connectivity configuration
config = rtw.connectivity.ConfigRegistry;
% Assign connectivity configuration name
config.ConfigName = 'My PIL Example';
% Associate the connectivity configuration with the connectivity
% API implementation
config.ConfigClass = 'mypil.ConnectivityConfig';

% Specify toolchains for host-based PIL
config.Toolchain =  rtw.connectivity.Utils.getHostToolchainNames;

% Through the HardwareBoard and TargetHWDeviceType properties,
% define compatible code for the target connectivity configuration
config.HardwareBoard = {};
config.TargetHWDeviceType = {'Generic->32-bit x86 compatible' ...
                             'Generic->Custom' ...
                             'Intel->x86-64 (Windows64)', ...
                             'Intel->x86-64 (Mac OS X)', ...
                             'Intel->x86-64 (Linux 64)'};
```

The function performs the following steps:

1 Creates an instance of the `rtw.connectivity.ConfigRegistry` class. For example:

```
config = rtw.connectivity.ConfigRegistry;
```

2 Assigns a connectivity configuration name to the `ConfigName` property of the object. For example:

```
config.ConfigName = 'My PIL Example';
```

3 Associates the connectivity configuration with the connectivity API implementation created in step 1. For example:

```
config.ConfigClass = 'mypil.ConnectivityConfig';
```

4 Defines compatible code for this target connectivity configuration, by setting the `HardwareBoard` and `TargetHWDeviceType` properties of the object. For example:

```
config.HardwareBoard = {};  % Any hardware board
config.TargetHWDeviceType = {'Generic->32-bit x86 compatible' ...
```

```
                                    'Generic->Custom' ...
                                    'Intel->x86-64 (Windows64)', ...
                                    'Intel->x86-64 (Mac OS X)', ...
                                    'Intel->x86-64 (Linux 64)'};
```

Create `sl_customization.m` File

This code is an example of an `sl_customization.m` file. Use the `sl_customization.m` file structure, and call the `registerTargetInfo` function exactly as shown.

```
function sl_customization(cm)
% SL_CUSTOMIZATION for PIL connectivity config:...
% mypil.ConnectivityConfig

% Copyright 2008 The MathWorks, Inc.

cm.registerTargetInfo(@loc_createConfig);

% local function
function config = loc_createConfig

config = rtw.connectivity.ConfigRegistry;
config.ConfigName = 'My PIL Example';
config.ConfigClass = 'mypil.ConnectivityConfig';

% Match only ert.tlc
config.SystemTargetFile = {'ert.tlc'};

% If you use a toolchain to build your generated code,
% specify the config.Toolchain property to match your
% Simulink model toolchain setting. Otherwise, for a
% non-toolchain approach, match the TMF
config.TemplateMakefile = {'ert_default_tmf' ...
                           'ert_unix.tmf', ...
                           'ert_vcx64.tmf', ...
                           'ert_lcc.tmf'};

% Match hardware boards and hardware device types
config.HardwareBoard = {};  % Any hardware board
config.TargetHWDeviceType = {'Generic->32-bit x86 compatible' ...
                             'Generic->Custom' ...
                             'Intel->x86-64 (Windows64)', ...
                             'Intel->x86-64 (Mac OS X)', ...
                             'Intel->x86-64 (Linux 64)'};
```

You must configure the file to perform the following steps when Simulink starts:

1. Create an instance of the `rtw.connectivity.ConfigRegistry` class. For example:

   ```
   config = rtw.connectivity.ConfigRegistry;
   ```
2. Assign a connectivity configuration name to the `ConfigName` property of the object. For example:

   ```
   config.ConfigName = 'My PIL Example';
   ```
3. Associate the connectivity configuration with the connectivity API implementation (created in step 1). For example:

   ```
   config.ConfigClass = 'mypil.ConnectivityConfig';
   ```

4 Define compatible models for this target connectivity configuration, by setting these properties of the properties of the object:

- `SystemTargetFile`
- `Toolchain` or `TemplateMakefile`
- `HardwareBoard`
- `TargetHWDeviceType`

For example:

```
config.SystemTargetFile = {'ert.tlc'};
config.TemplateMakefile = {'ert_default_tmf' ...
                           'ert_unix.tmf', ...
                           'ert_vcx64.tmf', ...
                           'ert_lcc.tmf'};
config.HardwareBoard = {};
config.TargetHWDeviceType = {'Generic->32-bit x86 compatible' ...
                             'Generic->Custom' ...
                             'Intel->x86-64 (Windows64)', ...
                             'Intel->x86-64 (Mac OS X)', ...
                             'Intel->x86-64 (Linux 64)'};
```

Using `rtw.connectivity.ConfigRegistry` in PIL Connectivity

For an example that shows how to use this object in setting up PIL connectivity, see:

- For MATLAB, “Processor-in-the-Loop Execution From Command Line”
- For Simulink, “Configure Processor-In-The-Loop (PIL) for a Custom Target”

See Also

`rtw.connectivity.Config`

Topics

“Create PIL Target Connectivity Configuration for MATLAB”
“Create PIL Target Connectivity Configuration for Simulink”
“Register Code Replacement Library”

Introduced in R2008b

rtw.connectivity.Launcher

Control downloading, starting, and resetting of a target application

Description

The `rtw.connectivity.Launcher` class, which runs on your development computer, controls execution of an application on the target processor.

Creation

Description

`rtw.connectivity.Launcher(componentArgs)` controls the download, start, and reset of an application, for example, a PIL application.

Make a subclass and implement the `startApplication` and `stopApplication` methods.

You can implement a destructor method that cleans up resources (for example, a handle to a third-party download tool) when the object is cleared from memory.

Object Functions

Function	Description
`getComponentArgs`	`componentArgs = obj.getComponentArgs` returns the `rtw.connectivity.ComponentArgs` object associated with the launcher object.
`setExe`	`setExe(exe)` specifies the application that runs on the target processor.
`getExe`	`exe=getExe()` returns the application that is running on the target processor.
`startApplication`	*`obj`*`.startApplication` is an abstract method that you implement in a subclass. Called by MATLAB or Simulink to start execution of the target application. MATLAB or Simulink calls the `setExe` method, which specifies the target application to run. To obtain this application, use the `getExe` method. For example: `exe = getExe()` The `startApplication` method resets the application to its initial state by ensuring that external and static (global) variables are zero-initialized.
`stopApplication`	*`obj`*`.stopApplication` is an abstract method that you must implement in a subclass. Called by MATLAB to stop execution of the target application.

Function	Description
getApplicationStatus	*obj*.getApplicationStatus is an optional method that you can implement in a subclass. Called by MATLAB or Simulink to detect the current status of the target application. The expected return values are: • rtw.connectivity.LauncherApplicationStatus.UNKNOWN • rtw.connectivity.LauncherApplicationStatus.NOT_RUNNING • rtw.connectivity.LauncherApplicationStatus.RUNNING If you do not implement the method, the default return value is rtw.connectivity.LauncherApplicationStatus.UNKNOWN.
getBuilder	builder = obj.getBuilder returns the rtw.connectivity.Builder object associated with the launcher object.

Examples

Using rtw.connectivity.Launcher in PIL Connectivity

For an example that shows how to use this object in setting up PIL connectivity, see:

- For MATLAB, “Processor-in-the-Loop Execution From Command Line”
- For Simulink, “Configure Processor-In-The-Loop (PIL) for a Custom Target”

See Also

rtw.connectivity.MakefileBuilder | rtw.connectivity.RtIOStreamHostCommunicator | rtw.connectivity.ComponentArgs

Topics

“Create PIL Target Connectivity Configuration for MATLAB”
“Create PIL Target Connectivity Configuration for Simulink”

Introduced in R2008b

rtw.connectivity.MakefileBuilder

Configure toolchain-based build process

Description

Control toolchain-based build process for the creation of a PIL application.

Creation

Description

`rtw.connectivity.MakefileBuilder(componentArgs, targetApplicationFramework, exeExtension)` creates an object with these arguments:

- *`componentArgs`* - An `rtw.connectivity.ComponentArgs` object
- *`TargetApplicationFramework`* - An `rtw.pil.RtIOStreamApplicationFramework` object. For example, `myPIL.TargetFramework`.
- *`exeExtension`* - Name extension of executable file for target system. The extension depends on the toolchain defined by `rtw.connectivity.ConfigRegistry`. For an embedded target, the extension can be, for example, `'.elf'`, `'.abs'`, `'.sre'`, or `'.hex'`. For a Windows development computer target, the extension is `'.exe'`. For a UNIX® development computer target, the extension is empty, `''`.

If you use the template makefile approach to build the PIL application, you must provide a template makefile that includes these tokens:

- `MAKEFILEBUILDER_TGT`
- `STANDALONE_SUPPRESS_EXE`

You can create the template makefile by customizing a copy of one of the supplied ERT template makefiles, for example, `ert_unix.tmf` or `ert_vc.tmf`. You must associate the `MAKEFILEBUILDER_TGT` and `STANDALONE_SUPPRESS_EXE` tokens with corresponding makefile rules. For more information, see “Customize Template Makefiles”.

Examples

Using `rtw.connectivity.MakefileBuilder` in PIL Connectivity

For an example that shows how to use this object in setting up PIL connectivity, see:

- For MATLAB, “Processor-in-the-Loop Execution From Command Line”
- For Simulink, “Configure Processor-In-The-Loop (PIL) for a Custom Target”

See Also

`rtw.connectivity.ComponentArgs` | `rtw.pil.RtIOStreamApplicationFramework`

Topics
"Create PIL Target Connectivity Configuration for MATLAB"
"Create PIL Target Connectivity Configuration for Simulink"

Introduced in R2008b

rtw.connectivity.RtIOStreamHostCommunicator

Configure development computer communications with target processor

Description

Configure communications between your development computer and the target processor by loading and initializing a shared library that implements the `rtiostream` functions.

Creation

Description

`rtw.connectivity.RtIOStreamHostCommunicator(componentArgs, launcher, rtiostreamLib)` creates an object by using these arguments:

- *`componentArgs`* -- `rtw.connectivity.ComponentArgs` object.
- *`launcher`* -- `rtw.connectivity.Launcher` object.
- *`rtiostreamLib`* -- `rtiostream` shared library that implements the development computer part of communications between the development computer and the target processor.

The object loads and initializes the shared library.

For your development computer, Embedded Coder provides a shared library for these communication protocols:

- TCP/IP
- serial

You must provide drivers for the target processors.

For other communication protocols, for example, USB, you must provide a shared library for the development computer and drivers for the target processors.

To create your instance of `rtw.connectivity.RtIOStreamHostCommunicator`, you have these options:

- Instantiate `rtw.connectivity.RtIOStreamHostCommunicator` directly, providing custom arguments for the `rtiostream` shared library.
- Create a subclass of `rtw.connectivity.RtIOStreamHostCommunicator`. Consider this option when more complex configuration is required. For example, when:
  - The subclass `rtw.connectivity.HostTCPIPCommunicator` includes additional code to determine the number of the TCP/IP port that the executable application serves.
  - You use a subclass to specify a serial port number.
  - You specify verbose or silent operation.

Object Functions

Function	Description
`setTimeoutRecvSecs`	`hostCommunicator.setTimeoutRecvSecs(`*`timeout`*`)` sets the timeout value for reading data. You can configure data reading to time out if no new data is received for a period greater than `timeout` seconds.
`setInitCommsTimeout`	`hostCommunicator.setInitCommsTimeout(`*`timeout`*`)` sets the timeout value for initial setup of the communications channel. For some target processors, you might need to set a timeout value for initial setup of the communications channel. For example, the target processor can take a few seconds to open its side of the communications channel. If you set a nonzero timeout value, the communicator repeatedly tries to open the communications channel until the timeout value is reached.

Examples

Using `rtw.connectivity.RtIOStreamHostCommunicator` in PIL Connectivity

For an example that shows how to use this object in setting up PIL connectivity, see:

- For MATLAB, "Processor-in-the-Loop Execution From Command Line"
- For Simulink, "Configure Processor-In-The-Loop (PIL) for a Custom Target"

See Also

`rtiostream_wrapper` | `rtw.connectivity.ComponentArgs` | `rtw.connectivity.Launcher`

Topics

"Create PIL Target Connectivity Configuration for MATLAB"
"Create PIL Target Connectivity Configuration for Simulink"

Introduced in R2008b

RTW.ModelCPPDefaultClass

Create C++ class interface object for configuring model class with default model step method

---

Note The `RTW.ModelCPPDefaultClass` class is not recommended. Use a `coder.mapping.api.CodeMappingCPP` object instead. For more information, see "Programmatically Configure C++ Interface".

Syntax

`obj` `= RTW.ModelCPPDefaultClass`

Description

`obj` `= RTW.ModelCPPDefaultClass` returns a handle, *`obj`*, to a newly created object of class `RTW.ModelCPPDefaultClass`.

Output Arguments

`obj`	Handle to a newly created C++ class interface object for configuring a model class with a default model step method. The object has not yet been configured or attached to an ERT-based Simulink model.

Alternatives

Use the **Code Mappings** editor to control the C++ class interfaces that are generated for your model. Once you validate and apply your changes, you can generate code based on your C++ class interface modifications.

RTW.ModelSpecificCPrototype

Create model-specific C prototype object

Syntax

`obj = RTW.ModelSpecificCPrototype`

Description

`obj = RTW.ModelSpecificCPrototype` creates a handle, *`obj`*, to an object of class `RTW.ModelSpecificCPrototype`.

Output Arguments

`obj`	Handle to model specific C prototype object.

Examples

Create a function control object, `a`, and use it to add argument configuration information to the model:

```
% Open the rtwdemo_counter model and specify the System Target File
rtwdemo_counter
set_param(gcs,'SystemTargetFile','ert.tlc')

%% Create a function control object
a=RTW.ModelSpecificCPrototype

%% Add argument configuration information for Input and Output ports
addArgConf(a,'Input','Pointer','inputArg','const *')
addArgConf(a,'Output','Pointer','outputArg','none')

%% Attach the function control object to the model
attachToModel(a,gcs)
```

Alternatives

Use the Configure C Step Function Interface dialog box to customize the base rate C step function for a rate-based model. See “Configure Name and Arguments for Individual Step Functions”.

See Also

`addArgConf`

Topics

“Configure C Code Generation for Model Entry-Point Functions”

rtw.pil.RtIOStreamApplicationFramework

Configure target-side communications

Description

Specify target-specific libraries and source files that are required to build the executable file. The libraries and source files must include the device drivers that implement the target-side of the `rtiostream` communications channel.

Creation

Description

`appFrameObj = rtw.pil.RtIOStreamApplicationFramework(componentArgs)` returns an object that provides access to an `RTW.BuildInfo` object containing PIL-specific files (including a PIL `main` function). `rtw.connectivity.MakefileBuilder` combines these files with the PIL component libraries to create the PIL application.

Make a subclass of `rtw.pil.RtIOStreamApplicationFramework`. In addition:

- Use the `addPILMain` method to specify a `main` function, which is required to build the PIL application.
- To the `RTW.BuildInfo` object, add data that is required for the implementation of the `rtiostream` target communications interface by using provided functions:
  - Source file names - `addSourceFiles`
  - Source file paths - `addSourcePaths`
  - Include file names - `addIncludeFiles`
  - Include file paths - `addIncludePaths`
  - Libraries - `addLinkObjects`
  - Preprocessor macro definitions - `addDefines`
  - Compiler options - `addCompileFlags`
  - Linker options - `addLinkFlags`

Object Functions

Function	Description
`getComponentArgs`	`componentArgs = appFrameObj.getComponentArgs` returns the `rtw.connectivity.ComponentArgs` object associated with `appFrameObj`.
`getBuildInfo`	`buildInfo = appFrameObj.getBuildInfo` returns the `RTW.BuildInfo` object associated with `appFrameObj`.

Function	Description

`addPILMain`	To build the PIL application, a `main` function is required. Use this method to add one of the two provided files to the application framework.
	
	To specify a `main` function that is adapted for on-target PIL and suitable for most PIL implementations, enter:
	
	`appFrameObj.addPILMain('target');`
	
	To specify a `main` function that is adapted for PIL on your development computer, enter:
	
	`appFrameObj.addPILMain('host');`
	
	Alternatively, you can specify your own `main` function:
	
	`componentArgs = appFrameObj.getComponentArgs;`
	`buildInfo = appFrameObj.getBuildInfo;`
	`buildInfo.addSourcePaths(`*`pathToMyMainC`*`);`
	`buildInfo.addSourceFiles(`*`myMainC`*`);`

Examples

Using `rtw.pil.RtIOStreamApplicationFramework` in PIL Connectivity

For an example that shows how to use this object in setting up PIL connectivity, see:

- For MATLAB, “Processor-in-the-Loop Execution From Command Line”
- For Simulink, “Configure Processor-In-The-Loop (PIL) for a Custom Target”

See Also

`rtiostream_wrapper` | `rtw.connectivity.ComponentArgs`

Topics

“Create PIL Target Connectivity Configuration for MATLAB”
“Create PIL Target Connectivity Configuration for Simulink”
“Build Information Object”

Introduced in R2008b

run

Execute CGV object

Syntax

result = *cgvObj*.run()

Description

result = *cgvObj*.run() executes the model once for each input data that you added to the object. *result* is a boolean value that indicates whether the run completed without execution error. *cgvObj* is a handle to a cgv.CGV object.

After each execution of the model, the object captures and writes the following metadata to a file in the output folder:
ErrorDetails — If errors occur, the error information.
status — The execution status.
ver — Version information for MathWorks® products.
hostname — Name of computer.
dateTime — Date and time of execution.
warnings — If warnings occur, the warning messages.
username — Name of user.
runtime — The amount of time that lapsed for the execution.

Tips

- Only call run once for each cgv.CGV object.
- The cgv.CGV methods that set up the object are ignored after a call to run. See the cgv.CGV for details.
- You can call run once without first calling addInputData. However, it is recommended that you first save the required data for execution to a MAT-file, including the model inputs and parameters. Then use addInputData to pass the MAT-file to the CGV object before calling run.
- The cgv.CGV object supports callback functions that you can define and add to the cgv.CGV object. These callback functions are called during cgv.CGV.run() in the following order:

Callback function	Add to object using...	cgv.CGV.run() executes callback function...
HeaderReportFcn	addHeaderReportFcn	Before executing input data in cgv.CGV
PreExecReportFcn	addPreExecReportFcn	Before executing each input data file in cgv.CGV
PreExecFcn	addPreExecFcn	Before executing each input data file in cgv.CGV
PostExecReportFcn	addPostExecReportFcn	After executing each input data file in cgv.CGV

Callback function	Add to object using...	`cgv.CGV.run()` executes callback function...
PostExecFcn	`addPostExecFcn`	After executing each input data file in `cgv.CGV`
TrailerReportFcn	`addTrailerReportFcn`	After the input data is executed in `cgv.CGV`

See Also

Topics
"Verify Numerical Equivalence with CGV"

runValidation

Validate model-specific C++ class interface against Simulink model

Note The `RTW.ModelCPPDefaultClass` class is not recommended. Use a `coder.mapping.api.CodeMappingCPP` object instead. For more information, see "Programmatically Configure C++ Interface".

Syntax

`[status, msg] = runValidation(obj)`

Description

`[status, msg] = runValidation(obj)` runs a validation check of the specified model-specific C ++ class interface against the ERT-based Simulink model to which it is attached.

Before calling this function, you must call either `attachToModel`, to attach a function prototype to a loaded model, or `RTW.getClassInterfaceSpecification`, to get the handle to a function prototype previously attached to a loaded model.

Input Arguments

obj	Handle to a model-specific C++ class interface control object, such as a handle previously returned by `obj = RTW.ModelCPPDefaultClass on page 1-366` or `obj = getClassInterfaceSpecification(modelName)`.

Output Arguments

status	Boolean value; true for a valid configuration, false otherwise.
msg	If *status* is false, *msg* contains a character vector of information describing why the configuration is invalid.

Alternatives

Use the **Code Mappings** editor to control the C++ class interfaces that are generated for your model. Once you validate and apply your changes, you can generate code based on your C++ class interface modifications.

runValidation

Validate model-specific C function prototype against Simulink model

Syntax

`[status, msg] = runValidation(obj)`

Description

`[status, msg] = runValidation(obj)` runs a validation check of the specified model-specific C function prototype against the ERT-based Simulink model to which it is attached.

Before calling this function, you must call either `attachToModel`, to attach a function prototype to a loaded model, or `RTW.getFunctionSpecification`, to get the handle to a function prototype previously attached to a loaded model.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.ModelSpecificCPrototype` or `obj = RTW.getFunctionSpecification (modelName)`.

Output Arguments

status	True for a valid configuration; false otherwise.
msg	If *status* is false, *msg* contains a character vector explaining why the configuration is invalid.

Alternatives

Click the **Validate** button on the Configure C Step Function Interface dialog box to run a validation check of the specified model-specific C function prototype against the ERT-based Simulink model to which it is attached. See “Configure Name and Arguments for Individual Step Functions”.

See Also

Topics

“Configure C Code Generation for Model Entry-Point Functions”

setArgCategory

Set argument category for Simulink model port in model-specific C function prototype

Syntax

```
setArgCategory(obj, portName, category)
```

Description

`setArgCategory(obj, portName, category)` sets the category, `'Value'` or `'Pointer'`, of the argument corresponding to a specified Simulink model inport or outport in a specified model-specific C function prototype.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.ModelSpecificCPrototype` or `obj = RTW.getFunctionSpecification(modelName)`.
portName	Character vector specifying the unqualified name of an inport or outport in your Simulink model.
category	Character vector specifying the argument category, `'Value'` or `'Pointer'`, that you set for the specified Simulink model port.

Note If you change the argument category for an outport from `'Pointer'` to `'Value'`, it causes the argument to move to the first argument position when you call `attachToModel` or `runValidation`.

Alternatives

Use the Configure C Step Function Interface dialog box to configure C step function arguments. See “Configure Name and Arguments for Individual Step Functions”.

See Also

Topics
“Configure C Code Generation for Model Entry-Point Functions”

setArgName

Set argument name for Simulink model port in model-specific C function prototype

Syntax

```
setArgName(obj, portName, argName)
```

Description

`setArgName(obj, portName, argName)` sets the argument name corresponding to a specified Simulink model inport or outport in a specified model-specific C function prototype.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.ModelSpecificCPrototype` or `obj = RTW.getFunctionSpecification (modelName)`.
portName	Character vector specifying the name of an inport or outport in your Simulink model.
argName	Character vector specifying the argument name to set for the specified Simulink model port. The argument must be a valid C identifier.

Alternatives

Use the Configure C Step Function Interface dialog box to configure C step function arguments. See “Configure Name and Arguments for Individual Step Functions”.

See Also

Topics
“Configure C Code Generation for Model Entry-Point Functions”

setArgPosition

Set argument position for Simulink model port in model-specific C function prototype

Syntax

`setArgPosition(obj, portName, position)`

Description

`setArgPosition(obj, portName, position)` sets the position — 1 for first, 2 for second, etc. — of the argument corresponding to a specified Simulink model inport or outport in a specified model-specific C function prototype. The specified argument moves to the specified position, and other arguments shift by one position accordingly.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.ModelSpecificCPrototype` or `obj = RTW.getFunctionSpecification (modelName)`.
portName	Character vector specifying the name of an inport or outport in your Simulink model.
position	Integer specifying the argument position — 1 for first, 2 for second, etc. — to be set for the specified Simulink model port. The value must be greater than or equal to 1 and less than or equal to the number of function arguments.

Alternatives

Use the Configure C Step Function Interface dialog box to configure C step function arguments. See "Configure Name and Arguments for Individual Step Functions".

See Also

Topics
"Configure C Code Generation for Model Entry-Point Functions"

setArgQualifier

Set argument type qualifier for Simulink model inport in model-specific C function prototype

Syntax

```
setArgQualifier(obj, portName, qualifier)
```

Description

`setArgQualifier(`*`obj`*`, `*`portName`*`, `*`qualifier`*`)` sets the type qualifier — `'none'`, `'const'`, `'const *'`, or `'const * const'`— of the argument corresponding to a specified Simulink model inport in a specified model-specific C function prototype.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by *`obj`* `= RTW.ModelSpecificCPrototype` or *`obj`* `= RTW.getFunctionSpecification (`*`modelName`*`)`.
portName	Character vector specifying the name of an inport in your model.
qualifier	Character vector specifying the argument type qualifier — `'none'`, `'const'`, `'const *'`, or `'const * const'`— to be set for the specified model inport.

Note If you specify a qualifier for an outport, the code generator ignores the argument setting.

Alternatives

Use the Configure C Step Function Interface dialog box to configure C step function arguments. See "Configure Name and Arguments for Individual Step Functions".

See Also

Topics

"Configure C Code Generation for Model Entry-Point Functions"

setFunctionName

Set function name in model-specific C function prototype

Syntax

`setFunctionName(obj, fcnName, fcnType)`

Description

`setFunctionName(obj, fcnName, fcnType)` sets the step or initialization function name in the specified function control object.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.ModelSpecificCPrototype` or `obj = RTW.getFunctionSpecification(modelName)`.
fcnName	Character vector specifying a new name for the function described by the function control object. The argument must be a valid C identifier.
fcnType	Optional. Character vector specifying which function to name. Valid options are `'step'` and `'init'`. If *fcnType* is not specified, sets the step function name.

Alternatives

Use the **Step function name** field on the Configure C Step Function Interface dialog box to configure the C step function name. See “Configure Name and Arguments for Individual Step Functions”.

See Also

Topics
“Configure C Code Generation for Model Entry-Point Functions”

setMode

Specify mode of execution

Syntax

`cgvObj.setMode(connectivity)`

Description

`cgvObj.setMode(connectivity)` specifies the mode of execution for the `cgv.CGV` object, *`cgvObj`*. The default value for the execution mode is set to either `normal` or `sim`.

Input Arguments

`connectivity`

Specify mode of execution

Value	Description
`sim` or `normal` (default)	Mode of execution is normal simulation.
`sil`	Mode of execution is SIL.
`pil`	Mode of execution is PIL.

Examples

After running a `cgv.CGV` object, copy the object. Before rerunning the object, call `setMode` to change the execution mode to `sil` for an existing `cgv.CGV` object.

```
cgvModel = 'rtwdemo_cgv';
cgvObj1 = cgv.CGV(cgvModel, 'connectivity', 'sim');
cgvObj1.run();
cgvObj2 = cgvObj1.copySetup()
cgvObj2.setMode('sil');
cgvObj2.run();
```

See Also

`copySetup` | `run`

Topics

"Verify Numerical Equivalence with CGV"

setNameSpace

Set namespace for C++ function entry in code replacement table

Syntax

`setNameSpace(hEntry,nameSpace)`

Description

`setNameSpace(hEntry,nameSpace)` specifies the namespace for a C++ function entry in a code replacement table.

During code generation, if the function entry is matched, the software emits the namespace in the generated function code (for example, `std::sin(tfl_cpp_U.In1)`).

If you created the function entry by using *`hEntry`* `= RTW.TflCFunctionEntry` or *`hEntry`* `=` *`MyCustomFunctionEntry`* (did not use `registerCPPFunctionEntry`), before calling the `setNameSpace` function, enable C++ support for the function entry by calling the `enableCPP` function.

Examples

Set Namespace for Implementation Function

This example shows how to use the `setNameSpace` function to set the namespace for the `sin` implementation function to `std`.

```
fcn_entry = RTW.TflCFunctionEntry;
fcn_entry.setTflCFunctionEntryParameters( ...
                              'Key',                      'sin', ...
                              'Priority',                 100, ...
                              'ImplementationName',       'sin', ...
                              'ImplementationHeaderFile', 'cmath' );
fcn_entry.enableCPP();
fcn_entry.setNameSpace('std');
```

Input Arguments

`hEntry` — Handle to a code replacement function entry
handle

The *`hEntry`* is a handle to a code replacement function entry previously returned by one of the following:

- *`hEntry`* `= RTW.TflCFunctionEntry`
- *`hEntry`* `=` *`MyCustomFunctionEntry`*, where *`MyCustomFunctionEntry`* is a class derived from `RTW.TflCFunctionEntry`
- A call to the `registerCPPFunctionEntry` function

Example: `fcn_entry`

`nameSpace` — Specifies the namespace in which the implementation function for the C++ function entry is defined
character vector | string scalar

Example: `'std'`

See Also

`enableCPP` | `registerCPPFunctionEntry`

Topics
"Define Code Replacement Library Optimizations"

Introduced in R2010a

setObjectiveName

Package: rtw.codegenObjectives

Specify objective name

Syntax

```
setObjectiveName(objective, name)
```

Description

`setObjectiveName(objective, name)` specifies the name for the code generation objective. The Configuration Set Objectives dialog box displays the name of the objective.

Examples

Create a Custom Code Generation Objective

Create a custom objective named `Reduced RAM Example` that runs checks and verifies parameter values to confirm that the model is configured to reduce the RAM used by the generated code.

Create a file `sl_customization.m` to contain a callback function that creates the custom objective.

```
function sl_customization(cm)
%SL_CUSTOMIZATION objective customization callback

objCustomizer = cm.ObjectiveCustomizer;
index = objCustomizer.addCallbackObjFcn(@addObjectives);
objCustomizer.callbackFcn{index}();

end
```

Create and configure the objective in the `addObjectives` function. Set the name of the objective and add checks and parameters to verify. Then register the objective in the Code Generation Advisor.

```
function addObjectives

% Create the custom objective
obj = rtw.codegenObjectives.Objective('ex_ram_1');
setObjectiveName(obj, 'Reduce RAM Example');

% Add parameters to the objective
addParam(obj, 'InlineParams', 'on');
addParam(obj, 'BooleanDataType', 'on');
addParam(obj, 'OptimizeBlockIOStorage', 'on');
addParam(obj, 'EnhancedBackFolding', 'on');
addParam(obj, 'BooleansAsBitfields', 'on');

% Add additional checks to the objective
% The Code Generation Advisor automatically includes 'Check model
% configuration settings against code generation objectives' in every
% objective.
addCheck(obj, 'Identify unconnected lines, input ports, and output ports');
addCheck(obj, 'Check model and local libraries for updates');
```

```
%Register the objective
register(obj);

end
```

Input Arguments

`objective` — Code generation objective

`rtw.codegenObjectives.Objective` object

Code generation objective, specified as a `rtw.codegenObjectives.Objective` object.

Example:

`name` — Name of objective

character vector | string scalar

Name of objective, specified as a character vector or string scalar.

Example:

See Also

Topics

"Create Custom Code Generation Objectives"

Introduced in R2009a

setOutputDir

Specify folder

Syntax

```
cgvObj.setOutputDir('path')
cgvObj.setOutputDir('path', 'overwrite', 'on')
```

Description

cgvObj.setOutputDir('*path*') is an optional method that specifies a location where the object writes the output and metadata files for execution. *cgvObj* is a handle to a cgv.CGV object. *path* is the absolute or relative path to the folder. If the path does not exist, the object attempts to create the folder. If you do not call setOutputDir, the object uses the current working folder.

cgvObj.setOutputDir('*path*', 'overwrite', '*on*') includes the property and value pair to allow read-only files in the working directory to be overwritten. The default value for 'overwrite' is '*off*'.

See Also

Topics
"Verify Numerical Equivalence with CGV"

setOutputFile

Specify output data file name

Syntax

cgvObj.setOutputFile(*InputIndex*,*OutputFile*)

Description

cgvObj.setOutputFile(*InputIndex*,*OutputFile*) is an optional method that changes the default file name for the output data. *cgvObj* is a handle to a cgv.CGV object. *InputIndex* is a unique numeric identifier that specifies which output data to write to the file. The *InputIndex* is associated with specific input data.*OutputFile* is the name of the file, with or without the .mat extension.

See Also

Topics
"Verify Numerical Equivalence with CGV"

setReservedIdentifiers

Register reserved identifiers to associate with code replacement library

Syntax

```
setReservedIdentifiers(hTable,ids)
```

Description

`setReservedIdentifiers(hTable,ids)` registers reserved identifier structures in a code replacement table.

In a code replacement table, the code generator registers each function implementation name defined by a table entry as a reserved identifier. You can register additional reserved identifiers for the table on a per-header-file basis. Providing additional reserved identifiers can help prevent duplicate symbols and other identifier-related compile and link issues.

The `setReservedIdentifiers` function lets you register up to four reserved identifier structures in a code replacement table. One set of reserved identifiers can be associated with a code replacement library, while the other three (if present) must be associated with libraries named `ANSI_C`, `ISO_C`, `ISO_C++`, or `GNU`.

For information about generating a list of reserved identifiers for the code replacement library that you use to generate code, see "Reserved Identifiers and Code Replacement".

Examples

Register Reserved Identifier Structures

This example shows how to use the `setReservedIdentifiers` function to register four reserved identifier structures, for `'ANSI_C'`, `'ISO_C'`,`'ISO_C++'`, and `'My Custom CRL'`, respectively.

```
hLib = RTW.TflTable;

% Create and register CRL entries here

.
.
.

% Create and register reserved identifiers
d{1}.LibraryName = 'ANSI_C';
d{1}.HeaderInfos{1}.HeaderName = 'math.h';
d{1}.HeaderInfos{1}.ReservedIds = {'a', 'b'};
d{1}.HeaderInfos{2}.HeaderName = 'foo.h';
d{1}.HeaderInfos{2}.ReservedIds = {'c', 'd'};

d{2}.LibraryName = 'ISO_C';
d{2}.HeaderInfos{1}.HeaderName = 'math.h';
d{2}.HeaderInfos{1}.ReservedIds = {'a', 'b'};
```

```
d{2}.HeaderInfos{2}.HeaderName = 'foo.h';
d{2}.HeaderInfos{2}.ReservedIds = {'c', 'd'};

d{3}.LibraryName = 'ISO_C++';
d{3}.HeaderInfos{1}.HeaderName = 'math.h';
d{3}.HeaderInfos{1}.ReservedIds = {'a', 'b'};
d{3}.HeaderInfos{2}.HeaderName = 'foo.h';
d{3}.HeaderInfos{2}.ReservedIds = {'c', 'd'};

d{4}.LibraryName = 'My Custom CRL';
d{4}.HeaderInfos{1}.HeaderName = 'my_math_lib.h';
d{4}.HeaderInfos{1}.ReservedIds = {'y1', 'u1'};
d{4}.HeaderInfos{2}.HeaderName = 'my_oper_lib.h';
d{4}.HeaderInfos{2}.ReservedIds = {'foo', 'bar'};

setReservedIdentifiers(hLib, d);
```

Input Arguments

`hTable` — Handle to a code replacement table
handle

The *`hTable`* is a handle to a code replacement table previously returned by *`hTable`* `= RTW.TflTable`.

Example: `hLib`

ids — Specifies reserved keywords to register for library
structure

The *`ids`* is a structure specifying reserved keywords to be registered for a library. The structure must contain:

- `LibraryName` element, a character vector or string scalar that specifies `'ANSI_C'`, `'ISO_C'`, `'ISO_C++'`, `'GNU'`.
- `HeaderInfos` element, a structure or cell array of structures containing:
  - `HeaderName` element, a character vector or string scalar that specifies the header file in which the identifiers are declared.
  - `ReservedIds` element, a cell array of character vectors or string array that specifies the names of the identifiers to be registered as reserved keywords.

Example: d

See Also

Topics
"Reserved Identifiers and Code Replacement"

Introduced in R2008a

setTflCFunctionEntryParameters

Set specified parameters for function entry in code replacement table

Syntax

`setTflCFunctionEntryParameters(hEntry,varargin)`

Description

`setTflCFunctionEntryParameters(hEntry,varargin)` sets specified parameters for a function entry in a code replacement table.

Examples

Specify Parameters for Function Entry

This example shows how to use the `setTflCFunctionEntryParameters` function to set specified parameters for a code replacement function entry for `sqrt`.

```
fcn_entry = RTW.TflCFunctionEntry;
fcn_entry.setTflCFunctionEntryParameters( ...
                                 'Key',                      'sqrt', ...
                                 'Priority',                 100, ...
                                 'ImplementationName',       'sqrt', ...
                                 'ImplementationHeaderFile', '<math.h>' );
```

Input Arguments

`hEntry` — Handle to a code replacement function entry
handle

The *`hEntry`* is a handle to a code replacement function entry previously returned by *`hEntry`* `=` `RTW.TflCFunctionEntry` or *`hEntry = MyCustomFunctionEntry`*, where *`MyCustomFunctionEntry`* is a class derived from `RTW.TflCFunctionEntry`.

Example: `fcn_entry`

`varargin` — Name-value pairs of arguments for function entry
name-value pairs

Example: `'Key','sqrt'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Key','sqrt'`

AcceptExprInput — Selects whether implementation function accepts expression inputs
true | false

The *AcceptExprInput* value flags the code generator that the implementation function described by this entry accepts expression inputs. The default value is `true` if `ImplType` equals `FCN_IMPL_FUNCT` and `false` if `ImplType` equals `FCN_IMPL_MACRO`.

If the value is `true`, expression inputs are integrated into the generated code in a form similar to this form:

```
 rtY.Out1 = mySin(rtU.In1 + rtU.In2);
```

If the value is `false`, a temporary variable is generated for the expression input:

```
real_T rtb_Sum;

rtb_Sum = rtU.In1 + rtU.In2;
rtY.Out1 = mySin(rtb_Sum);
```

Example: `'AcceptExprInput',true`

AdditionalHeaderFiles — Specifies additional header files for table entry
{} (default) | array of character vectors | string array

The *AdditionalHeaderFiles* value specifies additional header files for a code replacement table entry. The vectors can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalHeaderFiles',{}`

AdditionalIncludePaths — Specifies additional include paths for table entry
{} (default) | array of character vectors | string array

The *AdditionalIncludePaths* value specifies the full path of additional include paths for a code replacement entry. The character vectors or string array can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalIncludePaths',{}`

AdditionalLinkObjs — Specifies additional link objects for table entry
{} (default) | array of character vectors | string array

The *AdditionalLinkObjs* value specifies additional link objects for a code replacement table entry. The character vectors or string array can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalLinkObjs',{}`

AdditionalLinkObjsPaths — Specifying additional link object paths for table entry
{} (default) | array of character vectors | string array

The *AdditionalLinkObjsPaths* value specifies the full path of additional link object paths for a code replacement entry. The character vectors or string array can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the

MATLAB workspace or as a MATLAB function in the search path that returns a character vector. The default is `{}`.

Example: `'AdditionalLinkObjsPaths',{}`

`AdditionalSourceFiles` — Specifies additional source files for table entry
`{}` (default) | array of character vectors | string array

The *`AdditionalSourceFiles`* value specifies additional source files for a code replacement table entry. The character vectors or string array can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalSourceFiles',{}`

`AdditionalSourcePaths` — Specifies additional source paths for table entry
`{}` (default) | array of character vectors | string array

The *`AdditionalSourcePaths`* value specifies the full path of additional source paths for a code replacement entry. The character vectors or string array can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalSourcePaths',{}`

`AdditionalCompileFlags` — Specifies additional compiler flags for table entry
`{}` (default) | array of character vectors | string array

The *`AdditionalCompileFlags`* value specifies additional flags required to compile the source files defined for a code replacement table entry. The default is `{}`.

Example: `'AdditionalCompileFlags',{}`

`AdditionalLinkFlags` — Specifies additional linker flags for table entry
`{}` (default) | array of character vectors | string array

The *`AdditionalLinkFlags`* value specifies additional flags required to link the compiled files for a code replacement table entry.

Example: `'AdditionalLinkFlags',{}`

`ArrayLayout` — Specifies layout of array storage for table entry
`'COLUMN_MAJOR'` (default) | `'ROW_MAJOR'` | `'COLUMN_AND_ROW'`

The *`ArrayLayout`* value specifies the order of array elements in memory supported by the replacement implementation. By default, the replacement implementation supports column-major data layout. For `ROW-MAJOR`, the replacement implementation supports row-major data layout. For `COLUMN_AND_ROW`, the replacement implementation supports column-major and row-major data layouts.

Example: `'ArrayLayout','ROW_MAJOR'`

`EntryInfoAlgorithm` — Specifies computation or approximation method to match for table entry
`'RTW_DEFAULT'` | `'RTW_NEWTON_RAPHSON'` | `'RTW_CORDIC'` | `'RTW_UNSPECIFIED'`

The *`EntryInfoAlgorithm`* value specifies a computation or approximation method, configured for the specified math function, that must be matched in order for function replacement to occur. Code

replacement libraries support function replacement based on computation or approximation method for the math functions `rSqrt`, `sin`, `cos`, and `sincos`. The valid arguments for each supported function are listed in the table.

Function	Argument	Meaning
`rSqrt`	`RTW_DEFAULT`	Match the default computation method, `Exact`
	`RTW_NEWTON_RAPHSON`	Match the `Newton-Raphson` computation method
	`RTW_UNSPECIFIED`	Match a computation method

`sin`
`cos`
`sincos`
	`RTW_CORDIC`	Match the `CORDIC` approximation method

	`RTW_DEFAULT`	Match the default approximation method, `None`
	`RTW_UNSPECIFIED`	Match an approximation method

Example: `'EntryInfoAlgorithm','RTW_DEFAULT'`

`GenCallback` — Specifies callback that follows code generation
`''` (default) | `'RTW.copyFileToBuildDir'`

The *`GenCallback`* specifies a callback that follows code generation. If you specify `'RTW.copyFileToBuildDir'`, and if this function entry is matched and used, the code generator calls function `RTW.copyFileToBuildDir` after code generation. This callback function copies additional header, source, or object files that you have specified for this function entry to the build folder.

Example: `'GenCallback',''`

`ImplementationHeaderFile` — Specifies the name of the header file that declares the implementation function
`''` (default) | character vector | string scalar

The *`ImplementationHeaderFile`* value specifies the name of the header file that declares the implementation function, for example, `'<math.h>'`. The character vector or string scalar can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'ImplementationHeaderFile',''`

`ImplementationHeaderPath` — Specifies path to implementation header file
`''` (default) | character vector | string scalar

The *`ImplementationHeaderPath`* value specifies the full path to the implementation header file. The character vector or string scalar can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'ImplementationHeaderPath',''`

`ImplementationName` — Specifies name of implementation function
`''` (default) | character vector | string scalar

The *`ImplementationName`* value specifies the name of the implementation function, for example, `'sqrt'`, which can match or differ from the `Key` name.

Example: `'ImplementationName',''`

`ImplementationSourceFile` — Specifies name of implementation source file
`''` (default) | character vector | string scalar

The *`ImplementationSourceFile`* value specifies the name of the implementation source file. The character vector or string scalar can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'ImplementationSourceFile',''`

`ImplementationSourcePath` — Specifies path to implementation source file
`''` (default) | character vector | string scalar

The *`ImplementationSourcePath`* value specifies the full path to the implementation source file. The character vector or string scalar can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'ImplementationSourcePath',''`

`ImplType` — Specifies the type of entry
`'FCN_IMPL_FUNCT'` (default) | `'FCN_IMPL_MACRO'`

Use `FCN_IMPL_FUNCT` for function or `FCN_IMPL_MACRO` for macro.

Example: `'ImplType','FCN_IMPL_FUNCT'`

Key — Specifies name of function to replace
character vector | string scalar

The Key value specifies the name of the function to replace. The name must match a function name listed in "Code You Can Replace from MATLAB Code" or "Code You Can Replace From Simulink Models".

Example: `'Key','sqrt'`

`Priority` — Specifies the search priority for function entry
`100` (default) | integer 0..100

The *`Priority`* value specifies the search priority for the function entry, relative to other entries of the same function name and conceptual argument list within this table. Highest priority is 0, and lowest priority is 100. If the table provides two implementations for a function, the implementation with the higher priority shadows the one with the lower priority.

Example: `'Priority',100`

RoundingModes — Specifying rounding modes supported by implementation function
`'RTW_ROUND_UNSPECIFIED'` (default) | `'RTW_ROUND_FLOOR'` | `'RTW_ROUND_CEILING'` | `'RTW_ROUND_ZERO'` | `'RTW_ROUND_NEAREST'` | `'RTW_ROUND_NEAREST_ML'` | `'RTW_ROUND_SIMPLEST'` | `'RTW_ROUND_CONV'` | array of character vectors | string array

The *`RoundingModes`* value specifies one or more rounding modes supported by the implementation function.

Example: `'RoundingModes',{'RTW_ROUND_UNSPECIFIED'}`

SaturationMode — Specifying saturation mode supported by implementation function
'RTW_SATURATE_UNSPECIFIED' (default) | 'RTW_SATURATE_ON_OVERFLOW' | 'RTW_WRAP_ON_OVERFLOW'

The *SaturationMode* value specifies the saturation mode supported by the implementation function.

Example: 'SaturationMode','RTW_SATURATE_UNSPECIFIED'

SideEffects — Specifies whether to attempt to optimize away the implementation function
false (default) | true

The *SideEffects* value flags the code generator not to optimize away the implementation function described by this entry. This parameter applies to implementation functions that return void but are not to be optimized away, such as a memcpy implementation or an implementation function that accesses global memory values. For those implementation functions only, you must include this parameter and specify the value true.

Example: 'SideEffects',false

StoreFcnReturnInLocalVar — Specifies whether to store the implementation function regardless expression folding settings
false (default) | true

The *StoreFcnReturnInLocalVar* value flags the code generator that the return value of the implementation function described by this entry must be stored in a local variable regardless of other expression folding settings. If the value is false, other expression folding settings determine whether the return value is folded. Storing function returns in a local variable can increase the clarity of generated code. This example shows code generated with expression folding:

```
void sw_step(void)
{
    if (ssub(sadd(sw_U.In1, sw_U.In2), sw_U.In3) <=
        smul(ssub(sw_U.In4, sw_U.In5),sw_U.In6)) {
      sw_Y.Out1 = sw_U.In7;
    } else {
       sw_Y.Out1 = sw_U.In8;
    }
}
```

With StoreFcnReturnInLocalVar set to true, the generated code is potentially easier to understand and debug:

```
void sw_step(void)
{
    real32_T rtb_Switch;
    real32_T hoistedExpr;
    ......
    rtb_Switch = sadd(sw_U.In1, sw_U.In2);
    rtb_Switch = ssub(rtb_Switch, sw_U.In3);
    hoistedExpr = ssub(sw_U.In4, sw_U.In5);
    hoistedExpr = smul(hoistedExpr, sw_U.In6);
    if (rtb_Switch <= hoistedExpr) {
       sw_Y.Out1 = sw_U.In7;
    } else {
       sw_Y.Out1 = sw_U.In8;
    }
}
```

Example: `'StoreFcnReturnInLocalVar',false`

See Also

`addAdditionalHeaderFile` | `addAdditionalIncludePath` | `addAdditionalLinkObj` | `addAdditionalLinkObjPath` | `addAdditionalSourceFile` | `addAdditionalSourcepath`

Topics

"Define Code Replacement Library Optimizations"
"Code You Can Replace from MATLAB Code"
"Code You Can Replace From Simulink Models"

Introduced in R2007b

setTflCOperationEntryParameters

Set specified parameters for operator entry in code replacement table

Syntax

```
setTflCOperationEntryParameters(hEntry,varargin)
```

Description

`setTflCOperationEntryParameters(hEntry,varargin)` sets specified parameters for an operator entry in a code replacement table.

Examples

Set Parameters for Addition Operator Entry

This example shows how to use the `setTflCOperationEntryParameters` function to set parameters for a code replacement operator entry for `uint8` addition that matches a cast-after-sum algorithm.

```
op_entry = RTW.TflCOperationEntry;
op_entry.setTflCOperationEntryParameters( ...
    'Key',                      'RTW_OP_ADD', ...
    'EntryInfoAlgorithm',       'RTW_CAST_AFTER_OP', ...
    'Priority',                 90, ...
    'SaturationMode',           'RTW_SATURATE_UNSPECIFIED', ...
    'RoundingModes',            {'RTW_ROUND_UNSPECIFIED'}, ...
    'ImplementationName',       'u8_add_u8_u8', ...
    'ImplementationHeaderFile', 'u8_add_u8_u8.h', ...
    'ImplementationSourceFile', 'u8_add_u8_u8.c');
```

Set Parameters for Fixed-Point Division Operator Entry

This example shows how to use the `setTflCOperationEntryParameters` function to set parameters for a code replacement operator entry for fixed-point `int16` division. The table entry specifies a net scaling between the operator inputs and output to map a range of slope and bias values to a replacement operation.

```
op_entry = RTW.TflCOperationEntryGenerator_NetSlope;
op_entry.setTflCOperationEntryParameters( ...
    'Key',                      'RTW_OP_DIV', ...
    'Priority',                 90, ...
    'SaturationMode',           'RTW_WRAP_ON_OVERFLOW', ...
    'RoundingModes',            {'RTW_ROUND_CEILING'}, ...
    'NetSlopeAdjustmentFactor',   1.0, ...
    'NetFixedExponent',           0.0, ...
    'ImplementationName',       's16_div_s16_s16', ...
```

```
    'ImplementationHeaderFile', 's16_div_s16_s16.h', ...
    'ImplementationSourceFile', 's16_div_s16_s16.c' );
```

Set Parameters for Fixed-Point Addition Operator Entry

This example shows how to use the `setTflCOperationEntryParameters` function to set parameters for a code replacement operator entry for fixed-point `uint16` addition that matches a cast-after-sum algorithm. The parameters 'SlopesMustBeTheSame' and 'MustHaveZeroNetBias' must be set to `true` to specify equal slope and zero net bias across operator inputs and output. This maps relative slope and bias values (rather than a specific slope and bias combination) to a replacement operation.

```
op_entry = RTW.TflCOperationEntryGenerator;
op_entry.setTflCOperationEntryParameters( ...
    'Key',                      'RTW_OP_ADD', ...
    'EntryInfoAlgorithm',       'RTW_CAST_AFTER_OP', ...
    'Priority',                  90, ...
    'SaturationMode',            'RTW_WRAP_ON_OVERFLOW', ...
    'RoundingModes',             {'RTW_ROUND_UNSPECIFIED'}, ...
    'SlopesMustBeTheSame',       true, ...
    'MustHaveZeroNetBias',       true, ...
    'ImplementationName',        'u16_add_SameSlopeZeroBias', ...
    'ImplementationHeaderFile', 'u16_add_SameSlopeZeroBias.h', ...
    'ImplementationSourceFile', 'u16_add_SameSlopeZeroBias.c');
```

Input Arguments

`hEntry` — Handle to a code replacement table entry

handle

The *`hEntry`* is a handle to a code replacement table entry previously returned by one of the class instantiations in the table.

Class Instantiation	Support
`hEntry` `= RTW.TflCOperationEntry;`	Supports operator replacement.
`hEntry` `= RTW.TflCOperationEntryGenerator;`	Provides parameters for fixed-point addition and subtraction that are not available in `RTW.TflCOperationEntry` (`SlopesMustBeTheSame` and `MustHaveZeroNetBias`).
`hEntry` `= RTW.TflCOperationEntryGenerator_NetSlope;`	Provides net slope parameters for fixed-point multiplication and division that are not available in `RTW.TflCOperationEntry` (`NetSlopeAdjustmentFactor` and `NetFixedExponent`).
`hEntry` `= RTW.TflBlasEntryGenerator;`	Supports replacement of nonscalar operators with MathWorks BLAS functions.

Class Instantiation	Support
hEntry = RTW.TflCBlasEntryGenerator;	Supports replacement of nonscalar operators with ANSI/ISO® C BLAS functions.
hEntry = *MyCustomOperationEntry*; (where *MyCustomOperationEntry* is a class derived from RTW.TflCOperationEntry)	Supports operator replacement using custom code replacement table entries.

If you want to specify SlopesMustBeTheSame or MustHaveZeroNetBias for your operator entry, instantiate your table entry using *hEntry* = RTW.TflCOperationEntryGenerator rather than *hEntry* = RTW.TflCOperationEntry. If you want to use NetSlopeAdjustmentFactor and NetFixedExponent, instantiate your table entry by using *hEntry* = RTW.TflCOperationEntryGenerator_NetSlope.

Example: op_entry

varargin — Name-value pairs of arguments for operation entry
name-value pairs

Example: 'Key','RTW_OP_ADD'

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'Key','RTW_OP_ADD'

AcceptExprInput — Specifies whether implementation operation accepts expression inputs
true | false

The *AcceptExprInput* value flags the code generator that the implementation function described by this entry accepts expression inputs. The default value is true if ImplType equals FCN_IMPL_FUNCT and false if ImplType equals FCN_IMPL_MACRO.

If the value is true, expression inputs are integrated into the generated code in a form similar to this form:

```
 rtY.Out1 = myAdd(rtU.In1, rtU.In2 * rtU.In3);
```

If the value is false, a temporary variable is generated for the expression input:

```
real_T temp;

temp = rtU.In2 * rtU.In3;
rtY.Out1 = myAdd(rtU.In1, temp);
```

Example: 'AcceptExprInput',true

AdditionalHeaderFiles — Specifies additional header files for table entry
{} (default) | array of character vectors | string array

The *AdditionalHeaderFiles* value specifies additional header files for a code replacement table entry. The character vectors or string array can include tokens. For example, in the token $mytoken

$, mytoken is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: 'AdditionalHeaderFiles',{}

AdditionalIncludePaths — Specifies additional include paths for table entry
{} (default) | array of character vectors | string array

The *AdditionalIncludePaths* value specifies the full path of additional include paths for a code replacement entry. The character vectors or string array can include tokens. For example, in the token $mytoken$, mytoken is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: 'AdditionalIncludePaths',{}

AdditionalLinkObjs — Specifies additional link objects for table entry
{} (default) | array of character vectors | string array

The *AdditionalLinkObjs* value specifies additional link objects for a code replacement table entry. The character vectors or string array can include tokens. For example, in the token $mytoken$, mytoken is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: 'AdditionalLinkObjs',{}

AdditionalLinkObjsPaths — Specifies additional link object paths for table entry
{} (default) | array of character vectors | string array

The *AdditionalLinkObjsPaths* value specifies the full path of additional link object paths for a code replacement entry. The character vectors or string array can include tokens. For example, in the token $mytoken$, mytoken is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: 'AdditionalLinkObjsPaths',{}

AdditionalSourceFiles — specifies additional source files for table entry
{} (default) | array of character vectors | string array

The *AdditionalSourceFiles* value specifies additional source files for a code replacement table entry. The character vectors or string array can include tokens. For example, in the token $mytoken $, mytoken is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: 'AdditionalSourceFiles',{}

AdditionalSourcePaths — Specifies additional source paths for table entry
{} (default) | array of character vectors | string array

The *AdditionalSourcePaths* value specifies the full path of additional source paths for a code replacement entry. The character vectors or string array can include tokens. For example, in the token $mytoken$, mytoken is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: 'AdditionalSourcePaths',{}

AdditionalCompileFlags — Specifies additional compiler flags for table entry
{} (default) | array of character vectors | string array

The *AdditionalCompileFlags* value specifies additional flags required to compile the source files defined for a code replacement table entry.

Example: 'AdditionalCompileFlags',{}

AdditionalLinkFlags — Specifies additional linker flags for table entry
{} (default) | array of character vectors | string array

The *AdditionalLinkFlags* value specifies additional flags required to link the compiled files for a code replacement table entry.

Example: 'AdditionalLinkFlags',{}

AllowShapeAgnosticMatch — Enables matrix matches based on the total number of elements rather than specific matrix shape
false (default) | true

The *AllowShapeAgnosticMatch* value enables code replacement match based on total number of elements rather than specific matrix shape for matrices that are contiguously allocated in memory. For more information, see “Allow Shape Agnostic Match”.

Example: 'AllowShapeAgnosticMatch', false

ArrayLayout — Specifies layout of array storage for table entry
'COLUMN_MAJOR' (default) | 'ROW_MAJOR' | 'COLUMN_AND_ROW'

The *ArrayLayout* value specifies the order of array elements in memory supported by the replacement implementation. By default, the replacement implementation supports column-major data layout. For ROW-MAJOR, the replacement implementation supports row-major data layout. For COLUMN_AND_ROW, the replacement implementation supports column-major and row-major data layouts.

Example: 'ArrayLayout','ROW_MAJOR'

EntryInfoAlgorithm — Specifies math algorithm to match for table entry
'RTW_CAST_BEFORE_OP' (default) | 'RTW_CAST_AFTER_OP'

The *EntryInfoAlgorithm* value specifies the algorithm for the specified math function that must be matched for operator replacement to occur. Code replacement libraries support replacement based on the algorithm for math operations RTW_OP_ADD and RTW_OP_MINUS. Valid arguments for the supported operations are listed in the table. The arguments have the same meaning for both operations.

Argument	Meaning
RTW_CAST_BEFORE_OP	Before performing the operation, type cast input values to the output data type. If the output type cannot exactly represent the input values, losses can occur as a result of the cast to the output type. Additional loss can occur when the result of the operation is cast to the final output type.

Argument	Meaning
RTW_CAST_AFTER_OP	Compute the ideal result of the operation of inputs. Then, type cast the result to the output data type. Loss occurs during the type cast. This algorithm behaves similarly to the C language except when the signedness of the operands does not match. For example, when you add a signed long operation to an unsigned long operand, standard C language rules convert the signed long operand to an unsigned long operand. The result is a value that is not ideal.

Example: `'EntryInfoAlgorithm','RTW_CAST_AFTER_OP'`

`GenCallback` — Specifies callback that follows code generation
`''` (default) | `'RTW.copyFileToBuildDir'`

The *`GenCallback`* specifies a callback that follows code generation. If you specify `'RTW.copyFileToBuildDir'`, and if this operation entry is matched and used, the code generator calls function `RTW.copyFileToBuildDir` after code generation. This callback function copies additional header, source, or object files that you have specified for this entry to the build folder.

Example: `'GenCallback','RTW.copyFileToBuildDir'`

`ImplementationHeaderFile` — Specifies name of header file that declares implementation operation
`''` (default) | character vector | string scalar

The *`ImplementationHeaderFile`* value specifies the name of the header file that declares the implementation function. The character vector or string scalar can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'ImplementationHeaderFile','s32_mul.h'`

`ImplementationHeaderPath` — Specifies path to implementation header file
`''` (default) | character vector | string scalar

The *`ImplementationHeaderPath`* value specifies the full path to the implementation header file. The character vector or string scalar can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'ImplementationHeaderPath', fullfile('$MATLAB_ROOT', 'crl')`

`ImplementationName` — Specifies name of implementation function
`''` (default) | character vector | string scalar

The *`ImplementationName`* value specifies the name of the implementation function, which can match or differ from the `Key` name.

Example: `'ImplementationName','s32_mul_s32_s32_sat'`

`ImplementationSourceFile` — specifies name of implementation source file
`''` (default) | character vector | string scalar

The *ImplementationSourceFile* value specifies the name of the implementation source file. The character vector or string scalar can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'ImplementationSourceFile','s32_mul.c'`

ImplementationSourcePath — Specifies path to implementation source file
`''` (default) | character vector | string scalar

The *ImplementationSourcePath* value specifies the full path to the implementation source file. The character vector or string scalar can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'ImplementationSourcePath', fullfile('$MATLAB_ROOT', 'crl')`

ImplType — Specifies whether the table entry is for an implementation function or macro
`'FCN_IMPL_FUNCT'` (default) | `'FCN_IMPL_MACRO'`

The *ImplType* value specifies the type of table entry. Use `FCN_IMPL_FUNCT` for function or `FCN_IMPL_MACRO` for macro.

Example: `'ImplType','FCN_IMPL_FUNCT'`

Key — Specifies key for operator to replace
character vector | string scalar

The *Key* value specifies the key for the operator to replace. The key must match an operator key listed in “Code You Can Replace from MATLAB Code” or “Code You Can Replace From Simulink Models”.

Example: `'Key','RTW_OP_ADD'`

MustHaveZeroNetBias — Specifies net bias requirement for conceptual arguments of Add/Minus entries
`false` (default) | `true`

The *MustHaveZeroNetBias* value specifies whether a replacement match requires that the net bias for conceptual arguments of Add/Minus entries is zero. For the Add/Minus of fixed-point operator inputs and output this parameter must be set to `true`. Instantiate the entry by using *hEntry* = `RTW.TflCOperationEntryGenerator` rather than *hEntry* = `RTW.TflCOperationEntry`.

For Mul/Div/MulDiv/Shift/Cast entries:

- The code generator ignores the value of this argument.
- The bias of the conceptual arguments for Mul/Div/MulDiv/Shift/Cast entries must be zero for a replacement match to occur.

Example: `'MustHaveZeroNetBias',true`

NetFixedExponent — Specifies fixed exponent part of net slope for fixed-point conceptual arguments of Mul/Div/MulDiv/Shift/Cast entries
`0` (default) | `numeric scalar`

The *NetSlopeAdjustmentFactor* value specifies the fixed exponent (E) part of the net slope ($F2^E$, for example, -3.0) for fixed-point conceptual arguments required for a replacement match to occur for

Mul/Div/MulDiv/Shift/Cast entries. Instantiate an entry by using *`hEntry`* `= RTW.TflCOperationEntryGenerator_NetSlope` rather than *`hEntry`* `= RTW.TflCOperationEntry`.

For Add/Minus entries:

- The code generator ignores the value of this argument.
- The slope adjustment factor part of the net slope of the conceptual arguments for Add/Minus entries must be zero for a replacement match to occur.

Example: `'NetFixedExponent', -3.0`

`NetSlopeAdjustmentFactor` — Specifies slope adjustment part of net slope requirement for fixed-point conceptual arguments of Mul/Div/MulDiv/Shift/Cast entries
`1` (default) | numeric scalar

The *`NetSlopeAdjustmentFactor`* value specifies the slope adjustment part of the net slope ($F2^E$, for example, 1.0) for fixed-point conceptual arguments required for a replacement match to occur for Mul/Div/MulDiv/Shift/Cast entries. Instantiate an entry by using *`hEntry`* `= RTW.TflCOperationEntryGenerator_NetSlope` rather than *`hEntry`* `= RTW.TflCOperationEntry`.

For Add/Minus entries:

- The code generator ignores the value of this argument.
- The slope adjustment factor part of the net slope of the conceptual arguments for Add/Minus entries must be zero for a replacement match to occur.

Example: `'NetSlopeAdjustmentFactor',1.5`

`Priority` — Specifies the search priority of operator entry
`100` (default) | integer value 0..100

The *`Priority`* value specifies the search priority of the operation entry, relative to other entries of the same operation name and conceptual argument list within this table. Highest priority is 0, and lowest priority is 100. If the table provides two implementations for an operation, the implementation with the higher priority shadows the one with the lower priority.

Example: `'Priority',100`

RoundingModes — Specifies rounding modes supported by implementation function
`'RTW_ROUND_UNSPECIFIED'` (default) | `'RTW_ROUND_FLOOR'` | `'RTW_ROUND_CEILING'` | `'RTW_ROUND_ZERO'` | `'RTW_ROUND_NEAREST'` | `'RTW_ROUND_NEAREST_ML'` | `'RTW_ROUND_CONV'` | `'RTW_ROUND_SIMPLEST'` | array of character vectors | string array

The *`RoundingModes`* value specifies one or more rounding modes supported by the implementation function.

Example: `'RoundingModes',{'RTW_ROUND_UNSPECIFIED'}`

SaturationMode — specifies saturation mode supported by implementation function
`'RTW_SATURATE_UNSPECIFIED'` (default) | `'RTW_SATURATE_ON_OVERFLOW'` | `'RTW_WRAP_ON_OVERFLOW'` | character vector | string scalar

The *`SaturationMode`* value specifies the saturation mode supported by the implementation function.

Example: 'SaturationMode','RTW_SATURATE_UNSPECIFIED'

SideEffects — Specifies whether to attempt to optimize away the implementation function

false (default) | true

The *SideEffects* value flags the code generator not to optimize away the implementation function described by this entry. This parameter applies to implementation functions that return void but are not to be optimized away, such as a memcpy implementation or an implementation function that accesses global memory values. For those implementation functions only, you must include this parameter and specify the value true.

Example: 'SideEffects',false

SlopesMustBeTheSame — Specifies slope requirement for conceptual arguments of Mul/Div/MulDiv/Shift/Cast entries

false (default) | true

The *SlopesMustBeTheSame* value specifies whether a replacement match requires that the slope is the same for conceptual arguments of Mul/Div/MulDiv/Shift/Cast entries. Instantiate the entry by using *hEntry* = RTW.TflCOperationEntryGenerator rather than *hEntry* = RTW.TflCOperationEntry.

For Add/Minus entries:

- The code generator ignores the value of this argument.
- This parameter must be set to true. When set to true, the slopes of the conceptual arguments are equal for a replacement match to occur.

Example: 'SlopesMustBeTheSame',true

StoreFcnReturnInLocalVar — Specifies whether to store the implementation function regardless expression folding settings

false (default) | true

The *StoreFcnReturnInLocalVar* value flags the code generator that the return value of the implementation function described by this entry must be stored in a local variable regardless of other expression folding settings. If the value is false, other expression folding settings determine whether the return value is folded. Storing function returns in a local variable can increase the clarity of generated code. This example shows code generated with expression folding:

```
void sw_step(void)
{
    if (ssub(sadd(sw_U.In1, sw_U.In2), sw_U.In3) <=
        smul(ssub(sw_U.In4, sw_U.In5),sw_U.In6)) {
      sw_Y.Out1 = sw_U.In7;
    } else {
       sw_Y.Out1 = sw_U.In8;
    }
}
```

With StoreFcnReturnInLocalVar set to true, the generated code is potentially easier to understand and debug:

```
void sw_step(void)
{
```

```
    real32_T rtb_Switch;
    real32_T hoistedExpr;
    ......
    rtb_Switch = sadd(sw_U.In1, sw_U.In2);
    rtb_Switch = ssub(rtb_Switch, sw_U.In3);
    hoistedExpr = ssub(sw_U.In4, sw_U.In5);
    hoistedExpr = smul(hoistedExpr, sw_U.In6);
    if (rtb_Switch <= hoistedExpr) {
       sw_Y.Out1 = sw_U.In7;
    } else {
       sw_Y.Out1 = sw_U.In8;
    }
}
```

Example: `'StoreFcnReturnInLocalVar',false`

See Also

`addAdditionalHeaderFile` | `addAdditionalIncludePath` | `addAdditionalLinkObj` | `addAdditionalLinkObjPath` | `addAdditionalSourceFile` | `addAdditionalSourcepath`

Topics

"Define Code Replacement Library Optimizations"
"Scalar Operator Code Replacement"
"Addition and Subtraction Operator Code Replacement"
"Small Matrix Operation to Processor Code Replacement"
"Code You Can Replace from MATLAB Code"
"Code You Can Replace From Simulink Models"

Introduced in R2007b

setTflCSemaphoreEntryParameters

Set specified parameters for semaphore entry in code replacement table

Syntax

`setTflCSemaphoreEntryParameters(hEntry,varargin)`

Description

`setTflCSemaphoreEntryParameters(hEntry,varargin)` sets specified parameters for a semaphore entry in a code replacement table.

Examples

Specify Semaphore Initialization Parameters for Table Entry

This example shows how to use the `setTflCSemaphoreEntryParameters` function to set specified parameters for a code replacement table entry for a semaphore initialization replacement.

```
sem_entry = RTW.TflCSemaphoreEntry;
sem_entry.setTflCSemaphoreEntryParameters( ...
    'Key',                       'RTW_SEM_INIT', ...
    'Priority',                  100, ...
    'ImplementationName',        'mySemCreate', ...
    'ImplementationHeaderFile', 'mySem.h', ...
    'ImplementationSourceFile', 'mySem.c', ...
    'GenCallback',               'RTW.copyFileToBuildDir', ...
    'SideEffects',               true);
```

Input Arguments

`hEntry` — Handle to semaphore entry
handle

The *`hEntry`* is a handle to a code replacement library semaphore entry previously returned by *`hEntry`* `= RTW.TflCSemaphoreEntry;`.

Example: `sem_entry`

`varargin` — Name-value pairs of arguments for function entry
name-value pairs

Example: `'Key','RTW_SEM_INIT'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Key','RTW_SEM_INIT'`

`AcceptExprInput` — Specifies whether implementation function accepts expression inputs
`true` | `false`

The *`AcceptExprInput`* value flags the code generator that the implementation function described by this entry accepts expression inputs. The default value is `true` if `ImplType` equals `FCN_IMPL_FUNCT` and `false` if `ImplType` equals `FCN_IMPL_MACRO`.

If the value is `true`, expression inputs are integrated into the generated code in a form similar to this form:

```
 rtY.Out1 = mySin(rtU.In1 + rtU.In2);
```

If the value is `false`, a temporary variable is generated for the expression input:

```
real_T rtb_Sum;

rtb_Sum = rtU.In1 + rtU.In2;
rtY.Out1 = mySin(rtb_Sum);
```

Example: `'AcceptExprInput',true`

`AdditionalHeaderFiles` — Specifies additional header files for table entry
`{}` (default) | array of character vectors | string array

The *`AdditionalHeaderFiles`* value specifies additional header files for a code replacement table entry. The character vectors or string array can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalHeaderFiles',{}`

`AdditionalIncludePaths` — Specifies additional include paths for table entry
`{}` (default) | array of character vectors | string array

The *`AdditionalIncludePaths`* value specifies the full path of additional include paths for a code replacement entry. The character vectors or string array can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalIncludePaths',{}`

`AdditionalLinkObjs` — Specifies additional link objects for table entry
`{}` (default) | array of character vectors | string array

The *`AdditionalLinkObjs`* value specifies additional link objects for a code replacement table entry. The character vectors or string array can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalLinkObjs',{}`

`AdditionalLinkObjsPaths` — Specifies additional link object paths for table entry
`{}` (default) | array of character vectors | string array

The *`AdditionalLinkObjsPaths`* value specifies the full path of additional link object paths for a code replacement entry. The character vectors or string array can include tokens. For example, in the

token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalLinkObjsPaths',{}`

`AdditionalSourceFiles` — specifies additional source files for table entry
`{}` (default) | array of character vectors | string array

The *`AdditionalSourceFiles`* value specifies additional source files for a code replacement table entry. The character vectors or string array can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalSourceFiles',{}`

`AdditionalSourcePaths` — Specifies additional source paths for table entry
`{}` (default) | array of character vectors | string array

The *`AdditionalSourcePaths`* value specifies the full path of additional source paths for a code replacement entry. The character vectors or string array can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector. The default is `{}`.

Example: `'AdditionalSourcePaths',{}`

`AdditionalCompileFlags` — Specifies additional compiler flags for table entry
`{}` (default) | array of character vectors | string array

The *`AdditionalCompileFlags`* value specifies additional flags required to compile the source files defined for a code replacement table entry.

Example: `'AdditionalCompileFlags',{}`

`AdditionalLinkFlags` — Specifies additional linker flags for table entry
`{}` (default) | array of character vectors | string array

The *`AdditionalLinkFlags`* value specifies additional flags required to link the compiled files for a code replacement table entry.

Example: `'AdditionalLinkFlags',{}`

`GenCallback` — Specifies callback that follows code generation
`''` (default) | `'RTW.copyFileToBuildDir'`

The *`GenCallback`* specifies a callback that follows code generation. If you specify `'RTW.copyFileToBuildDir'`, and if this function entry is matched and used, the function `RTW.copyFileToBuildDir` is called after code generation. This callback function copies additional header, source, or object files that you have specified for this function entry to the build folder.

Example: `'GenCallback',''`

`ImplementationHeaderFile` — Specifies name of header file that declares implementation function
`''` (default) | character vector | string scalar

The *`ImplementationHeaderFile`* value specifies the name of the header file that declares the implementation function. The character vector or string scalar can include tokens. For example, in

the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'ImplementationHeaderFile','<math.h>'`

ImplementationHeaderPath — Specifies path to implementation header file
`''` (default) | character vector | string scalar

The *ImplementationHeaderPath* value specifies the full path to the implementation header file. The character vector or string scalar can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'ImplementationHeaderPath',''`

ImplementationName — Specifies name of implementation function
`''` (default) | character vector | string scalar

The *ImplementationName* value specifies the name of the implementation function, which can match or differ from the `Key` name.

Example: `'ImplementationName','sqrt'`

ImplementationSourceFile — specifies name of implementation source file
`''` (default) | character vector | string scalar

The *ImplementationSourceFile* value specifies the name of the implementation source file. The character vector or string scalar can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'ImplementationSourceFile',''`

ImplementationSourcePath — Specifies path to implementation source file
`''` (default) | character vector | string scalar

The *ImplementationSourcePath* value specifies the full path to the implementation source file. The character vector or string scalar can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'ImplementationSourcePath',''`

ImplType — Specifies the type of table entry
`'FCN_IMPL_FUNCT'` (default) | `'FCN_IMPL_MACRO'`

The *ImplType* value specifies the type of table entry. Use `FCN_IMPL_FUNCT` for function or `FCN_IMPL_MACRO` for macro.

Example: `'ImplType','FCN_IMPL_FUNCT'`

Key — Specifies key for operator to replace
character vector | string scalar

The *Key* value specifies the key for the operator to replace. The name must match a function name listed in “Code You Can Replace from MATLAB Code” or “Code You Can Replace From Simulink Models”.

Example: `'Key','RTW_OP_ADD'`

`Priority` — Specifies the search priority of the function entry

`100` (default) | integer value 0..100

The *`Priority`* value specifies the search priority of the function entry, relative to other entries of the same function name and conceptual argument list within this table. Highest priority is 0, and lowest priority is 100. The default is 100. If the table provides two implementations for a function, the implementation with the higher priority shadows the one with the lower priority.

Example: `'Priority',100`

`RoundingModes` — Specifies rounding modes supported by implementation function

`'RTW_ROUND_UNSPECIFIED'` (default) | `'RTW_ROUND_FLOOR'` | `'RTW_ROUND_CEILING'` | `'RTW_ROUND_ZERO'` | `'RTW_ROUND_NEAREST'` | `'RTW_ROUND_NEAREST_ML'` | `'RTW_ROUND_CONV'` | `'RTW_ROUND_SIMPLEST'` | array of character vectors | string array

The *`RoundingModes`* value specifies one or more rounding modes supported by the implementation function.

Example: `'RoundingModes',{'RTW_ROUND_UNSPECIFIED'}`

`SaturationMode` — specifies saturation mode supported by implementation function

`'RTW_SATURATE_UNSPECIFIED'` (default) | `'RTW_SATURATE_ON_OVERFLOW'` | `'RTW_WRAP_ON_OVERFLOW'` | character vector | string scalar

The *`SaturationMode`* value specifies the saturation mode supported by the implementation function.

Example: `'SaturationMode','RTW_SATURATE_UNSPECIFIED'`

`SideEffects` — Specifies whether to attempt to optimize away the implementation function

`false` (default) | `true`

The *`SideEffects`* value flags the code generator not to optimize away the implementation function described by this entry. This parameter applies to implementation functions that return `void` but are not to be optimized away, such as a `memcpy` implementation or an implementation function that accesses global memory values. For those implementation functions only, you must include this parameter and specify the value `true`.

Example: `'SideEffects',false`

`StoreFcnReturnInLocalVar` — Specifies whether to store the implementation function regardless expression folding settings

`false` (default) | `true`

The *`StoreFcnReturnInLocalVar`* value flags the code generator that the return value of the implementation function described by this entry must be stored in a local variable regardless of other expression folding settings. If the value is `false`, other expression folding settings determine whether the return value is folded. Storing function returns in a local variable can increase the clarity of generated code. This example shows code generated with expression folding:

```
void sw_step(void)
{
    if (ssub(sadd(sw_U.In1, sw_U.In2), sw_U.In3) <=
        smul(ssub(sw_U.In4, sw_U.In5),sw_U.In6)) {
      sw_Y.Out1 = sw_U.In7;
```

```
    } else {
       sw_Y.Out1 = sw_U.In8;
    }
}
```

With `StoreFcnReturnInLocalVar` set to `true`, the generated code is potentially easier to understand and debug:

```
void sw_step(void)
{
    real32_T rtb_Switch;
    real32_T hoistedExpr;
    ......
    rtb_Switch = sadd(sw_U.In1, sw_U.In2);
    rtb_Switch = ssub(rtb_Switch, sw_U.In3);
    hoistedExpr = ssub(sw_U.In4, sw_U.In5);
    hoistedExpr = smul(hoistedExpr, sw_U.In6);
    if (rtb_Switch <= hoistedExpr) {
       sw_Y.Out1 = sw_U.In7;
    } else {
       sw_Y.Out1 = sw_U.In8;
    }
}
```

Example: `'StoreFcnReturnInLocalVar',false`

See Also

`addAdditionalHeaderFile` | `addAdditionalIncludePath` | `addAdditionalLinkObj` | `addAdditionalLinkObjPath` | `addAdditionalSourceFile` | `addAdditionalSourcepath`

Topics

“Define Code Replacement Library Optimizations”
“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”
“Mutex and Semaphore Functions”

Introduced in R2013a

coder.MATLABCodeTemplate.setTokenValue

Class: `coder.MATLABCodeTemplate`
Package: `coder`

Set value of token for code generation template

Syntax

`setTokenValue(tokenName,tokenValue)`

Description

`setTokenValue(tokenName,tokenValue)` sets the value of a token for a code generation template.

Input Arguments

`tokenName`

The name of the token

Default:

`tokenValue`

The value of the token

Default: empty

Examples

Create a MATLABCodeTemplate object from a custom template. Set the value for a custom token in the template.

```
newObj = coder.MATLABCodeTemplate('myCGTFile');
% Create a MATLABCodeTemplate object from a custom template file
newObj.setTokenValue('myCustomToken', 'myValue');
% Set the value of a custom token in the file
newObj.getTokenValue('myCustomToken')
% Check value of the custom token
```

See Also

`coder.MATLABCodeTemplate.emitSection` | `coder.MATLABCodeTemplate.getCurrentTokens` | `coder.MATLABCodeTemplate.getTokenValue`

Topics

“Generate Custom File and Function Banners for C/C++ Code”
“Code Generation Template Files for MATLAB Code”

setAlgorithmParameters

Set algorithm parameters for lookup table function code replacement table entry

Syntax

`setAlgorithmParameters(tableEntry, algParams)`

Description

`setAlgorithmParameters(tableEntry, algParams)` sets the algorithm parameters for the lookup table function identified in the code replacement table entry `tableEntry`.

Examples

Set Algorithm Parameters for `prelookup` Function Table Entry

Create a code replacement table.

```
crTable = RTW.TflTable;
```

Create a table entry for a function.

```
tableEntry = RTW.TflCFunctionEntry;
```

Identify the table entry as an entry for the `prelookup` function.

```
setTflCFunctionEntryParameters(tableEntry, ...
    'Key', 'prelookup', ...
    'Priority', 100, ...
    'ImplementationName', 'Ifx_DpSearch_u8');
```

Get the algorithm parameter settings for the `prelookup` function table entry.

```
algParams = getAlgorithmParameters(tableEntry)
```

```
algParams =

  Prelookup with properties:

             ExtrapMethod: [1x1 coder.algorithm.parameter.ExtrapMethod]
                  RndMeth: [1x1 coder.algorithm.parameter.RndMeth]
        IndexSearchMethod: [1x1 coder.algorithm.parameter.IndexSearchMethod]
        UseLastBreakpoint: [1x1 coder.algorithm.parameter.UseLastBreakpoint]
    RemoveProtectionInput: [1x1 coder.algorithm.parameter.RemoveProtectionInput]
```

Display the valid values for parameter `UseLastBreakpoint` for the `prelookup` function.

```
algParams.UseLastBreakpoint
```

```
ans =

  UseLastBreakpoint with properties:

       Name: 'UseLastBreakpoint'
    Options: {'off'  'on'}
    Primary: 0
      Value: {'off'  'on'}
```

Display the valid values for parameter `RemoveProtectionInput` for the `prelookup` function.

```
algParams.RemoveProtectionInput
```

```
ans = 

  RemoveProtectionInput with properties:

       Name: 'RemoveProtectionInput'
    Options: {'off'  'on'}
    Primary: 0
      Value: {'off'  'on'}
```

Set parameters `UseLastBreakpoint` and `RemoveProtectionInput` to `on` and `off`, respectively.

```
algParams.UseLastBreakpoint = 'on';
algParams.RemoveProtectionInput = 'off';
```

When you set each parameter, the algorithm parameter software checks for and reports errors for invalid syntax, parameter names, and values.

Update the parameter settings for the code replacement table entry.

```
setAlgorithmParameters(tableEntry, algParams);
```

Get the new algorithm parameter settings for the `prelookup` function table entry.

```
algParams = getAlgorithmParameters(tableEntry);
```

Examine the new value for `UseLastBreakpoint`.

```
algParams.UseLastBreakpoint
```

```
ans = 

  UseLastBreakpoint with properties:

       Name: 'UseLastBreakpoint'
    Options: {'off'  'on'}
    Primary: 0
      Value: {'on'}
```

Examine the new value for `RemoveProtectionInput`.

```
algParams.RemoveProtectionInput
```

```
ans = 

  RemoveProtectionInput with properties:

       Name: 'RemoveProtectionInput'
    Options: {'off'  'on'}
    Primary: 0
      Value: {'off'}
```

Set Algorithm Parameters for `lookup2D` Function Table Entry

Create a code replacement table.

```
crTable = RTW.TflTable;
```

Create a table entry for a function.

```
tableEntry = RTW.TflCFunctionEntry;
```

Identify the table entry as an entry for the `lookup2D` function.

```
setTflCFunctionEntryParameters(tableEntry, ...
    'Key', 'lookup2D', ...
    'Priority', 100, ...
    'ImplementationName', 'myLookup2D');
```

Get the algorithm parameter settings for the `lookup2D` function table entry.

```
algParams = getAlgorithmParameters(tableEntry)
```

```
algParams = 

  Lookup with properties:

                 InterpMethod: [1x1 coder.algorithm.parameter.InterpMethod]
                 ExtrapMethod: [1x1 coder.algorithm.parameter.ExtrapMethod]
         UseRowMajorAlgorithm: [1x1 coder.algorithm.parameter.UseRowMajorAlgorithm]
                      RndMeth: [1x1 coder.algorithm.parameter.RndMeth]
            IndexSearchMethod: [1x1 coder.algorithm.parameter.IndexSearchMethod]
            UseLastTableValue: [1x1 coder.algorithm.parameter.UseLastTableValue]
        RemoveProtectionInput: [1x1 coder.algorithm.parameter.RemoveProtectionInput]
    SaturateOnIntegerOverflow: [1x1 coder.algorithm.parameter.SaturateOnIntegerOverflow]
      SupportTunableTableSize: [1x1 coder.algorithm.parameter.SupportTunableTableSize]
              BPPower2Spacing: [1x1 coder.algorithm.parameter.BPPower2Spacing]
```

Display the valid values for algorithm parameter `IndexSearchMethod` for the `lookup2D` function.

```
algParams.IndexSearchMethod
```

```
ans = 

  IndexSearchMethod with properties:

       Name: 'IndexSearchMethod'
    Options: {'Linear search'  'Binary search'  'Evenly spaced points'}
    Primary: 0
      Value: {'Binary search'  'Evenly spaced points'  'Linear search'}
```

Set parameter `IndexSearchMethod` to `Evenly spaced points`.

```
algParams.IndexSearchMethod = 'Evenly spaced point';
```

```
Error using coder.algorithm.parameter.validateValue (line 58)
Invalid value '{Evenly spaced point}' for algorithm parameter
'coder.algorithm.parameter.IndexSearchMethod'. Valid values are '{Linear
search, Binary search, Evenly spaced points}'.

Error in coder.algorithm.parameter.AlgorithmParameter/set.Value (line 49)
            obj.Value = coder.algorithm.parameter.validateValue(obj, val);

Error in coder.algorithm.parameter.AlgorithmParameter/setAP (line 36)
                obj.Value = value;

Error in coder.algorithm.parameterset.Lookup/set.IndexSearchMethod (line 39)
            obj.IndexSearchMethod = obj.IndexSearchMethod.setAP(value);
```

The code replacement software flags the ‘`s`’ that is missing from ‘`points`’.

Adjust the parameter setting.

```
algParams.IndexSearchMethod = 'Evenly spaced points';
```

Update the parameter settings for the code replacement table entry.

```
setAlgorithmParameters(tableEntry, algParams);
```

Get the updated algorithm parameter settings for the `lookup2D` function table entry.

```
algParams = getAlgorithmParameters(tableEntry);
```

Verify the new value of `IndexSearchMethod`.

```
algParams.IndexSearchMethod
```

```
ans = 

  IndexSearchMethod with properties:

       Name: 'IndexSearchMethod'
    Options: {'Linear search'  'Binary search'  'Evenly spaced points'}
```

```
    Primary: 0
      Value: {'Evenly spaced points'}
```

Input Arguments

`tableEntry` — Code replacement table entry for a lookup table function

object

Code replacement table entry that you previously created and represents a potential code replacement for a lookup table function. The entry must identify the lookup table function for which you are calling `setAlgorithmParameters`.

1. Create the entry. For example, call the function `RTW.TflCFunctionEntry`.

   ```
   tableEntry = RTW.TflCFunctionEntry;
   ```

2. Specify the name of the lookup table function for which you created the entry. Use the `Key` parameter in a call to `setTflCFunctionEntryParameters`. The following function call specifies the lookup table function `prelookup`.

   ```
   setTflCFunctionEntryParameters(tableEntry, ...
      'Key', 'prelookup', ...
      'Priority', 100, ...
      'ImplementationName', 'Ifx_DpSearch_u8');
   ```

`algParams` — Algorithm parameter settings for a lookup table function

object

Algorithm parameter settings for the lookup table function identified with the `Key` parameter in `tableEntry`.

See Also

`RTW.TflCFunctionEntry` | `RTW.TflTable` | `addEntry` | `getAlgorithmParameters` | `setTflCFunctionEntryParameters`

Topics

“Lookup Table Function Code Replacement”
“Define Code Replacement Library Optimizations”
“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”

Introduced in R2015a

coder.mapping.create

Create C code mapping environment for model

Syntax

```
coder.mapping.create(model)
coder.mapping.create(model,cs)
```

Description

`coder.mapping.create(model)` creates an environment to configure code generation for data and functions of the specified model. Before calling other default mapping functions, call this function.

`coder.mapping.create(model,cs)` creates a code mapping environment for the specified model that includes code customization settings stored in a configuration set object. The configuration set object can specify memory sections for data and functions and a naming rule for shared utilities. Specify a configuration set object to preserve memory section definitions or shared utility naming rules applied to a model in a version of Embedded Coder prior to R2018a.

Examples

Create Environment to Configure Code Mappings for Model

For model `rtwdemo_configdefaults`, create the environment for configuring data and functions for code generation.

```
coder.mapping.create('rtwdemo_configdefaults');
```

After calling this function, use calls to these functions to look up category names, property names, and values that you can use to configure aspects of code generation for model data and functions:

- `coder.mapping.defaults.dataCategories`
- `coder.mapping.defaults.functionCategories`
- `coder.mapping.defaults.allowedProperties`
- `coder.mapping.defaults.allowedValues`

Then, specify category, property, and value combinations in calls to `coder.mapping.defaults.set`.

Input Arguments

`model` — Name of model

character vector

Model file, specified as a character vector. The model must be loaded (for example, by using `load_system`) or open. You can omit the `.slx` file extension.

Example: `'myLoadedModel'`

Data Types: `char`

cs — Configuration set object
object

Configuration set object from which to import code customization settings for data and functions.

Example: 'cs_basic'

Data Types: char

See Also
coder.mapping.defaults.allowedProperties | coder.mapping.defaults.allowedValues | coder.mapping.defaults.dataCategories | coder.mapping.defaults.functionCategories | coder.mapping.defaults.get | coder.mapping.defaults.set

Topics
"Configure Default C Code Generation for Categories of Data Elements and Functions"

Introduced in R2018a

coder.mapping.defaults.allowedProperties

Return properties for model default mapping category

Syntax

```
properties = coder.mapping.defaults.allowedProperties(model,category)
```

Description

`properties = coder.mapping.defaults.allowedProperties(model,category)` returns a cell array of names for properties that are relevant to `category` for the specified model. Use the property names that the `coder.mapping.defaults.allowedProperties` function returns in subsequent calls to `coder.mapping.defaults.allowedValues` and `coder.mapping.defaults.set`.

Examples

Get Properties for Model Default Data Categories

Get a list of the properties for the model default data categories `Inports`, `Outports`, `ModelParameters`, and `InternalData` by using calls to `coder.mapping.defaults.allowedProperties`.

Load the model.

```
open_system('rtwdemo_configdefaults');
```

Get a list of the properties for the model default data category `Inports`.

```
coder.mapping.defaults.allowedProperties('rtwdemo_configdefaults',...
 'Inports')
```

```
ans =

  3×1 cell array

    {'StorageClass'      }
    {'HeaderFile'        }
    {'PreserveDimensions'}
```

Get a list of the properties for the model default data category `Outports`.

```
coder.mapping.defaults.allowedProperties('rtwdemo_configdefaults',...
 'Outports')
```

```
ans =

  5×1 cell array

    {'StorageClass'      }
    {'HeaderFile'        }
```

```
    {'DefinitionFile'    }
    {'Owner'             }
    {'PreserveDimensions'}
```

Get a list of the properties for the model default data category `ModelParameters`.

```
coder.mapping.defaults.allowedProperties('rtwdemo_configdefaults',...
 'ModelParameters')
```

```
ans =

  1×1 cell array

    {'StorageClass'}
```

Get a list of the properties for the model default data category `InternalData`.

```
coder.mapping.defaults.allowedProperties('rtwdemo_configdefaults',...
 'InternalData')
```

```
ans =

  2×1 cell array

    {'StorageClass' }
    {'MemorySection'}
```

Get Properties for Model Default Function Categories

Get a list of the properties for the model default function categories `InitializeTerminate` and `Execution` by using calls to `coder.mapping.defaults.allowedProperties`.

Load the model.

```
open_system('rtwdemo_configdefaults');
```

Get a list of the properties for the model default function category `InitializeTerminate`.

```
catData = coder.mapping.defaults.allowedProperties('rtwdemo_configdefaults',...
  'InitializeTerminate');
catData
```

```
catData =

  1×1 cell array

    {'FunctionCustomizationTemplate'}
```

Get a list of the properties for the model default function category `Execution`.

```
catFunctions = coder.mapping.defaults.allowedProperties('rtwdemo_configdefaults',...
'Execution');
catFunctions
```

```
catFunctions =

  1×1 cell array
```

```
{'FunctionCustomizationTemplate'}
```

Input Arguments

`model` — Name of model

character vector

Model file, specified as a character vector. The model must be loaded (for example, by using `load_system`) or open. You can omit the `.slx` file extension.

Example: `'myLoadedModel'`

Data Types: `char`

`category` — Name of default mapping category

character vector

Category name, specified as a character vector. To get valid data and function category names, call the functions `coder.mappings.defaults.dataCategories` and `coder.mappings.defaults.functionCategories`.

Example: `'Inports'`

Data Types: `char`

Output Argument

`properties` — Names of properties for category

cell array

Cell array of names for properties of a default category for the specified model.

See Also

`coder.mapping.defaults.allowedValues` | `coder.mapping.defaults.dataCategories` | `coder.mapping.defaults.functionCategories` | `coder.mapping.defaults.get` | `coder.mapping.defaults.set`

Topics

"Configure Default C Code Generation for Categories of Data Elements and Functions"

Introduced in R2018a

coder.mapping.defaults.allowedValues

Return value of property for model default mapping category

Syntax

`values = coder.mapping.defaults.allowedValues(model,category,property)`

Description

`values = coder.mapping.defaults.allowedValues(model,category,property)` returns a cell array of values that are relevant to the specified combination of `category` and `property` for the specified model. To set up category, property, and value combinations for a model, use the value names that the function returns in calls to `coder.mapping.defaults.set`.

Examples

Get Storage Class Values for Default Data Category Model Parameters

Get the list of values that you can specify for property `StorageClass` for model default data category `ModelParameters` by calling `coder.mapping.defaults.allowedValues`.

```
lclparam_scs = coder.mapping.defaults.allowedValues('rtwdemo_configdefaults', 'ModelParameters',...
  'StorageClass')
lclparam_scs

lclparam_scs =

  14×1 cell array

    {'Default'              }
    {'ExportedGlobal'       }
    {'ImportedExtern'       }
    {'ImportedExternPointer'}
    {'Const'                }
    {'Volatile'             }
    {'ConstVolatile'        }
    {'Define'               }
    {'ImportedDefine'       }
    {'ExportToFile'         }
    {'ImportFromFile'       }
    {'FileScope'            }
    {'GetSet'               }
    {'CompilerFlag'         }
```

Input Arguments

`model` — Name of model
character vector

Model file, specified as a character vector. The model must be loaded (for example, by using `load_system`) or open. You can omit the `.slx` file extension.

Example: `'myLoadedModel'`

Data Types: `char`

`category` — Name of default mapping category
character vector

Category name, specified as a character vector. To get valid data and function category names, call the functions `coder.mappings.defaults.dataCategories` and `coder.mappings.defaults.functionCategories`.

Example: `'ModelParameters'`

Data Types: `char`

`property` — Name of property for default mapping category
character vector

Property name, specified as a character vector. To get valid property names for a default mapping category, call the function `coder.mappings.defaults.allowedProperties`.

Example: `'StorageClass'`

Data Types: `char`

Output Argument

`values` — Values for category and property combination
cell array

Cell array of values that are for a default category and property combination for the specified model.

See Also

`coder.mapping.defaults.allowedProperties` | `coder.mapping.defaults.dataCategories` | `coder.mapping.defaults.functionCategories` | `coder.mapping.defaults.get` | `coder.mapping.defaults.set`

Topics

"Configure Default C Code Generation for Categories of Data Elements and Functions"

Introduced in R2018a

coder.mapping.defaults.dataCategories

Return default mapping categories for model data

Syntax

`categories = coder.mapping.defaults.dataCategories()`

Description

`categories = coder.mapping.defaults.dataCategories()` returns a cell array of names for categories of model data elements that you can map to property settings, including a storage class and memory section. The storage class mapped to a category defines how the code generator produces code for that category of data. To set up data category, property, and value combinations for a model, use the category names that the function returns in calls to:

- `coder.mapping.defaults.allowedProperties`
- `coder.mapping.defaults.allowedValues`
- `coder.mapping.defaults.set`

Example

Get Model Data Element Categories

Get a list of the available data categories by calling `coder.mapping.defaults.dataCategories`.

```
catData = coder.mapping.defaults.dataCategories()
catData

ans =

  1×9 cell array

  Columns 1 through 4

    {'Inports'}    {'Outports'}    {'ModelParameters'}    {'ModelParameterA...'}

  Columns 5 through 8

    {'ExternalParamet...'}    {'SharedLocalData...'}    {'GlobalDataStores'}    {'InternalData'}

  Column 9

    {'Constants'}
```

Output Arguments

`categories` — Names of data categories

cell array

Cell array of names for default mapping data categories.

See Also

`coder.mapping.defaults.allowedProperties` | `coder.mapping.defaults.allowedValues` | `coder.mapping.defaults.dataCategories` |

`coder.mapping.defaults.functionCategories` | `coder.mapping.defaults.get` | `coder.mapping.defaults.set`

Topics
"Configure Default C Code Generation for Categories of Data Elements and Functions"

Introduced in R2018a

coder.mapping.defaults.functionCategories

Return default mapping categories for model functions

Syntax

`categories = coder.mapping.defaults.functionCategories()`

Description

`categories = coder.mapping.defaults.functionCategories()` returns a cell array of names for categories of model functions that you can map to property settings, including a function customization template and memory section. The function customization template mapped to a category defines how the code generator produces code for that category of functions. To set up function category, property, and value combinations for a model, use the category names that the function returns in calls to:

- `coder.mapping.defaults.allowedProperties`
- `coder.mapping.defaults.allowedValues`
- `coder.mapping.defaults.set`

Examples

Get Model Function Categories

Get a list of the available function categories by calling `coder.mapping.defaults.functionCategories`.

```
catFunc = coder.mapping.defaults.functionCategories();
catFunc

catFunc =

  1×3 cell array

    {'InitializeTerminate'}    {'Execution'}    {'SharedUtility'}
```

Output Arguments

`categories` — Names of function categories
cell array

Cell array of names for default mapping function categories.

See Also
`coder.mapping.defaults.allowedProperties` | `coder.mapping.defaults.allowedValues` | `coder.mapping.defaults.dataCategories` | `coder.mapping.defaults.get` | `coder.mapping.defaults.set`

Topics
"Configure Default C Code Generation for Categories of Data Elements and Functions"

Introduced in R2018a

coder.mapping.defaults.get

Return value of property for model default mapping category

Syntax

`value = coder.mapping.defaults.get(model,category,property)`

Description

`value = coder.mapping.defaults.get(model,category,property)` returns the value of a property for a data or function default mapping category for a model. To determine valid category and property combinations, use calls to functions `coder.mapping.defaults.dataCategories`, `coder.mapping.defaults.functionCategories`, and `coder.mapping.defaults.allowedProperties`.

Examples

Return Storage Class Setting for Data Imported Into Model

For model `rtwdemo_configuredefaults`, return the storage class that the code generator uses for data imported into the model from external header and definitions files.

Determine the category name to specify for model input data by calling `coder.mapping.defaults.dataCategories`.

```
coder.mapping.defaults.dataCategories()

ans =

  1×9 cell array

  Columns 1 through 4

    {'Inports'}    {'Outports'}    {'ModelParameters'}    {'ModelParameterA...'}

  Columns 5 through 8

    {'ExternalParamet...'}    {'SharedLocalData...'}    {'GlobalDataStores'}    {'InternalData'}

  Column 9

    {'Constants'}
```

Specify `Inports` as the category name.

Identify properties that you can configure for category **Outports** by calling `coder.mapping.defaults.allowedProperties`.

```
coder.mapping.defaults.allowedProperties('rtwdemo_configdefaults', 'Outports')

ans =

  4×1 cell array

    {'StorageClass'  }
    {'HeaderFile'    }
    {'DefinitionFile'}
    {'Owner'         }
```

Use a call to function `coder.mapping.defaults.get` to return the setting for category **Inports** and property `StorageClass`.

```
coder.mapping.defaults.get('rtwdemo_configdefaults', 'Inports', 'StorageClass')

ans =

    'ImportFromFile'
```

Return Memory Section Setting for Model Execution Entry-Point Functions

For model `rtwdemo_configuredefaults`, return the memory section that the code generator uses for model execution entry-point functions, such as `step`.

Determine the category name to specify for execution functions by calling `coder.mapping.defaults.functionCategories`.

```
coder.mapping.defaults.functionCategories()

ans =

  1×3 cell array

    {'InitializeTerminate'}    {'Execution'}    {'SharedUtility'}
```

Specify `Execution` as the category name.

Identify properties that you can configure for category `Execution` by calling `coder.mapping.defaults.allowedProperties`.

```
coder.mapping.defaults.allowedProperties('rtwdemo_configdefaults', 'Execution')

ans =

  1×1 cell array

    {'FunctionCustomizationTemplate'}
```

Use a call to function `coder.mapping.defaults.get` to return the setting for category `Execution` and property `FunctionCustomizationTemplate`.

```
coder.mapping.defaults.get('rtwdemo_configdefaults', 'Execution', 'FunctionCustomizationTemplate')

ans =

    'exFastFunction'
```

Input Arguments

`model` — Name of model
character vector

Model file, specified as a character vector. The model must be loaded (for example, by using `load_system`) or open. You can omit the `.slx` file extension.

Example: `'myLoadedModel'`

Data Types: `char`

`category` — Name of default mapping category
character vector

Category name, specified as a character vector. To get valid data and function category names, call the functions `coder.mappings.defaults.dataCategories` and `coder.mappings.defaults.functionCategories`.

Example: `'ModelParameters'`

Data Types: `char`

`property` — Name of property for default mapping category
character vector

Property name, specified as a character vector. To get valid property names for a default mapping category, call the function `coder.mappings.defaults.allowedProperties`.

Example: `'StorageClass'`

Data Types: `char`

Output Argument

`value` — Value of property for default mapping category
character vector

Character vector that is the setting of the specified default mapping category and property for the specified model.

See Also

`coder.mapping.defaults.allowedProperties` | `coder.mapping.defaults.allowedValues` | `coder.mapping.defaults.dataCategories` | `coder.mapping.defaults.functionCategories` | `coder.mapping.defaults.set`

Topics
"Configure Default C Code Generation for Categories of Data Elements and Functions"

Introduced in R2018a

coder.mapping.defaults.set

Set value for property of model default mapping category

Syntax

`coder.mapping.defaults.set(model,category,property,value,...)`

Description

`coder.mapping.defaults.set(model,category,property,value,...)` sets property values for a data or function default mapping category for a model. To determine valid category, property, and value combinations for a model, use calls to:

- `coder.mapping.defaults.dataCategories`
- `coder.mapping.defaults.functionCategories`
- `coder.mapping.defaults.allowedProperties`
- `coder.mapping.defaults.allowedValues`

Examples

Configure Default Code Generation Settings for Model Output Data

For model `rtwdemo_configuredefaults`, configure how the code generator handles model output data by default.

Determine the category name to specify for model output data by calling `coder.mapping.defaults.dataCategories`.

```
coder.mapping.defaults.dataCategories()

ans =

  1×9 cell array

  Columns 1 through 4

    {'Inports'}    {'Outports'}    {'ModelParameters'}    {'ModelParameterA...'}

  Columns 5 through 8

    {'ExternalParamet...'}    {'SharedLocalData...'}    {'GlobalDataStores'}    {'InternalData'}

  Column 9

    {'Constants'}
```

Specify `Outports` as the category name.

Identify properties that you can configure for category `Outports` by calling `coder.mapping.defaults.allowedProperties`.

```
coder.mapping.defaults.allowedProperties('rtwdemo_configdefaults', 'Outports')

ans =
```

```
  4×1 cell array

    {'StorageClass'  }
    {'HeaderFile'    }
    {'DefinitionFile'}
    {'Owner'         }
```

For this example, set values for properties `StorageClass`, `HeaderFile`, and `DefinitionFile`.

Look up the values that you can specify for properties `StorageClass`, `HeaderFile`, and `DefinitionFile`.

```
coder.mapping.defaults.allowedValues('rtwdemo_configdefaults', 'Outports', 'StorageClass')

ans =

  9×1 cell array

    {'Default'              }
    {'ExportedGlobal'       }
    {'ImportedExtern'       }
    {'ImportedExternPointer'}
    {'Volatile'             }
    {'ExportToFile'         }
    {'ImportFromFile'       }
    {'AutoScope'            }
    {'GetSet'               }

coder.mapping.defaults.allowedValues('rtwdemo_configdefaults', 'Outports', 'HeaderFile')

ans =

  0×1 empty cell array

coder.mapping.defaults.allowedValues('rtwdemo_configdefaults', 'Outports', 'DefinitionFile')

ans =

  0×1 empty cell array
```

Use a call to function `coder.mapping.defaults.set` to configure the default settings. For category `Outports`, set `StorageClass` to `ExportToFile`. Specify `exSysOut.h` and `exSysOut.c` for the header and definition files.

```
coder.mapping.defaults.set('rtwdemo_configdefaults', 'Outports',...
  'Storageclass', 'ExportToFile',...
  'HeaderFile', 'exSysOut.h',...
  'DefinitionFile', 'exSysOut.c')
```

Configure Default Location in Memory for Storing Code Generated for Model Internal Data

For model `rtwdemo_configuredefaults`, configure the default location in memory for storing code generated for model data elements such as signals, states, and zero crossings.

Determine the category name to specify for model internal data by calling `coder.mapping.defaults.dataCategories`.

```
coder.mapping.defaults.dataCategories()

ans =

  1×9 cell array

  Columns 1 through 4

    {'Inports'}    {'Outports'}    {'ModelParameters'}    {'ModelParameterA...'}

  Columns 5 through 8
```

```
    {'ExternalParamet...'}    {'SharedLocalData...'}    {'GlobalDataStores'}    {'InternalData'}

  Column 9

    {'Constants'}
```

Specify `InternalData` as the category name.

Identify properties that you can configure for category `InternalData` by calling `coder.mapping.defaults.allowedProperties`.

```
coder.mapping.defaults.allowedProperties('rtwdemo_configdefaults', 'InternalData')

ans =

  2×1 cell array

    {'StorageClass' }
    {'MemorySection'}
```

To configure the memory location, set the value for property `MemorySection`.

Look up the values that you can specify for property `MemorySection`.

```
coder.mapping.defaults.allowedValues('rtwdemo_configdefaults', 'InternalData', 'MemorySection')

ans =

  5×1 cell array

    {'None'           }
    {'MemVolatile'    }
    {'functionFastMem'}
    {'functionSlowMem'}
    {'internalDataMem'}
```

Use a call to function `coder.mapping.defaults.set` to configure the default setting. For category `InternalData`, set `MemorySection` to `internalDataMem`.

```
coder.mapping.defaults.set('rtwdemo_configdefaults', 'InternalData',...
  'MemorySection', 'internalDataMem')
```

Input Arguments

model — Name of model

character vector

Model file, specified as a character vector. The model must be loaded (for example, by using `load_system`) or open. You can omit the `.slx` file extension.

Example: `'myLoadedModel'`

Data Types: `char`

category — Name of default mapping category

character vector

Category name, specified as a character vector. To get valid data and function category names, call the functions `coder.mappings.defaults.dataCategories` and `coder.mappings.defaults.functionCategories`.

Example: `'ModelParameters'`

Data Types: `char`

`property` — Name of property for default mapping category
character vector

Property name, specified as a character vector. To get valid property names for a default mapping category, call the function `coder.mappings.defaults.allowedProperties`.

Example: `'StorageClass'`

Data Types: `char`

`value` — Value of property for default mapping category
character vector

Property value, specified as a character vector. To get a list of values that you can specify for a category and property combination, call the function `coder.mappings.defaults.allowedValues`.

Example: `'ExportToFile'`

See Also

`coder.mapping.defaults.allowedProperties` | `coder.mapping.defaults.allowedValues` | `coder.mapping.defaults.dataCategories` | `coder.mapping.defaults.functionCategories` | `coder.mapping.defaults.get`

Topics

"Configure Default C Code Generation for Categories of Data Elements and Functions"

Introduced in R2018a

coder.mapping.api.CodeMappingCPP

Model data and interface configuration for C++ code generation

Description

A code mappings object and related functions enable you to configure C++ code generation for data and functions of a Simulink model. For model data elements, code mappings associate data elements with configurations that consist of a storage class and storage class properties. For functions, code mappings associate entry-point functions with configurations that consist of a function customization template. Reduce the effort of preparing a model for C++ code generation by specifying default configurations for categories of data elements and functions across a model. Override default configurations by configuring data elements or functions individually. For smaller models, you can choose to configure each data element and function individually.

Creation

When you select the **Embedded Coder** app from the Apps tab in the Simulink Editor, the app creates a `coder.mapping.api.CodeMappingCPP` object if code mappings do not already exist. The app creates code mappings based on code customization settings stored in the model active configuration set object. The configuration set object can specify memory sections for data and functions.

Access a `coder.mapping.api.CodeMappingCPP` object programmatically by using the `coder.mapping.utils.create` or `coder.mapping.api.get` functions.

Object Functions

find	Get model elements for the category of model code mappings
getClassName	Get class name of model
setClassName	Set class name of model
getClassNamespace	Get class namespace for a model
setClassNamespace	Set class namespace of model
getData	Get code mapping configuration for model data
setData	Configure model data for C++ code generation
getFunction	Get code configuration from code mappings for model function
setFunction	Set code mapping information for model function

Examples

Create Environment to Configure C++ Code Mappings for Model

For model `rtwdemo_cppclass`, create the environment for configuring model data and functions for code generation. After calling this function, use calls to other functions listed under Object Functions to configure aspects of code generation for model interface elements.

```
coder.mapping.utils.create('rtwdemo_cppclass');
```

See Also

`coder.mapping.api.get` | `coder.mapping.utils.create`

Topics

"Interactively Configure C++ Interface"
"Programmatically Configure C++ Interface"

Introduced in R2021a

getClassName

Get class name of model

Syntax

```
name = getClassName(myCPPMappingObj)
```

Description

`name = getClassName(myCPPMappingObj)` returns the class name of the model.

Examples

Get Class Name of Model

Open the model. To access the `CodeMappingCPP` object associated with the model, use the `coder.mapping.api.get` function.

```
open_system('rtwdemo_cppclass');
cm = coder.mapping.api.get('rtwdemo_cppclass');
```

To access the class name of the model, use the `getClassName` function. If you did not specify a class name for the model, the `getClassName` function returns an empty character vector, and the class name in the generated code uses a default class name of the form *`model`*`ModelClass`.

```
name = getClassName(cm)
```

```
name =

    'ModelClass'
```

Specify a class name for the model by using the `setClassName` function.

```
setClassName(cm, 'myClassName');
```

The `getClassName` function now returns the specified class name.

```
name = getClassName(cm)
```

```
name =

    'myClassName'
```

Input Arguments

`myCPPMappingObj` — C++ code mapping object
`CodeMappingCPP` object

C++ code mapping object, returned by a call to either the `coder.mapping.utils.create` function or the `coder.mapping.api.get` function.

Output Arguments

name — Class name of model
character vector

Class name of model, returned as a character vector. If you do not specify a class name, the class name in the generated code uses a default class name of the form `*model*ModelClass`.

See Also

`getClassNamespace` | `getData` | `getFunctionName` | `setClassName` | `setClassNamespace` | `setData` | `setFunctionName`

Topics

"Interactively Configure C++ Interface"
"Programmatically Configure C++ Interface"

Introduced in R2021a

find

Get model elements for the category of model code mappings

Syntax

```
modelElementsFound = find(myCPPMappingObj,category)
modelElementsFound = find(myCPPMappingObj,category,'MethodName', methodName)
```

Description

`modelElementsFound = find(myCPPMappingObj,category)` returns the elements in the model code mappings of the specified category as an array of objects.

`modelElementsFound = find(myCPPMappingObj,category,'MethodName', methodName)` returns the model functions in the model code mappings with the specified method name.

Examples

Find all model functions with an unset method name

You can use the `find` function to find all model elements of a specific category in the code mappings.

Open the model. To access the `CodeMappingCPP` object associated with the model, use the `coder.mapping.api.get` function.

```
open_system('rtwdemo_cppclass');
cm = coder.mapping.api.get('rtwdemo_cppclass');
```

Find all functions with an unset method name by using the `find` function.

```
unsetMethods = find(cm, 'Functions', 'MethodName', '')
```

```
unsetMethods = 

  1×2 string array

    "Initialize"    "Terminate"
```

To specify a method name for the functions, use the `setFunction` function.

```
setFunction(cm, unsetMethods, 'MethodName', 'my_$N')
```

Input Arguments

myCPPMappingObj — C++ code mapping object
`CodeMappingCPP` object

C++ code mapping object, returned by a call to either the `coder.mapping.utils.create` function or the `coder.mapping.api.get` function.

`category` — Model element category
`Functions` | `ExportedFunctions` | `PartitionFunctions` | `PeriodicFunctions` | `ResetFunctions` | `SimulinkFunctions`

Category of model elements to search for in the model code mappings.

Example: `'ResetFunctions'`

`methodName` — Name of entry-point function
character vector | string scalar

Name of an entry-point function generated for a model.

Example: `'my_$N'`

Data Types: `char` | `string`

Output Arguments

`modelElementsFound` — Model elements found
array | string vector

Model elements found, returned as an array or string vector of functions. Each object identifies a model element of the specified category. If you specify additional search criteria, the array or string vector includes objects for model elements of the specified category that meet the additional search criteria.

See Also

`getClassName` | `getClassNamespace` | `getData` | `getFunction` | `setClassName` | `setClassNamespace` | `setData` | `setFunction`

Topics

"Interactively Configure C++ Interface"
"Programmatically Configure C++ Interface"

Introduced in R2021a

setClassName

Set class name of model

Syntax

```
setClassName(myCPPMappingObj, name)
```

Description

`setClassName(myCPPMappingObj, name)` sets the class name of the model in the generated code.

Examples

Set Class Name of Model

Open the model. To access the `CodeMappingCPP` object associated with the model, use the `coder.mapping.api.get` function.

```
open_system('rtwdemo_cppclass');
cm = coder.mapping.api.get('rtwdemo_cppclass');
```

To access the class name of the model, use the `getClassName` function. If you did not specify a class name for the model, the `getClassName` function returns an empty character vector, and the class name in the generated code uses the default class name.

```
name = getClassName(cm)
```

```
name =

    'ModelClass'
```

Specify a class name for the model by using the `setClassName` function.

```
setClassName(cm, 'myClassName');
```

The `getClassName` function now returns the specified class name.

```
name = getClassName(cm)
```

```
name =

    'myClassName'
```

Input Arguments

`myCPPMappingObj` — C++ code mapping object
`CodeMappingCPP` object

C++ code mapping object, returned by a call to either the `coder.mapping.utils.create` function or the `coder.mapping.api.get` function.

name — Class name of model
character vector

Class name of model in the generated code, specified as a character vector. If you do not specify a class name, the class name of the model in the generated code is set to the name of the model.

Data Types: char | string

See Also

find | getClassName | getClassNamespace | getData | getFunction | setClassNamespace | setData | setFunction

Topics
"Interactively Configure C++ Interface"
"Programmatically Configure C++ Interface"

Introduced in R2021a

getClassNamespace

Get class namespace for a model

Syntax

```
namespace = getClassNamespace(myCPPMappingObj)
```

Description

`namespace = getClassNamespace(myCPPMappingObj)` returns the class namespace specified for the model. Class namespaces can help to prevent name conflicts in large projects.

Examples

Access Class Namespace for Model

Open the model. To access the `CodeMappingCPP` object associated with the model, use the `coder.mapping.api.get` function.

```
open_system('rtwdemo_cppclass');
cm = coder.mapping.api.get('rtwdemo_cppclass');
```

To access the namespace of the model, use the `getClassNamespace` function. If you did not specify a namespace for the model, the `getClassNamespace` function returns an empty character vector.

```
name = getClassNamespace(cm)
```

```
name =

  'TopNS'
```

Specify a namespace for the model by using the `setClassNamespace` function.

```
setClassNamespace(cm, 'myClassNamespace');
```

The `getClassNamespace` function now returns the specified class namespace.

```
name = getClassNamespace(cm)
```

```
name =

    'myClassNamespace'
```

Input Arguments

myCPPMappingObj — C++ code mapping object
`CodeMappingCPP` object

C++ code mapping object, returned by a call to either the `coder.mapping.utils.create` function or the `coder.mapping.api.get` function.

Output Arguments

namespace — Class namespace of model
character vector

Class namespace of model, returned as a character vector. If you did not specify a namespace for the model, the `getClassNamespace` function returns an empty character vector.

See Also

`find` | `getClassName` | `getData` | `getFunctionName` | `setClassName` | `setClassNamespace` | `setData` | `setFunctionName`

Topics
"Interactively Configure C++ Interface"
"Programmatically Configure C++ Interface"

Introduced in R2021a

setClassNamespace

Set class namespace of model

Syntax

```
setClassNamespace(myCPPMappingObj, namespace)
```

Description

`setClassNamespace(myCPPMappingObj, namespace)` sets the class namespace of the model in the generated code. Control the scope of the generated code by specifying a namespace for the generated class. In systems that use a model hierarchy, you can specify a different namespace for each model in the hierarchy.

Examples

Set Class Namespace for Model

Specify a class namespace for a model and generate C++ code.

Open the model. To access the `CodeMappingCPP` object associated with the model, use the `coder.mapping.api.get` function.

```
open_system('rtwdemo_cppclass');
cm = coder.mapping.api.get('rtwdemo_cppclass');
```

To specify a namespace for the model in the generated code, use the `setClassNamespace` function.

```
setClassNamespace(cm, 'myClassNamespace');
```

Input Arguments

`myCPPMappingObj` — C++ code mapping object
`CodeMappingCPP` object

C++ code mapping object, returned by a call to either the `coder.mapping.utils.create` function or the `coder.mapping.api.get` function.

`namespace` — Class namespace of model
character vector

Class namespace of model in the generated code, specified as a character vector. If you do not specify a class namespace, the code generated for the model does not use a namespace.

Data Types: `char` | `string`

See Also

`find` | `getClassName` | `getClassNamespace` | `getData` | `getFunction` | `setClassName` | `setData` | `setFunction`

Topics
“Interactively Configure C++ Interface”
“Programmatically Configure C++ Interface”

Introduced in R2021a

getData

Get code mapping configuration for model data

Syntax

```
value = getData(myCPPMappingObj, category, property)
```

Description

`value = getData(myCPPMappingObj, category, property)` returns the code mapping information for the property specified by `property` and the model data category specified by `category`.

Examples

Configure data visibility of model parameters

Configure the data visibility of model parameters to be public.

Open the model. Use the `coder.mapping.api.get` function to access the `CodeMappingCPP` object associated with the model.

```
open_system('rtwdemo_cppclass');
cm = coder.mapping.api.get('rtwdemo_cppclass');
```

To view the data visibility of the model parameters, use the `getData` function.

```
value = getData(cm, 'ModelParameters', 'DataVisibility')
```

```
value =

    'private'
```

To configure the data visibility, specify the `'DataVisibility'` parameter using the `setData` function.

```
setData(cm, 'ModelParameters', 'DataVisibility', 'public');
```

Input Arguments

`myCPPMappingObj` — C++ code mapping object
`CodeMappingCPP` object

C++ code mapping object, returned by a call to either the `coder.mapping.utils.create` function or the `coder.mapping.api.get` function.

`category` — Category of model data
`'Inports'` | `'Outports'` | `'ModelParameters'` | `'ModelParameterArguments'` | `'InternalData'`

Category of model data to access, specified as one of these categories.

Category	Description
'Inports'	Root-level input ports of a model, such as Inport and In Bus Element blocks.
'Outports'	Root-level output ports of a model, such as Outport and Out Bus Element blocks.
'ModelParameters'	Parameters that are defined within a model, such as parameters in the model workspace. Excludes model arguments.
'ModelParameterArguments'	Parameters in the model workspace configured as model arguments. These parameters are exposed at the model block to enable each model instance to provide its own value.
'InternalData'	Data elements that are internal to a model, such as block output signals, discrete block states, data stores, and zero-crossing signals.

Data Types: char | string

property — Property of model data
'MemberAccessMethod' | 'DataVisibility' | 'DataAccess'

Property of model data to access, specified as either 'MemberAccessMethod', 'DataVisibility', or 'DataAccess'.

The MemberAccessMethod property specifies how the methods, if any, are generated for the data elements.

The DataVisibility property specifies the visibility (private, public, or protected) of the data category in the generated code.

The DataAccess property specifies if model parameter arguments are stored by value or pointer in the generated code.

Data Types: char | string

Output Arguments

value — Code mapping property value of category
character vector

The code mapping property value of the specified category, returned as a character vector.

See Also

find | getClassName | getClassNamespace | getFunction | setClassName | setClassNamespace | setData | setFunction

Topics
“Interactively Configure C++ Interface”
“Programmatically Configure C++ Interface”

Introduced in R2021a

setData

Configure model data for C++ code generation

Syntax

```
setData(myCPPMappingObj, category, Name,Value)
```

Description

`setData(myCPPMappingObj, category, Name,Value)` configures code mapping information for the model data specified by `category`.

Examples

Configure data visibility of model parameters

Configure the data visibility of model parameters to be public.

Open the model. Use the `coder.mapping.api.get` function to access the `CodeMappingCPP` object associated with the model.

```
open_system('rtwdemo_cppclass');
cm = coder.mapping.api.get('rtwdemo_cppclass');
```

To view the data visibility of the model parameters, use the `getData` function.

```
value = getData(cm, 'ModelParameters', 'DataVisibility')
```

```
value =

    'private'
```

To configure the data visibility, specify the `'DataVisibility'` parameter using the `setData` function.

```
setData(cm, 'ModelParameters', 'DataVisibility', 'public');
```

Input Arguments

`myCPPMappingObj` **— C++ code mapping object**
`CodeMappingCPP` object

C++ code mapping object, returned by a call to either the `coder.mapping.utils.create` function or the `coder.mapping.api.get` function.

`category` **— Category of model data**
`'Inports'` | `'Outports'` | `'ModelParameters'` | `'ModelParameterArguments'` | `'InternalData'`

Category of model data to configure, specified as one of these categories.

Category	Description
'Inports'	Root-level input ports of a model, such as Inport and In Bus Element blocks.
'Outports'	Root-level output ports of a model, such as Outport and Out Bus Element blocks.
'ModelParameters'	Parameters that are defined within a model, such as parameters in the model workspace. Excludes model arguments.
'ModelParameterArguments'	Parameters in the model workspace configured as model arguments. These parameters are exposed at the model block to enable each model instance to provide its own value.
'InternalData'	Data elements that are internal to a model, such as block output signals, discrete block states, data stores, and zero-crossing signals.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `setData(myCPPMappingObj, 'Inports', 'DataVisibility', 'private')`

`MemberAccessMethod` — How methods are generated for data elements

`'Method'` | `'Inlined method'` | `'Structure-based method'` | `'Inlined structure-based method'` | `'None'`

How the methods, if any, are generated for the data elements. This configuration controls how application code can view and modify the class member data.

Member Access Method	Description
'Method'	The `get` and `set` methods for each element in the category appear in the generated class. Only `set` methods appear for model elements in the `Inports` category. Only `get` methods appear for model elements in the `Outports` category.
'Inlined method'	The `get` and `set` methods defined in their declarations appear for each element in a category in the generated class. Only `set` methods appear for model elements in the `Inports` category. Only `get` methods appear for model elements in the `Outports` category.
'Structure-based method'	Data elements appear as a structure in the class and aggregate `get` and `set` methods are generated for each category.

Member Access Method	Description
'Inlined structure-based method'	Data elements appear as a structure in the class. Aggregate get and set methods defined in their declaration are generated for each category.
'None'	If you configure the access of a model element category to None, get and set methods do not appear in the generated class. The application code can directly access the data.

Data Types: char | string

DataVisibility — Visibility of data category in generated code
'private' | 'public' | 'protected'

The visibility of the data category in generated code, specified as either private, public, or protected. If you configure data elements as public, they appear as public members of the generated class. If you configure elements as private, they appear as private members of the generated class.

Data Types: char | string

DataAccess — Access of model parameter arguments in generated code
'value' (default) | 'pointer'

The access of model parameter arguments in generated code, specified as either 'pointer' or 'value'. This parameter applies only to model parameter arguments whose data visibility is not set to 'Individual Arguments'.

Data Types: char | string

See Also

find | getClassName | getClassNamespace | getData | getFunction | setClassName | setClassNamespace | setFunction

Topics
"Interactively Configure C++ Interface"
"Programmatically Configure C++ Interface"

Introduced in R2021a

getFunction

Get code configuration from code mappings for model function

Syntax

```
propertyValue = getFunction(myCPPMappingObj,function,property)
```

Description

`propertyValue = getFunction(myCPPMappingObj,function,property)` returns the value of a property for the specified model function.

Examples

Get the method name for the initialize function for a model

Open the model. To access the `CodeMappingCPP` object associated with the model, use the `coder.mapping.api.get` function.

```
open_system('rtwdemo_cppclass');
cm = coder.mapping.api.get('rtwdemo_cppclass');
```

To access the method name of the initialize function for the model, use the `getFunction` function. If you did not specify a method name for the initialize function, the `getFunction` function returns an empty character vector, and the method name in the generated code uses the default method name.

```
value = getFunction(cm, 'Initialize', 'MethodName')
```

```
value =

  0×0 empty char array
```

Specify a method name for the initialize function by using the `setFunction` function.

```
setFunction(cm, 'Initialize', 'MethodName', 'my_$N')
```

The `getFunction` function now returns the specified method name.

```
value = getFunction(cm, 'Initialize', 'MethodName')
```

```
value =

    'my_$N'
```

Input Arguments

`myCPPMappingObj` — C++ code mapping object
`CodeMappingCPP` object

C++ code mapping object, returned by a call to either the `coder.mapping.utils.create` function or the `coder.mapping.api.get` function.

`function` — Model function
`Initialize` | `Terminate` | `Periodic:slIdentifier` | `Partition:slIdentifier` | `Reset:slIdentifier` | `ExportedFunction:slIdentifier` | `SimulinkFunction:slIdentifier`

Model function for which to return a code mapping property value. Specify one of the values listed in this table.

Type of Model Function	Value
Exported function	`ExportedFunction:`*`slIdentifier`*, where *`slIdentifier`* is the name of the function-call Inport block in the model
Initialize function	`Initialize`
Partition function	`Partition:`*`slIdentifier`*, where *`slIdentifier`* is a partition that was created explicitly from a block in the model and shown in the Simulink Schedule Editor (for example, `P1`)
Periodic multitasking function	`Periodic:`*`slIdentifier`*, where *`slIdentifier`* is an annotation that corresponds to the sample time period for a periodic or continuous rate of a multi-tasking model (for example, `D1`)
Periodic single-tasking function	`Periodic`
Reset function	`Reset:`*`slIdentifier`*, where *`slIdentifier`* is the name of the reset function in the model
Simulink function	`SimulinkFunction:`*`slIdentifier`*, where *`slIdentifier`* is the name of the Simulink function in the model
Terminate function	`Terminate`

For information about model partitioning, see “Create Partitions”.

Example: `'Periodic:D1'`

`property` — Code mapping property value to return
`MethodName` | `Arguments`

Code mapping property value to return. Specify one of the property names listed in this table.

Information to Return	Property Name
Name to use for the method in the generated code	`MethodName`
For periodic, single-tasking functions and Simulink functions, a string that shows the block names, argument names, type qualifiers, and order of arguments as they will appear in the generated code	`Arguments`

Example: `'MethodName'`

Output Arguments

`propertyValue` — Name of function, or argument specification
character vector | string scalar

Name of the function or argument specification returned as a character vector or string scalar.

Data Types: `char` | `string`

See Also

`find` | `getClassName` | `getClassNamespace` | `getData` | `setClassName` | `setClassNamespace` | `setData` | `setFunction`

Topics

"Interactively Configure C++ Interface"
"Programmatically Configure C++ Interface"

Introduced in R2021a

setFunction

Set code mapping information for model function

Syntax

```
setFunction(myCPPMappingObj,function,Name,Value)
```

Description

`setFunction(myCPPMappingObj,function,Name,Value)` sets code mapping information for the specified model function. Use this function to set the method name for a model function. For single-tasking periodic functions and Simulink functions, you can use this function to set the argument specification, including argument names, type qualifiers, and argument order.

Examples

Get the method name for the initialize function for a model

Open the model. To access the `CodeMappingCPP` object associated with the model, use the `coder.mapping.api.get` function.

```
open_system('rtwdemo_cppclass');
cm = coder.mapping.api.get('rtwdemo_cppclass');
```

To access the method name of the initialize function for the model, use the `getFunction` function. If you did not specify a method name for the initialize function, the `getFunction` function returns an empty character vector, and the method name in the generated code uses the default method name.

```
value = getFunction(cm, 'Initialize', 'MethodName')
```

```
value =

  0×0 empty char array
```

Specify a method name for the initialize function by using the `setFunction` function.

```
setFunction(cm, 'Initialize', 'MethodName', 'my_$N')
```

The `getFunction` function now returns the specified method name.

```
value = getFunction(cm, 'Initialize', 'MethodName')
```

```
value =

    'my_$N'
```

Input Arguments

myCPPMappingObj — C++ code mapping object
`CodeMappingCPP` object

C++ code mapping object, returned by a call to either the `coder.mapping.utils.create` function or the `coder.mapping.api.get` function.

`function` — Model function
`Initialize` | `Terminate` | `Periodic:slIdentifier` | `Partition:slIdentifier` | `Reset:slIdentifier` | `ExportedFunction:slIdentifier` | `SimulinkFunction:slIdentifier`

Model function for which to set code mapping property value. Specify one of the values listed in this table.

Type of Model Function	Value
Exported function	`ExportedFunction:`*`slIdentifier`*, where *`slIdentifier`* is the name of the function-call Inport block in the model
Initialize function	`Initialize`
Partition function	`Partition:`*`slIdentifier`*, where *`slIdentifier`* is a partition that was created explicitly from a block in the model and shown in the Simulink Schedule Editor (for example, `P1`)
Periodic multitasking function	`Periodic:`*`slIdentifier`*, where *`slIdentifier`* is an annotation that corresponds to the sample time period for a periodic or continuous rate of a multi-tasking model (for example, `D1`)
Periodic single-tasking function	`Periodic`
Reset function	`Reset:`*`slIdentifier`*, where *`slIdentifier`* is the name of the reset function in the model
Simulink function	`SimulinkFunction:`*`slIdentifier`*, where *`slIdentifier`* is the name of the Simulink function in the model
Terminate function	`Terminate`

For information about model partitioning, see “Create Partitions”.

Example: `'Periodic:D1'`

Name-Value Pair Arguments

Specify comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments as `Name1,Value1,…,NameN,ValueN`. The order of the name and value pair arguments does not matter.

Example: 'MethodName', 'my_$N'

`MethodName` — Name of method
character vector | string scalar

Name for the entry-point method in the generated C++ code, specified as a character vector or string scalar.

Data Types: `char` | `string`

`Arguments` — Argument specification
character vector | string scalar

Argument specification for the entry-point method in the generated C++ code, specified as a character vector or string scalar. The specification is a method prototype that shows argument names, type qualifiers, and argument order, for example, `'y =(u1, const *u2)'`.

For an entry-point method that does not return a value, do not include an output argument in the prototype specification. For example,

```
setFunction(cm, 'SimulinkFunction:f','Arguments','(u, *y)');
```

When the model function specified is a periodic function, you can also customize parameter names in the argument specification.

```
setFunction(cm,'Periodic:D1','Arguments',...
'(In1_1s & myParam1, In2_2s arg_In2_2s, * Out1 arg_Out1)');
```

Data Types: `char` | `string`

See Also

`find` | `getClassName` | `getClassNamespace` | `getData` | `getFunction` | `setClassName` | `setClassNamespace` | `setData`

Topics

"Interactively Configure C++ Interface"
"Programmatically Configure C++ Interface"

Introduced in R2021a

Simulink Coder Functions

addCompileFlags

Add compiler options to build information

Syntax

`addCompileFlags(buildinfo,options,groups)`

Description

`addCompileFlags(buildinfo,options,groups)` specifies the compiler options to add to the build information.

The function requires the *`buildinfo`* and *`options`* arguments. You can use an optional *`groups`* argument to group your options.

The code generator stores the compiler options in a build information object. The function adds options to the object based on the order in which you specify them.

Examples

Add Compiler Flags to OPTS Group

Add the compiler option `-O3` to the build information `myBuildInfo` and place the option in the group `OPTS`.

```
myBuildInfo = RTW.BuildInfo;
addCompileFlags(myBuildInfo,'-O3','OPTS');
```

Add Compiler Flags to OPT_OPTS Group

Add the compiler options `-Zi` and `-Wall` to the build information `myBuildInfo` and place the options in the group `OPT_OPTS`.

```
myBuildInfo = RTW.BuildInfo;
addCompileFlags(myBuildInfo,'-Zi -Wall','OPT_OPTS');
```

Add Compiler Flags to Build Information

For a non-makefile build environment, add the compiler options `-Zi`, `-Wall`, and `-O3` to the build information `myBuildInfo`. Place the options `-Zi` and `-Wall` in the group `Debug` and the option `-O3` in the group `MemOpt`.

```
myBuildInfo = RTW.BuildInfo;
addCompileFlags(myBuildInfo,{'-Zi -Wall' '-O3'}, ...
  {'Debug' 'MemOpt'});
```

Input Arguments

buildinfo — Build information object
object

`RTW.BuildInfo` object that contains information for compiling and linking generated code.

options — List of compiler options to add to build information
character vector | array of character vectors | string

You can specify the *options* argument as a character vector, as an array of character vectors, or as a string You can specify the *options* argument as multiple compiler flags within a single character vector, for example `'-Zi -Wall'`. If you specify the *options* argument as multiple character vectors, for example, `'-Zi -Wall'` and `'-O3'`, the *options* argument is added to the build information as an array of character vectors.

Example: `{'-Zi -Wall' '-O3'}`

groups — Optional group name for the added compiler options
character vector | array of character vectors | string

You can specify the *groups* argument as a character vector, as an array of character vectors, or as a string. If you specify multiple *groups*, for example, `'Debug' 'MemOpt'`, the function relates the *groups* to the *options* in order of appearance. For example, the *options* argument `{'-Zi -Wall' '-O3'}` is an array of character vectors with two elements. The first element is in the `'Debug'` group and the second element is in the `'MemOpt'` group.

Note The template makefile-based build process considers only compiler flags in the `'OPTS'`, `'OPT_OPTS'`, and `'OPTIMIZATION_FLAGS'` groups when generating the makefile.

For the build process to consider compiler flags in other groups, the template makefile must contain the token `|>COMPILE_FLAGS_OTHER<|`.

Example: `{'Debug' 'MemOpt'}`

See Also

`addDefines` | `addLinkFlags` | `getCompileFlags`

Topics

"Customize Post-Code-Generation Build Processing"
"Customize Template Makefiles"

Introduced in R2006a

addDefines

Add preprocessor macro definitions to build information

Syntax

`addDefines(buildinfo,macrodefs,groups)`

Description

`addDefines(buildinfo,macrodefs,groups)` specifies the preprocessor macro definitions to add to the build information.

The function requires the *`buildinfo`* and *`macrodefs`* arguments. You can use an optional *`groups`* argument to group your options.

The code generator stores the definitions in a build information object. The function adds definitions to the object based on the order in which you specify them.

Examples

Add Macro Definitions to OPTS Group

Add the macro definition `-DPRODUCTION` to the build information `myBuildInfo` and place the definition in the group `OPTS`.

```
myBuildInfo = RTW.BuildInfo;
addDefines(myBuildInfo,'-DPRODUCTION','OPTS');
```

Add Macro Definitions to OPT_OPTS Group

Add the macro definitions `-DPROTO` and `-DDEBUG` to the build information `myBuildInfo` and place the definitions in the group `OPT_OPTS`.

```
myBuildInfo = RTW.BuildInfo;
addDefines(myBuildInfo, ...
  '-DPROTO -DDEBUG','OPT_OPTS');
```

Add Macro Definitions to Build Information

For a non-makefile build environment, add the macro definitions `-DPROTO`, `-DDEBUG`, and `-DPRODUCTION` to the build information `myBuildInfo`. Place the definitions `-DPROTO` and `-DDEBUG` in the group `Debug` and the definition `-DPRODUCTION` in the group `Release`.

```
myBuildInfo = RTW.BuildInfo;
addDefines(myBuildInfo, ...
```

```
{'-DPROTO -DDEBUG' '-DPRODUCTION'}, ...
{'Debug' 'Release'});
```

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo`
object

Object contains information for compiling and linking generated code.

`macrodefs` — List of macro definitions to add to build information
character vector | array of character vectors | string

You can specify the *`macrodefs`* argument as a character vector, as an array of character vectors, or as a string. You can specify the *`macrodefs`* argument as multiple definitions within a single character vector, for example `'-DRT -DDEBUG'`. If you specify the *`macrodefs`* argument as multiple character vectors, for example `'-DPROTO -DDEBUG'` and `'-DPRODUCTION'`, the *`macrodefs`* argument is added to the build information as an array of character vectors.

Example: `{'-DPROTO -DDEBUG' '-DPRODUCTION'}`

`groups` — Optional group name for the added compiler options
character vector | array of character vectors | string

You can specify the *`groups`* argument as a character vector, as an array of character vectors, or as a string. If you specify multiple *`groups`*, for example `'Debug' 'Release'`, the function relates the *`groups`* to the *`macrodefs`* in order of appearance. For example, the *`macrodefs`* argument `{'-DPROTO -DDEBUG' '-DPRODUCTION'}` is an array of character vectors with two elements. The first element is in the `'Debug'` group and the second element is in the `'Release'` group.

Note The template makefile-based build process considers only macro definitions in the `'OPTS'`, `'OPT_OPTS'`, `'OPTIMIZATION_FLAGS'`, and `'Custom'` groups when generating the makefile.

For the build process to consider definitions in other groups, the template makefile must contain the token `|>DEFINES_OTHER<|`.

Example: `{'Debug' 'Release'}`

See Also

`addCompileFlags` | `addLinkFlags` | `getDefines`

Topics
"Customize Post-Code-Generation Build Processing"
"Customize Template Makefiles"

Introduced in R2006a

addIncludeFiles

Add include files to build information

Syntax

`addIncludeFiles(buildinfo,filenames,paths,groups)`

Description

`addIncludeFiles(buildinfo,filenames,paths,groups)` specifies included files and paths to add to the build information.

The function requires the *`buildinfo`* and *`filenames`* arguments. You can use an optional *`paths`* argument to specify the included file paths and use an optional *`groups`* argument to group your options.

The code generator stores the included file and path options in a build information object. The function adds options to the object based on the order in which you specify them.

Note The function does **not**:

- Add the file paths to the compiler search path. See `addIncludePaths`.
- Produce `#include` directives in the generated code

Examples

Add Included File to SysFiles Group

Add the include file `mytypes.h` to the build information `myBuildInfo` and place the file in the group `SysFiles`.

```
myBuildInfo = RTW.BuildInfo;
addIncludeFiles(myBuildInfo, ...
   'mytypes.h','/proj/src','SysFiles');
```

Add Included Files to AppFiles Group

Add the include files `etc.h` and `etc_private.h` to the build information `myBuildInfo`, and place the files in the group `AppFiles`.

```
myBuildInfo = RTW.BuildInfo;
addIncludeFiles(myBuildInfo, ...
   {'etc.h' 'etc_private.h'}, ...
   '/proj/src','AppFiles');
```

Add Included Files to SysFiles and AppFiles Groups

Add the include files `etc.h`, `etc_private.h`, and `mytypes.h` to the build information `myBuildInfo`. Group the files `etc.h` and `etc_private.h` with the character vector `AppFiles` and the file `mytypes.h` with the character vector `SysFiles`.

```
myBuildInfo = RTW.BuildInfo;
addIncludeFiles(myBuildInfo, ...
{'etc.h' 'etc_private.h' 'mytypes.h'}, ...
   '/proj/src', ...
   {'AppFiles' 'AppFiles' 'SysFiles'});
```

Add Included Files with Wildcard to HFiles Group

Add the include files (`.h` files identified with a wildcard character) in a specified folder to the build information `myBuildInfo`, and place the files in the group `HFiles`.

```
myBuildInfo = RTW.BuildInfo;
addIncludeFiles(myBuildInfo, ...
   '*.h', '/proj/src', 'HFiles');
```

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo`
object

Object provides information for compiling and linking generated code.

`filenames` — List of included files to add to build information
character vector | array of character vectors | string

You can specify the *`filenames`* argument as a character vector, as an array of character vectors, or as a string. If you specify the *`filenames`* argument as multiple character vectors, for example, `'etc.h' 'etc_private.h'`, the *`filenames`* argument is added to the build information as an array of character vectors.

If the dot delimiter (`.`) is present, the file name text can include wildcard characters. Examples are `'*.*'`, `'*.h'`, and `'*.h*'`.

The function removes duplicate included file entries with an exact match of a path and file name to a previously defined entry in the build information object.

Example: `'*.h'`

`paths` — List of included file paths to add to build information
character vector | array of character vectors | string

You can specify the *`paths`* argument as a character vector, as an array of character vectors, or as a string. If you specify a single path as a character vector, the function uses that path for all files. If you specify the *`paths`* argument as multiple character vectors, for example, `'/proj/src'` and `'/proj/inc'`, the *`paths`* argument is added to the build information as an array of character vectors.

Example: `'/proj/src'`

groups — Optional group name for the added included files
character vector | array of character vectors | string

You can specify the *groups* argument as a character vector, as an array of character vectors, or as a string. If you specify multiple *groups*, for example, `'AppFiles' 'AppFiles' 'SysFiles'`, the function relates the *groups* to the *filenames* in order of appearance. For example, the *filenames* argument `'etc.h' 'etc_private.h' 'mytypes.h'` is an array of character vectors with three elements. The first element is in the `'AppFiles'` group, the second element is in the `'AppFiles'` group, and the third element is in the `'SysFiles'` group.

Example: `'AppFiles' 'AppFiles' 'SysFiles'`

See Also

`addIncludePaths` | `addSourcePaths` | `findIncludeFiles` | `getIncludeFiles` | `updateFilePathsAndExtensions` | `updateFileSeparator`

Topics

"Customize Post-Code-Generation Build Processing"

Introduced in R2006a

addIncludePaths

Add include paths to build information

Syntax

`addIncludePaths(buildinfo,paths,groups)`

Description

`addIncludePaths(buildinfo,paths,groups)` specifies included file paths to add to the build information.

The function requires the *`buildinfo`* and *`paths`* arguments. You can use an optional *`groups`* argument to group your options.

The code generator stores the included file path options in a build information object. The function adds options to the object based on the order in which you specify them.

The function adds the file paths to the compiler search path.

The code generator does not check whether a specified path is valid.

Examples

Add Include File Path to Build Information

Add the include path `/etcproj/etc/etc_build` to the build information `myBuildInfo`.

```
myBuildInfo = RTW.BuildInfo;
addIncludePaths(myBuildInfo,...
   '/etcproj/etc/etc_build');
```

Add Include File Paths to a Group

Add the include paths `/etcproj/etclib` and `/etcproj/etc/etc_build` to the build information `myBuildInfo` and place the files in the group `etc`.

```
myBuildInfo = RTW.BuildInfo;
addIncludePaths(myBuildInfo,...
   {'/etcproj/etclib' '/etcproj/etc/etc_build'},'etc');
```

Add Include File Paths to Groups

Add the include paths `/etcproj/etclib`, `/etcproj/etc/etc_build`, and `/common/lib` to the build information `myBuildInfo`. Group the paths `/etc/proj/etclib` and `/etcproj/etc/etc_build` with the character vector `etc` and the path `/common/lib` with the character vector `shared`.

```
myBuildInfo = RTW.BuildInfo;
addIncludePaths(myBuildInfo,...
   {'/etc/proj/etclib' '/etcproj/etc/etc_build'...
   '/common/lib'}, {'etc' 'etc' 'shared'});
```

Input Arguments

buildinfo — Name of build information object returned by RTW.BuildInfo
object

paths — List of included file paths to add to build information
character vector | array of character vectors | string

You can specify the *paths* argument as a character vector, as an array of character vectors, or as a string. If you specify a single path as a character vector, the function uses that path for all files. If you specify the *paths* argument as multiple character vectors, for example, '/proj/src' and '/proj/inc', the *paths* argument is added to the build information as an array of character vectors.

The function removes duplicate include file path entries with an exact match of a path and file name to a previously defined entry in the build information object.

Example: '/proj/src'

groups — Optional group name for the added included files
character vector | array of character vectors | string

You can specify the *groups* argument as a character vector, as an array of character vectors, or as a string. If you specify multiple *groups*, for example, 'etc' 'etc' 'shared', the function relates the *groups* to the *paths* in order of appearance. For example, the *paths* argument '/etc/proj/etclib' '/etcproj/etc/etc_build' '/common/lib' is an array of character vectors with three elements. The first element is in the 'etc' group, the second element is in the 'etc' group, and the third element is in the 'shared' group.

Example: 'etc' 'etc' 'shared'

See Also

addIncludeFiles | addSourceFiles | addSourcePaths | getIncludePaths | updateFilePathsAndExtensions | updateFileSeparator

Topics

"Customize Post-Code-Generation Build Processing"

Introduced in R2006a

addLinkFlags

Add link options to build information

Syntax

`addLinkFlags(buildinfo,options,groups)`

Description

`addLinkFlags(buildinfo,options,groups)` specifies the linker options to add to the build information.

The function requires the *`buildinfo`* and *`options`* arguments. You can use an optional *`groups`* argument to group your options.

The code generator stores the linker options in a build information object. The function adds options to the object based on the order in which you specify them.

Examples

Add Linker Flags to OPTS Group

Add the linker `-T` option to the build information `myBuildInfo` and place the option in the group `OPTS`.

```
myBuildInfo = RTW.BuildInfo;
addLinkFlags(myBuildInfo,'-T','OPTS');
```

Add Linker Flags to OPT_OPTS Group

Add the linker options `-MD` and `-Gy` to the build information `myBuildInfo` and place the options in the group `OPT_OPTS`.

```
myBuildInfo = RTW.BuildInfo;
addLinkFlags(myBuildInfo,'-MD -Gy','OPT_OPTS');
```

Add Linker Flags to Build Information

For a non-makefile build environment, add the linker options `-MD`, `-Gy`, and `-T` to the build information `myBuildInfo`. Place the options `-MD` and`-Gy` in the group `Debug` and the option `-T` in the group`Temp`.

```
myBuildInfo = RTW.BuildInfo;
addLinkFlags(myBuildInfo, {'-MD -Gy' '-T'}, ...
   {'Debug' 'Temp'});
```

Input Arguments

buildinfo — Name of build information object returned by RTW.BuildInfo
object

options — List of linker options to add to build information
character vector | array of character vectors | string

You can specify the *options* argument as a character vector, as an array of character vectors, or as a string. You can specify the *options* argument as multiple compiler flags within a single character vector, for example `'-MD -Gy'`. If you specify the *options* argument as multiple character vectors, for example, `'-MD -Gy'` and `'-T'`, the *options* argument is added to the build information as an array of character vectors.

Example: `{'-MD -Gy' '-T'}`

groups — Optional group name for the added linker options
character vector | array of character vectors | string

You can specify the *groups* argument as a character vector, as an array of character vectors, or as a string. If you specify multiple *groups*, for example, `'Debug' 'Temp'`, the function relates the *groups* to the *options* in order of appearance. For example, the *options* argument `{'-MD -Gy' '-T'}` is an array of character vectors with two elements. The first element is in the `'Debug'` group and the second element is in the `Temp'` group.

Example: `{'Debug' 'Temp'}`

See Also

`addCompileFlags` | `addDefines` | `getLinkFlags`

Topics

"Customize Post-Code-Generation Build Processing"

Introduced in R2006a

addLinkObjects

Add link objects to build information

Syntax

```
addLinkObjects(buildinfo,linkobjs,paths,priority,precompiled,linkonly,groups)
```

Description

`addLinkObjects(buildinfo,linkobjs,paths,priority,precompiled,linkonly,groups)` specifies included files and paths to add to the build information.

The function requires the *`buildinfo`*, *`linkobs`*, and *`paths`* arguments. You can optionally select *`priority`* for link objects, select whether the objects are *`precompiled`*, select whether the objects are *`linkonly`* objects, and apply a *`groups`* argument to group your options.

The code generator stores the included link object and path options in a build information object. The function adds options to the object based on the order in which you specify them.

Examples

Add Link Objects to Build Information

Add the linkable objects `libobj1` and `libobj2` to the build information `myBuildInfo`. Mark both objects as link-only. Since individual priorities are not specified, the function adds the objects to the vector in the order specified.

```
myBuildInfo = RTW.BuildInfo;
addLinkObjects(myBuildInfo,{'libobj1' 'libobj2'}, ...
   {'/proj/lib/lib1' '/proj/lib/lib2'},1000, ...
   false,true);
```

Add Prioritized Link-Only Link Objects to Build Information

Add the linkable objects `libobj1` and `libobj2` to the build information `myBuildInfo`. Set the priorities of the objects to 26 and 10, respectively. Because `libobj2` is assigned the lower numeric priority value and has the higher priority, the function orders the objects such that `libobj2` precedes `libobj1` in the vector.

```
myBuildInfo = RTW.BuildInfo;
addLinkObjects(myBuildInfo, {'libobj1' 'libobj2'}, ...
   {'/proj/lib/lib1' '/proj/lib/lib2'},[26 10]);
```

Add Precompiled Link Objects to MyTest Group

Add the linkable objects `libobj1` and `libobj2` to the build information `myBuildInfo`. Set the priorities of the objects to 26 and 10, respectively. Mark both objects as precompiled. Group them under the name `MyTest`.

```
myBuildInfo = RTW.BuildInfo;
addLinkObjects(myBuildInfo,{'libobj1' 'libobj2'}, ...
   {'/proj/lib/lib1' '/proj/lib/lib2'},[26 10], ...
   true,false,'MyTest');
```

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo`
object

`linkobjs` — List of linkable object files to add to build information
character vector | array of character vectors | string

You can specify the *`linkobjs`* argument as a character vector, as an array of character vectors, or as a string. If you specify the *`linkobjs`* argument as multiple character vectors, for example, `'libobj1' 'libobj2'`, the *`linkobjs`* argument is added to the build information as an array of character vectors.

The function removes duplicate linkable object entries with an exact match of a path and file name to a previously defined entry in the build information object.

Example: `'libobj1'`

`paths` — List of included file paths to add to build information
character vector | array of character vectors | string

You can specify the *`paths`* argument as a character vector, as an array of character vectors, or as a string. If you specify a single path as a character vector, the function uses that path for all files. If you specify the *`paths`* argument as multiple character vectors, for example, `'/proj/lib/lib1'` and `'/proj/lib/lib2'`, the *`paths`* argument is added to the build information as an array of character vectors. The number of elements in *`paths`* must match the number of elements in the `linkobjs` argument.

Example: `'/proj/lib/lib1'`

`priority` — List of priority values for link objects to add to build information
`1000` (default) | numeric value | array of numeric values

A numeric value or an array of numeric values that indicates the relative priority of each specified link object. Lower values have higher priority.

Example: `1000`

`precompiled` — List of precompiled indicators for link objects to add to build information
false (default) | true | array of logical values

A logical value or an array of logical values that indicates whether each specified link object is precompiled. The logical value `true` indicates precompiled.

Example: `false`

`linkonly` — List of link-only indicators for link objects to add to build information
`false` (default) | true

A logical value or an array of logical values that indicates whether each specified link object is link-only (not precompiled). The logical value `true` indicates link-only. If *`linkonly`* is `true`, the value of the *`precompiled`* argument is ignored.

Example: `false`

`groups` — Optional group name for the added link object files
character vector | array of character vectors | string

You can specify the *`groups`* argument as a character vector, as an array of character vectors, or as a string. If you specify multiple *`groups`*, for example, `'MyTest1' 'MyTest2'`, the function relates the *`groups`* to the *`linkobjs`* in order of appearance. For example, the *`linkobjs`* argument `'libobj1' 'libobj2'` is an array of character vectors with two elements. The first element is in the `'MyTest1'` group, and the second element is in the `'MyTest2'` group.

Example: `'MyTest1' 'MyTest2'`

See Also

`addIncludePaths` | `addSourceFiles` | `addSourcePaths` | `findIncludeFiles` | `getIncludeFiles` | `updateFilePathsAndExtensions` | `updateFileSeparator`

Topics

"Customize Post-Code-Generation Build Processing"

Introduced in R2006a

addNonBuildFiles

Add nonbuild-related files to build information

Syntax

`addNonBuildFiles(buildinfo,filenames,paths,groups)`

Description

`addNonBuildFiles(buildinfo,filenames,paths,groups)` specifies nonbuild-related files and paths to add to the build information.

The function requires the *`buildinfo`* and *`filenames`* arguments. You can use an optional *`paths`* argument to specify the included file paths and use an optional *`groups`* argument to group your options.

The code generator stores the nonbuild-related file and path options in a build information object. The function adds options to the object based on the order in which you specify them.

Examples

Add Nonbuild File to DocFiles Group

Add the nonbuild-related file `readme.txt` to the build information `myBuildInfo`, and place the file in the group `DocFiles`.

```
myBuildInfo = RTW.BuildInfo;
addNonBuildFiles(myBuildInfo, ...
   'readme.txt','/proj/docs','DocFiles');
```

Add Nonbuild Files to DLLFiles Group

Add the nonbuild-related files `myutility1.dll` and `myutility2.dll` to the build information `myBuildInfo`, and place the files in the group `DLLFiles`.

```
myBuildInfo = RTW.BuildInfo;
addNonBuildFiles(myBuildInfo, ...
   {'myutility1.dll' 'myutility2.dll'}, ...
   '/proj/dlls','DLLFiles');
```

Add Nonbuild Files with Wildcard to DLLFiles Group

Add nonbuild-related files (`.dll` files identified with a wildcard character) in a specified folder to the build information `myBuildInfo`, and place the files in the group `DLLFiles`.

```
myBuildInfo = RTW.BuildInfo;
addNonBuildFiles(myBuildInfo, ...
   '*.dll','/proj/dlls','DLLFiles');
```

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo`
object

`filenames` — List of nonbuild-related files to add to build information
character vector | array of character vectors | string

You can specify the *`filenames`* argument as a character vector, as an array of character vectors, or as a string. If you specify the *`filenames`* argument as multiple character vectors, for example, `'etc.dll' 'etc_private.dll'`, the *`filenames`* argument is added to the build information as an array of character vectors.

If the dot delimiter (`.`) is present, the file name text can include wildcard characters. Examples are `'*.*'`, `'*.dll'`, and `'*.d*'`.

The function removes duplicate nonbuild-related file entries with an exact match of a path and file name to a previously defined entry in the build information object.

Example: `'*.dll'`

`paths` — List of nonbuild-related file paths to add to build information
character vector | array of character vectors | string

You can specify the *`paths`* argument as a character vector, as an array of character vectors, or as a string. If you specify a single path as a character vector, the function uses that path for all files. If you specify the *`paths`* argument as multiple character vectors, for example, `'/proj/dll'` and `'/proj/docs'`, the *`paths`* argument is added to the build information as an array of character vectors.

Example: `'/proj/dll'`

`groups` — Optional group name for the added nonbuild-related files
character vector | array of character vectors | string

You can specify the *`groups`* argument as a character vector, as an array of character vectors, or as a string. If you specify multiple *`groups`*, for example, `'DLLFiles' 'DLLFiles' 'DocFiles'`, the function relates the *`groups`* to the *`filenames`* in order of appearance. For example, the *`filenames`* argument `''myutility1.dll'' ''myutility2.dll'' 'readme.txt'` is an array of character vectors with three elements. The first element is in the `'DLLFiles'` group, the second element is in the `'DLLFiles'` group, and the third element is in the `'DocFiles'` group.

Example: `'DLLFiles' 'DLLFiles' 'DocFiles'`

See Also

`getNonBuildFiles`

Topics

"Customize Post-Code-Generation Build Processing"

Introduced in R2008a

addSourceFiles

Add source files to build information

Syntax

`addSourceFiles(buildinfo,filenames,paths,groups)`

Description

`addSourceFiles(buildinfo,filenames,paths,groups)` specifies source files and paths to add to the build information.

The function requires the *`buildinfo`* and *`filenames`* arguments. You can use an optional *`groups`* argument to group your options.

The code generator stores the source file and path options in a build information object. The function adds options to the object based on the order in which you specify them.

Examples

Add Source File to Drivers Group

Add the source file `driver.c` to the build information `myBuildInfo` and place the file in the group `Drivers`.

```
myBuildInfo = RTW.BuildInfo;
addSourceFiles(myBuildInfo,'driver.c', ...
   '/proj/src', 'Drivers');
```

Add Source Files to a Group

Add the source files `test1.c` and `test2.c` to the build information `myBuildInfo` and place the files in the group `Tests`.

```
myBuildInfo = RTW.BuildInfo;
addSourceFiles(myBuildInfo, ...
   {'test1.c' 'test2.c'}, ...
   '/proj/src','Tests');
```

Add Source Files to Groups

Add the source files `test1.c`, `test2.c`, and `driver.c` to the build information `myBuildInfo`. Group the files `test1.c` and `test2.c` with the character vector `Tests`. Group the file `driver.c` with the character vector `Drivers`.

```
myBuildInfo = RTW.BuildInfo;
addSourceFiles(myBuildInfo, ...
```

```
   {'test1.c' 'test2.c' 'driver.c'}, ...
   '/proj/src', ...
   {'Tests' 'Tests' 'Drivers'});
```

Add Source Files with Wildcard to CFiles Group

Add the `.c` files in a specified folder to the build information `myBuildInfo` and place the files in the group `CFiles`.

```
myBuildInfo = RTW.BuildInfo;
addIncludeFiles(myBuildInfo, ...
   '*.c','/proj/src','CFiles');
```

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo`
object

`filenames` — List of source files to add to build information
character vector | array of character vectors | string

You can specify the *`filenames`* argument as a character vector, as an array of character vectors, or as a string. If you specify the *`filenames`* argument as multiple character vectors, for example, `'etc.c' 'etc_private.c'`, the *`filenames`* argument is added to the build information as an array of character vectors.

If the dot delimiter (`.`) is present, the file name text can include wildcard characters. Examples are `'*.*'`, `'*.c'`, and `'*.c*'`.

The function removes duplicate included file entries with an exact match of a path and file name to a previously defined entry in the build information object.

Example: `'*.c'`

`paths` — List of source file paths to add to build information
character vector | array of character vectors | string

You can specify the *`paths`* argument as a character vector, as an array of character vectors, or as a string. If you specify a single path as a character vector, the function uses that path for all files. If you specify the *`paths`* argument as multiple character vectors, for example, `'/proj/src'` and `'/proj/inc'`, the *`paths`* argument is added to the build information as an array of character vectors.

Example: `'/proj/src'`

`groups` — Optional group name for the added source files
character vector | array of character vectors | string

You can specify the *`groups`* argument as a character vector, as an array of character vectors, or as a string. If you specify multiple *`groups`*, for example, `'Tests' 'Tests' 'Drivers'`, the function relates the *`groups`* to the *`filenames`* in order of appearance. For example, the *`filenames`* argument `'test1.c' 'test2.c' 'driver.c'` is an array of character vectors with three elements. The first element is in the `'Tests'` group, and the second element is in the `'Tests'` group, and the third element is in the `'Drivers'` group.

Example: 'Tests' 'Tests' 'Drivers'

See Also

addIncludeFiles | addIncludePaths | addSourcePaths | getSourceFiles | updateFilePathsAndExtensions | updateFileSeparator

Topics

"Customize Post-Code-Generation Build Processing"

Introduced in R2006a

addSourcePaths

Add source paths to build information

Syntax

`addSourcePaths(buildinfo,paths,groups)`

Description

`addSourcePaths(buildinfo,paths,groups)` specifies source file paths to add to the build information.

The function requires the *`buildinfo`* and *`paths`* arguments. You can use an optional *`groups`* argument to group your options.

The code generator stores the source file path options in a build information object. The function adds options to the object based on the order in which you specify them.

The code generator does not check whether a specified path is valid.

Note If you want to add source files and the corresponding file paths to build information, use the `addSourceFiles` function. Do not use `addSourcePaths`.

Examples

Add Source File Path to Build Information

Add the source path `/etcproj/etc/etc_build` to the build information `myBuildInfo`.

```
myBuildInfo = RTW.BuildInfo;
addSourcePaths(myBuildInfo, ...
   '/etcproj/etc/etc_build');
```

Add Source File Paths to a Group

Add the source paths `/etcproj/etclib` and `/etcproj/etc/etc_build` to the build information `myBuildInfo` and place the files in the group `etc`.

```
myBuildInfo = RTW.BuildInfo;
addSourcePaths(myBuildInfo, ...
   {'/etcproj/etclib' '/etcproj/etc/etc_build'},'etc');
```

Add Source File Paths to Groups

Add the source paths `/etcproj/etclib`, `/etcproj/etc/etc_build`, and `/common/lib` to the build information `myBuildInfo`. Group the paths `/etc/proj/etclib` and `/etcproj/etc/`

etc_build with the character vector etc and the path /common/lib with the character vector shared.

```
myBuildInfo = RTW.BuildInfo;
addSourcePaths(myBuildInfo,...
   {'/etc/proj/etclib' '/etcproj/etc/etc_build'...
   '/common/lib'}, {'etc' 'etc' 'shared'});
```

Input Arguments

buildinfo — Name of build information object returned by RTW.BuildInfo
object

paths — List of source file paths to add to build information
character vector | array of character vectors | string

You can specify the *paths* argument as a character vector, as an array of character vectors, or as a string. If you specify a single path as a character vector, the function uses that path for all files. If you specify the *paths* argument as multiple character vectors, for example, '/proj/src' and '/proj/inc', the *paths* argument is added to the build information as an array of character vectors.

The function removes duplicate source file path entries with an exact match of a path and file name to a previously defined entry in the build information object.

Example: '/proj/src'

groups — Optional group name for the added source files
character vector | array of character vectors | string

You can specify the *groups* argument as a character vector, as an array of character vectors, or as a string. If you specify multiple *groups*, for example, 'etc' 'etc' 'shared', the function relates the *groups* to the *paths* in order of appearance. For example, the *paths* argument '/etc/proj/etclib' '/etcproj/etc/etc_build' '/common/lib' is an array of character vectors with three elements. The first element is in the 'etc' group, the second element is in the 'etc' group, and the third element is in the 'shared' group.

Example: 'etc' 'etc' 'shared'

See Also

addIncludeFiles | addIncludePaths | addSourceFiles | getSourcePaths | updateFilePathsAndExtensions | updateFileSeparator

Topics
"Customize Post-Code-Generation Build Processing"

Introduced in R2006a

addTMFTokens

Add template makefile (TMF) tokens to build information

Syntax

`addTMFTokens(buildinfo,tokennames,tokenvalues,groups)`

Description

`addTMFTokens(buildinfo,tokennames,tokenvalues,groups)` specifies TMF tokens and values to add to the build information.

To provide build-time information to help customize makefile generation, call the `addTMFTokens` function inside a post-code-generation command. The tokens specified in the `addTMFTokens` function call must be handled in the template makefile (TMF) for the target selected for your . For example, you can call `addTMFTokens` in a post-code-generation command to add a TMF token named `|>CUSTOM_OUTNAME<|` with a token value that specifies an output file name for the build. To achieve the result you want, the TMF must apply an action with the value of `|>CUSTOM_OUTNAME<|`. (See "Examples" on page 2-0 .)

The `addTMFTokens` function adds specified TMF token names and values to the build information. The code generator stores the TMF tokens in a vector. The function adds the tokens to the end of the vector in the order that you specify them.

The function requires the *`buildinfo`*, *`tokennames`*, and *`tokenvalues`* arguments. You can use an optional *`groups`* argument to group your options. You can specify *`groups`* as a character vector or as an array of character vectors.

Examples

Add TMF Tokens to Build Information

Inside a post-code-generation command, add the TMF token `|>CUSTOM_OUTNAME<|` and its value to build information `myBuildInfo`, and place the token in the group `LINK_INFO`.

```
myBuildInfo = RTW.BuildInfo;
addTMFTokens(myBuildInfo, ...
             '|>CUSTOM_OUTNAME<|', 'foo.exe', 'LINK_INFO');
```

Apply Build Information as Tokens in TMF Build

In the TMF for the target selected for your , this code uses the token value to achieve the result that you want:

```
CUSTOM_OUTNAME = |>CUSTOM_OUTNAME<|
...
```

```
target:
$(LD) -o $(CUSTOM_OUTNAME) ...
```

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo`

object

`tokennames` — Specifies names of TMF tokens to add to the build information

character vector | array of character vectors | string

You can specify the *`tokennames`* argument as a character vector, as an array of character vectors, or as a string. If you specify the *`tokennames`* argument as multiple character vectors, for example, `'|>CUSTOM_OUTNAME<|' '|>COMPUTER<|'`, the *`tokennames`* argument is added to the build information as an array of character vectors.

Example: `'|>CUSTOM_OUTNAME<|' '|>COMPUTER<|'`

`tokenvalues` — Specifies TMF token values (for the added tokens) to add to the build information

character vector | array of character vectors | string

You can specify the *`tokenvalues`* argument as a character vector, as an array of character vectors, or as a string. If you specify the *`tokenvalues`* argument as multiple character vectors, for example, `'|>CUSTOM_OUTNAME<|' 'PCWIN64'`, the *`tokennames`* argument is added to the build information as an array of character vectors.

Example: `'foo.exe' 'PCWIN64'`

`groups` — Optional group name for the added TMF tokens

character vector | array of character vectors | string

You can specify the *`groups`* argument as a character vector, as an array of character vectors, or as a string. If you specify multiple *`groups`*, for example, `'LINK_INFO' 'COMPUTER_INFO'`, the function relates the *`groups`* to the *`tokennames`* in order of appearance. For example, the *`tokennames`* argument `'|>CUSTOM_OUTNAME<|' '|>COMPUTER<|'` is an array of character vectors with two elements. The first element is in the `'LINK_INFO'` group, and the second element is in the `'COMPUTER_INFO'` group.

Example: `'LINK_INFO' 'COMPUTER_INFO'`

See Also

Topics
"Customize Post-Code-Generation Build Processing"

Introduced in R2009b

buildStandaloneCoderAssumptions

Create application to check code generator assumptions

Syntax

`buildStandaloneCoderAssumptions(buildFolder)`

Description

`buildStandaloneCoderAssumptions(buildFolder)` creates an application for your target hardware to check code generator assumptions. The application checks that code generator assumptions based on model parameter settings or build configuration settings are correct with reference to the target hardware.

The function creates the target application in the `buildFolder\coderassumptions\standalone` subfolder.

Examples

Create Application to Check Code Generator Assumptions

For an example that shows how to create an application to check code generator assumptions, see “Check Code Generator Assumptions for Development Computer”.

Input Arguments

`buildFolder` — Build folder
character vector | string scalar

Path to the build folder that contains the generated code.

See Also

Topics
“Check Code Generation Assumptions”

Introduced in R2018b

codebuild

Compile and link generated code

Syntax

```
buildResults = codebuild(buildFolder)
codebuild(buildFolder, Name,Value)
```

Description

`buildResults = codebuild(buildFolder)` loads data from the `buildInfo.mat` file in `buildFolder`, generates a makefile in `buildFolder`, and uses the specified toolchain or template makefile to compile source code that is registered in the `RTW.BuildInfo` object. If the object is at the top of a hierarchy, the function performs the process for each object in the hierarchy.

The function saves compilation artifacts, including object code files, in `buildFolder`.

The function returns an object that contains the display output. To view the output, run `disp(buildResults)`.

`codebuild(buildFolder, Name,Value)` specifies additional options using one or more name-value pairs.

Examples

Relocate and Compile Generated Code

For an example that shows how to relocate and compile generated code in another development environment, see Compile Code in Another Development Environment.

Input Arguments

`buildFolder` — Build folder
character vector | string

Path to the build folder, which typically contains the generated source code. The folder must contain the `buildInfo.mat` file.

Example: `codebuild(`*`pathToCodeFolder`*`, 'BuildMethod', `*`myToolchain`*`)`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `codebuild(`*`pathToCodeFolder`*`, 'BuildMethod', `*`myToolchain`*`)`

`BuildMethod` — Build method
character vector | string

Use one these build methods:

- Toolchain — Specify the name of the toolchain, for example, `'GNU gcc/g++ | gmake (64-bit Linux)'`.
- Template makefile — Specify the path to the template makefile.
- CMake — Specify `'cmake'`, which generates `CMakeLists.txt` configuration files for the CMake build system. The argument value is case-insensitive. For example, you can also specify `'Cmake'` or `'CMake'`.

Example: `codebuild(pathToCodeFolder, 'BuildMethod', 'CMake')`

`BuildVariant` — Build variant
'STANDALONE_EXECUTABLE' | 'MODEL_REFERENCE_CODER' | 'MEX_FILE' | 'SHARED_LIBRARY' | 'STATIC_LIBRARY'

Specify the type of build output:

- `'STANDALONE_EXECUTABLE'` –– Generates a standalone, executable file.
- `'MODEL_REFERENCE_CODER'` –– Generates a static library.
- `'MEX_FILE'` –– Generates a MEX file. Use this value only for building a simulation target, for example, model reference simulation target (`ModelReferenceSimTarget`) and accelerator mode.
- `'SHARED_LIBRARY'` –– Generates a dynamic library.
- `'STATIC_LIBRARY'` –– Generates a static library.

Example: `codebuild(pathToCodeFolder, 'BuildVariant', 'SHARED_LIBRARY')`

Output Arguments

`buildResults` — Build results
object

Capture display output from build process. To view the display output, in the Command Window, run `disp(buildResults)`.

See Also
`slbuild`

Topics
Compile Code in Another Development Environment

Introduced in R2020b

coder.buildstatus.close

Close Build Status window

Syntax

```
coder.buildstatus.close()
coder.buildstatus.close(model)
coder.buildstatus.close(subsystem)
```

Description

`coder.buildstatus.close()` closes Build Status windows.

The Build Status window supports parallel builds of referenced model hierarchies. Do not use the Build Status window for serial builds.

`coder.buildstatus.close(model)` closes the Build Status window for `model`.

`coder.buildstatus.close(subsystem)` closes the Build Status window for `subsystem`.

Examples

Close Build Status Windows

Close Build Status windows that are open.

```
coder.buildstatus.close()
```

Close Build Status Window for a Model

After generating code for `rtwdemo_counter`, close the Build Status window for the model.

```
coder.buildstatus.close('rtwdemo_counter')
```

Close Build Status Window for a Subsystem

Close the Build Status window for the subsystem '`Amplifier`' in model '`rtwdemo_counter`'.

```
coder.buildstatus.close('rtwdemo_counter/Amplifier')
```

Input Arguments

`model` — Model name
character vector

Model name specified as a character vector

Example: '`rtwdemo_counter`'

Data Types: `char`

`subsystem` — Subsystem name
character vector

Subsystem name specified as a character vector

Example: '`rtwdemo_counter/Amplifier`'

Data Types: `char`

See Also

`coder.buildstatus.open` | `coder.report.close` | `slbuild`

Topics
"Monitor Parallel Building of Referenced Models"

Introduced in R2018a

coder.buildstatus.open

Open Build Status window

Syntax

```
coder.buildstatus.open(model)
coder.buildstatus.open(model,systemTarget)
```

Description

`coder.buildstatus.open(model)` opens the Build Status window for `model`.

The Build Status window supports parallel builds of referenced model hierarchies. Do not use the Build Status window for serial builds.

If the current working folder is the model build folder and the folder contains information from a previous parallel build, opening the Build Status window displays the previous build information. When you start a model parallel build, the current build information replaces the previous build information in the window.

`coder.buildstatus.open(model,systemTarget)` opens the Build Status window for `model` and displays the `model` tab. The available tabs are **Simulation Targets** and **Code Generation Targets**

Examples

Open Build Status Window for a Model

After generating code for model `'rtwdemo_parabuild_a_1'`, open the Build Status window for the model.

```
coder.buildstatus.open('rtwdemo_parabuild_a_1')
```

Open Build Status Window with Simulation Targets

Open the Build Status window for the model `'rtwdemo_parabuild_a_1'` and display the **Simulation Targets** tab.

```
coder.buildstatus.open('rtwdemo_parabuild_a_1','sim')
```

Input Arguments

`model` — Model name
character vector | string scalar

Model name specified as a character vector or a string scalar

Example: `'rtwdemo_parabuild_a_1'`

Data Types: `char` | `string`

`systemTarget` — System targets name
`sim` | `rtw`

System targets tab name specified as a character vector or string scalar, `sim` for **Simulation Targets** and `rtw` for **Code Generation Targets**. When build information is available for a system target from a previous build, the *`systemTarget`* argument directs the **Build Status** dialog box to display the tab for the system target. If this optional argument is omitted, when build information is available, dialog opens both the **Simulation Targets** tab and **Code Generation Targets** tab. If build information for a target is not available, the dialog does not open the corresponding system targets tab.

Example: `'rtw'`

Data Types: `char` | `string`

See Also

`coder.buildstatus.close` | `coder.report.open` | `slbuild`

Topics

"Monitor Parallel Building of Referenced Models"

Introduced in R2018a

coder.mapping.api.CodeMapping

Model data and function interface configuration for C code generation

Description

A code mappings object and related functions enable you to configure C code generation for data and functions of a Simulink model. For model data elements, code mappings associate data elements with configurations that consist of a storage class and storage class properties. For functions, code mappings associate entry-point functions with configurations that consist of a function customization template. Reduce the effort of preparing a model for C code generation by specifying default configurations for categories of data elements and functions across a model. Override default configurations by configuring data elements or functions individually. For smaller models, you can choose to configure each data element and function individually.

Creation

When you select a code generation app from the Apps tab in the Simulink Editor, such as the **Simulink Coder** or **Embedded Coder** app, the app creates a `coder.mapping.api.CodeMapping` object if code mappings do not already exist. The app creates code mappings based on code customization settings stored in the model active configuration set object. The configuration set object can specify memory sections for data and functions.

Create a `coder.mapping.api.CodeMapping` object programmatically by calling the function `coder.mapping.utils.create`. Create a mapping based on the active configuration set object or based on the default memory section and shared utility naming rule configurations of another configuration set object.

Object Functions

addSignal	Add block output signal to model code mappings
coder.mapping.api.get	Get code mappings for model
coder.mapping.utils.create	Create code mappings object for configuring data and function interface for C and C++ code generation
find	Get model elements for the category of model code mappings
getDataDefault	Get default storage class or storage class property setting for model data category
getDataStore	Get code configuration from code mappings for local or shared local data store
getFunction	Get code configuration from code mappings for model function
getFunctionDefault	Get default function customization template or memory section for model functions category
getInport	Get code configuration from code mappings for root-level inport
getModelParameter	Get code configuration from code mappings for model parameters
getOutport	Get code configuration from code mappings for root-level outport
getSignal	Get code configuration from code mappings for block output signal
getState	Get code configuration from code mappings for block state
removeSignal	Remove block output signal from model code mappings

setDataDefault	Set default storage class and storage class property values for model data category
setDataStore	Configure local or shared local data store for code generation
setFunction	Set code mapping information for model function
setFunctionDefault	Set default function customization template and memory section for model functions category
setInport	Configure root-level inports for code generation
setModelParameter	Configure model parameter for code generation
setOutport	Configure root-level outport for code generation
setSignal	Configure block signal data for code generation
setState	Configure block states for code generation

Examples

Create Environment to Configure Code Mappings for Model

For model `myConfigModel`, create the environment for configuring model data and functions for code generation. After calling this function, use calls to other functions listed under Object Functions to configure aspects of code generation for model interface elements.

```
coder.mapping.utils.create('myConfigModel');
```

See Also

`coder.mapping.api.CodeMappingCPP` | `coder.mapping.api.CoderDictionary` | `coder.mapping.api.get` | `coder.mapping.utils.create`

Topics

"C Code Generation Configuration for Model Interface Elements"
"Programmatically Configure C++ Interface"

Introduced in R2020b

addSignal

Add block output signal to model code mappings

Syntax

```
addSignal(myCodeMappingObj,portHandle)
addSignal(myCodeMappingObj,portHandle,Name,Value)
```

Description

`addSignal(myCodeMappingObj,portHandle)` adds signals specified by the block output port handles to the specified model code mappings.

This function does not apply to signals that originate from root-level Inport blocks.

`addSignal(myCodeMappingObj,portHandle,Name,Value)` adds signals specified by the block output port handles to the model code mappings. It configures the storage class and values of storage class properties that the code generator uses to produce C code for the signal data.

Examples

Add Block Output Signals to Model Code Mappings

For model `myConfigModel`, add the output signals of lookup table blocks `Table1D` and `Table2D` to the model code mappings. After creating the object `cm` by calling function `coder.mapping.api.get`, get handles to the output ports for lookup table blocks. Add the output signals to the code mappings with a call to `addSignal`.

```
cm = coder.mapping.api.get('myConfigModel');
lut1D_ports = get_param('myConfigModel/Table1D','PortHandles');
lut2D_ports = get_param('myConfigModel/Table2D','PortHandles');
lut1D_outPort = lut1D_ports.Outport;
lut2D_outPort = lut2D_ports.Outport;
addSignal(cm,[lut1D_outPort,lut2D_outPort]);
```

Add Block Output Signals to Model Code Mappings and Configure Storage Class for Signals

For model `myConfigModel`, add the output signals of lookup table blocks `Table1D` and `Table2D` to the model code mappings. After creating the object `cm` by calling function `coder.mapping.api.get`, get handles to the output ports for lookup table blocks. Add the output signals to the code mappings and set the storage class for the signals to `ExportedGlobal` with a call to `addSignal`.

```
cm = coder.mapping.api.get('myConfigModel');
lut1D_ports = get_param('myConfigModel/Table1D','PortHandles');
lut2D_ports = get_param('myConfigModel/Table2D','PortHandles');
lut1D_outPort = lut1D_ports.Outport;
```

```
lut2D_outPort = lut2D_ports.Outport;
addSignal(cm,[lut1D_outPort,lut2D_outPort],'StorageClass','ExportedGlobal');
```

Input Arguments

myCodeMappingObj — Code mapping object
CodeMapping object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

Example: myCM

portHandle — Output port handle of signal source block
port handle | array of port handles

Signal to add to the code mappings, specified as a handle of an output port of the source block of the signal. To specify multiple port handles, use an array.

Example: portHandle

Data Types: port_handle | array

Name-Value Pair Arguments

Specify comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments as `Name1,Value1,…,NameN,ValueN`. The order of the name and value pair arguments does not matter.

Example: 'StorageClass' 'ExportedGlobal'

StorageClass — Name of storage class
Auto | Bitfield | Dictionary default | ExportedGlobal | ExportToFile | FileScope | GetSet | ImportedExtern | ImportedExternPointer | ImportFromFile | Localizable | Model default | Struct | Volatile | storage class name

Storage class to set for the specified signals. The name of a predefined storage class or a storage class that is defined in the Embedded Coder Dictionary associated with the model. See “Configure Signal Data for C Code Generation”

Identifier — Name of variable
character vector | string scalar

Name for the variable that represents the signal data in the generated code.

Data Types: char | string

DefinitionFile — C source file
character vector | string scalar

File name for a C source file that contains definitions for global data read by data elements and external code. Applies to storage classes `ExportToFile` and `Volatile`.

Data Types: char | string

GetFunction — Name of get function
character vector | string scalar

Name of a `get` function that a data element calls in the generated code. Applies to storage class `GetSet`.

Data Types: `char` | `string`

`HeaderFile` — C header file
character vector | string scalar

File name for a C header file that contains declarations for global data read by data elements and external code. Applies to storage classes `ExportToFile`, `GetSet`, `ImportFromFile`, and `Volatile`.

Data Types: `char` | `string`

`Owner` — Owner of global data
character vector | string scalar

Name of the model that owns global data that is used by other models in the same model hierarchy. The code generated for the owner model includes the global data definition. Applies to storage classes `ExportToFile` and `Volatile`.

Data Types: `char` | `string`

`PreserveDimensions` — Boolean flag indicating whether to preserve dimensions of multidimensional arrays
`True` | `False`

When model configuration parameter **Array layout** is set to `Row-major`, a flag that indicates whether to preserve dimensions of a data element that is represented in generated code as a multidimensional array. Applies to storage classes `ExportToFile`, `ImportFromFile`, `Localizable`, and `Volatile`.

Data Types: `logical`

`SetFunction` — Name of set function
character string | string scalar

Name of a `set` function that a data element calls in the generated code. Applies to storage class `GetSet`.

Data Types: `char` | `string`

`StructName` — Name of structure
character vector | string scalar

Name that the code generator uses to identify the structure for a data element in the generated code. Applies to storage classes `Bitfield` and `Struct`.

Data Types: `char` | `string`

`storageClassPropertyName` — Value of storage class property
depends on property definition

Storage class property defined in the Embedded Coder Dictionary. Values that you can specify vary depending on the storage class definition.

See Also

`coder.mapping.api.CodeMapping` | `coder.mapping.api.get` | `find` | `getDataDefault` | `getSignal` | `removeSignal` | `setDataDefault` | `setSignal`

Topics

"Configure Signal Data for C Code Generation"
"C Code Generation Configuration for Model Interface Elements"

Introduced in R2020b

find

Package: `coder.mapping.api`

Get model elements for the category of model code mappings

Syntax

```
modelElementsFound= find(myCodeMappingObj,category)
modelElementsFound= find(myCodeMappingObj,category,Name,Value)
```

Description

`modelElementsFound= find(myCodeMappingObj,category)` returns the elements in the model code mappings of the specified category as an array of objects.

`modelElementsFound= find(myCodeMappingObj,category,Name,Value)` returns the elements in the model code mappings of the specified category that match specified property and value criteria.

Examples

Find Model Parameters in Code Mappings

In the model code mappings for model `myConfigModel`, find model workspace parameters.

```
cm = coder.mapping.api.get('myConfigModel');
inportBlkHandles = find(cm,'ModelParameters');
```

Find Inport Blocks That Have Storage Class Set to `Auto`

For model `myConfigModel`, find Inport blocks that have storage class set to `Auto`. For each Inport block found, change the storage class setting to `Model default`.

```
cm = coder.mapping.api.get('myConfigModel');
inportBlkHandles = find(cm,'Inports','StorageClass','Auto');
setInport(cm,inportBlkHandles,'StorageClass','Model default');
```

Find Entry-Point Functions That Have Memory Section Set to `Model default`

For model `myConfigModel`, find functions that are configured to use the model default setting for memory sections. For each function found, change the memory section setting to `None`.

```
cm = coder.mapping.api.get('myConfigModel');
functionObjects = find(cm,'Functions','MemorySection','Model default');
setFunction(cm,funcitonObjects,'MemorySection','None');
```

Input Arguments

myCodeMappingObj — Code mapping object
CodeMapping object

Code mapping object returned by a call to function coder.mapping.api.get.

Example: myCM

category — Model element category
DataStores | ExportedFunctions | ExternalParameterObjects | Inports | ModelParameters | Outports | PartitionFunctions | PartitionUpdateFunctions | PeriodicFunctions | PeriodicUpdateFunctions | ResetFunctions | Signals | SimulinkFunctions | States

Category of model elements that you search for in the model code mappings.

Example: 'Inports'

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments as Name1,Value1,…,NameN,ValueN. The order of the name and value pair arguments does not matter.

Example: 'Identifier','mp_table1'

StorageClass — Name of storage class
Auto | Bitfield | CompileFlag | Const | ConstVolatile | Define | Dictionary default | ExportedGlobal | ExportToFile | FileScope | GetSet | ImportedDefine | ImportedExtern | ImportedExternPointer | ImportFromFile | Localizable | Model default | Struct | Volatile | storage class name

Data element storage class to include in code mappings search criteria. The name of a predefined storage class or storage class that is defined in the Embedded Coder Dictionary associated with the model. Values that you can specify vary depending on the category that you specify.

Identifier — Code identifier
character vector | string scalar

Name that the code generator uses to identify a data element in generated code. Applies to storage classes other than Auto.

Data Types: char | string

DefinitionFile — C source file
character vector | string scalar

File name for a C source file that contains definitions for global data read by data elements and external code. Applies to storage classes Const, ConstVolatile, ExportToFile, and Volatile.

Data Types: char | string

`FunctionCustomizationTemplate` — Name of function customization template
character vector | string scalar

Name of a function customization template for a model that is defined in the Embedded Coder Dictionary.

Data Types: `char` | `string`

`FunctionName` — Name of entry-point function
character vector | string scalar

Name of an entry-point function generated for a model.

Data Types: `char` | `string`

`GetFunction` — Name of get function
character vector | string scalar

Name of a `get` function that a data element calls in the generated code. Applies to storage class `GetSet`.

Data Types: `char` | `string`

`HeaderFile` — C header file
character vector | string scalar

File name for a C header file that contains declarations for global data read by data elements and external code. Applies to storage classes `Const`, `ConstVolatile`, `Define`, `ExportToFile`, `GetSet`, `ImportedDefine`, `ImportFromFile`, and `Volatile`.

Data Types: `char` | `string`

`MemorySection` — Name of memory section
character vector | string scalar

Name of a memory section for a model that is defined in the Embedded Coder Dictionary.

Data Types: `char` | `string`

`Owner` — Owner of global data
character vector | string scalar

Name of the model that owns global data used by other models in the same model hierarchy. The code generated for the owner model includes the global data definition. Applies to storage classes `Const`, `ConstVolatile`, `ExportToFile`, and `Volatile`.

Data Types: `char` | `string`

`PreserveDimensions` — Boolean flag indicating whether to preserve dimensions of multidimensional arrays
`True` | `False`

When model configuration parameter **Array layout** is set to `Row-major`, a flag that indicates whether to preserve dimensions of a data element that is represented in generated code as a multidimensional array. Applies to storage classes `Const`, `ConstVolatile`, , `ExportToFile`, `FileScope`, `ImportFromFile`, `Localizable`, and `Volatile`.

Data Types: `logical`

`SetFunction` — Name of set function
character string | string scalar

Name of a `set` function that a data element calls in the generated code. Applies to storage class `GetSet`.

Data Types: `char` | `string`

`StructName` — Name of structure
character vector | string scalar

Name that the code generator uses to identify the structure for a data element in the generated code. Applies to storage classes `Bitfield` and `Struct`.

Data Types: `char` | `string`

`storageClassPropertyName` — Value of storage class property
depends on property definition

Storage class property defined in the model Embedded Coder Dictionary. Values that you can specify vary depending on the storage class definition.

Output Arguments

`modelElementsFound` — Model elements found
array | string vector

Model elements found, returned as an array or string vector of objects. Each object identifies a model element of the specified category. If you specify additional search criteria, the array or string vector includes objects for model elements of the specified category that meet the additional search criteria. The object returned for an element depends on the category that you specify.

Category	Type of Object Returned
Inports, Outports, and States	Block handle
Signals	Port handle
DataStores	Block handle
ModelParameters	Model parameter name
ExportedFunctions, Functions, PartitionFunctions, PartitionUpdateFunctions, PeriodicFunctions, PeriodicUpdateFunctions, ResetFunctions, and SimulinkFunctions,	Function

See Also
`coder.mapping.api.get`

Topics
"C Code Generation Configuration for Model Interface Elements"

Introduced in R2020b

coder.mapping.api.get

Get code mappings for model

Syntax

```
myCodeMappingObj = coder.mapping.api.get(model)
myCodeMappingObj = coder.mapping.api.get(dictionary)
myCodeMappingObj = coder.mapping.api.get(model,codeMappingType)
```

Description

`myCodeMappingObj = coder.mapping.api.get(model)` returns the active code mappings for the specified model as object `myCodeMappingObj`. Code mappings associate model data elements and functions with configurations for code generation. If a model has multiple code mappings, the active code mappings are the mappings associated with the active system target file.

If code mappings do not exist, Simulink returns an error. Simulink creates a code mappings object when you open a model in a coder app. If you have not opened a model in a coder app, you can create a code mappings object with a call to `coder.mapping.util.create`.

`myCodeMappingObj = coder.mapping.api.get(dictionary)` returns the active code mappings for the specified dictionary as object `myCodeMappingObj`. Code mappings associate data elements and functions in the data dictionary with configurations for code generation.

If code mappings do not exist, Simulink returns an error. Simulink creates a code mappings object when you open a model in a coder app. If you have not opened a model in a coder app, you can create a code mappings object with a call to `coder.mapping.util.create`.

`myCodeMappingObj = coder.mapping.api.get(model,codeMappingType)` returns the code mappings for your model that correspond to the specified code mapping type as object `myCodeMappingObj`. Code mappings enable you to associate a model with code generation configurations for C rapid prototyping (Simulink Coder and C language) and C production (Embedded Coder and C language) platforms. The code mappings type specifies your platform of interest. If a code mapping of the specified type does not exist, Simulink returns an error.

Examples

Get Code Mappings for Model

For model `myConfigModel`, return the code mappings to object `myCodeMappingObj`. Specify the returned object as the first argument in subsequent calls to other code mappings API functions. This example specifies the returned object in a call to `getInport`.

```
myCodeMappingObj = coder.mapping.api.get('myConfigModel');
myInput = getInport(myCodeMappingObj, 'In1', 'myConfigModel');
```

Create and Get Code Mappings for Model

In this example, for model myConfigModel, create the code mappings object myCodeMappingObj by calling coder.mapping.util.create. Then, return the object with a call to coder.mapping.api.get. This example specifies the returned object in a call to getInport.

```
myCodeMappingObj = coder.mapping.utils.create('myConfigModel');
cm = coder.mapping.api.get(myCodeMappingObj);
myInput = getInport(cm, 'In1', 'myConfigModel');
```

Get Simulink Coder C Code Mappings for Model

For model myConfigModel, return the Simulink Coder C language code mappings to object mySCCodeMappingObj. Specify the returned object as the first argument in subsequent calls to other code mappings API functions. This example specifies the returned object in a call to getInport.

```
mySCCodeMappingObj = coder.mapping.api.get('myConfigModel','SimulinkCoderC');
myInput = getInport(mySCCodeMappingObj, 'In1', 'myConfigModel');
```

Input Arguments

model — Name of model
handle | character vector | string scalar

Model for which to return code mappings object, specified as a handle or a character vector or string scalar representing the model name. The model must be loaded (for example, by using load_system) or open. Omit the .slx file extension.

Example: 'myConfigModel'

Data Types: char | string | model_handle

dictionary — Name of data dictionary
character vector | string scalar

Data dictionary for which to return code mappings object, specified as a character vector or string scalar representing the dictionary name.

Example: 'exCodeDefs.sldd'

Data Types: char | string

codeMappingType — Type of code mapping
SimulinkCoderC | EmbeddedCoderC | EmbeddedCoderCPP

The type of code mappings to return for the specified model or dictionary. Code mappings enable you to associate a model with code generation configurations for C rapid prototyping (Simulink Coder and C language) and C and C++ production (Embedded Coder and C and C++ language) platforms. The code mappings type specifies your platform of interest, SimulinkCoderC, EmbeddedCoderC, or EmbeddedCoderCPP. If a code mapping of the specified type does not exist, Simulink returns an error.

Example: 'SimulinkCoderC'

Output Arguments

`myCodeMappingObj` — Code mapping object
`CodeMapping` object | `CodeMappingCPP` object | `CoderDictionary` object

The model or dictionary code mappings, returned as a `CodeMapping` object, a `CodeMappingCPP` object, or a `CoderDictionary` object.

Output	Input Object	Code Mapping Type
`coder.mapping.api.CodeMapping`	Simulink model	`SimulinkCoderC` or `EmbeddedCoderC`
`coder.mapping.api.CodeMappingCPP`	Simulink model	`EmbeddedCoderCPP`
`coder.mapping.api.CoderDictionary`	Simulink data dictionary	N/A

See Also

`coder.mapping.api.CodeMapping` | `coder.mapping.api.CodeMappingCPP` | `coder.mapping.api.CoderDictionary` | `coder.mapping.utils.create`

Topics
"C Code Generation Configuration for Model Interface Elements"
"Programmatically Configure C++ Interface"

Introduced in R2020b

getDataDefault

Get default storage class or storage class property setting for model data category

Syntax

```
propertyValue = getDataDefault(myCodeMappingObj,category,property)
```

Description

`propertyValue = getDataDefault(myCodeMappingObj,category,property)` returns the value from the code mappings of the specified property for the specified data category.

Examples

Get Default Storage Class Setting for Root-Level Inports

From the model code mappings for model `myConfigModel`, get the default storage class setting for root-level inports.

```
cm = coder.mapping.api.get('myConfigModel');
defaultStorageClass = getDataDefault(cm,'Inports','StorageClass');
```

Get Default Header File Setting for Root-Level Inports

From the model code mappings for model `myConfigModel`, get the default header file setting for root-level inports.

```
cm = coder.mapping.api.get('myConfigModel');
defaultInputHeaderFile = getDataDefault(cm,'Inports','HeaderFile');
```

Input Arguments

`myCodeMappingObj` — Code mapping object
`CodeMapping` object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

Example: `myCM`

`category` — Model data element category
`Constants` | `ExternalParameterObjects` | `GlobalDataStores` | `Inports` | `InternalData` | `ModelParameters` | `ModelParameterArguments` | `Outports` | `SharedLocalDataStores`

Category of model data elements that you return a property value for.

Example: `'Inports'`

`property` — Code mapping property value to return
`StorageClass` | `Identifier` | `DefinitionFile` | `GetFunction` | `HeaderFile` | `MemorySection` | `Owner` | `PreserveDimensions` | `SetFunction` | `StructName` | storage class property name

Code mapping property that you return a value for. Specify one of these property names or a property name for a storage class defined in the Embedded Coder Dictionary associated with the model.

Information to Return	Property Name
Name of storage class	`StorageClass`
Name of variable for data element in the generated code	`Identifier`
Name of source definition file that contains definitions for global data that is read by the data element and external code	`DefinitionFile`
Name of `get` function called by code generated for the data element	`GetFunction`
Name of source header file that contains declarations for global data that is read by the model data element and external code	`HeaderFile`
Name of model for which the code generator places the definition for data element shared by multiple models in a model hierarchy	`Owner`
Boolean value indicating whether code generator preserves dimensions of data that is represented as a multidimensional array	`PerserveDimensions`
Name of `set` function called by code generated for data element	`SetFunction`
Name of structure in generated code for data element	`StructName`

Example: `'Identifier'`

Output Arguments

`propertyValue` — Name of storage class or value of storage class property
character vector | string scalar | `Auto` | `Bitfield` | `CompileFlag` | `Const` | `ConstVolatile` | `Define` | `Dictionary default` | `ExportedGlobal` | `ExportToFile` | `FileScope` | `GetSet` | `ImportedDefine` | `ImportedExtern` | `ImportedExternPointer` | `ImportFromFile` | `Localizable` | `Model default` | `Struct` | `Volatile`

The property value is one of these values depending on the category and property that you specify.

Property	Value Returned
`DefinitionFile`	Character vector or string scalar that names a C source file that contains definitions for global data read by data elements and external code. Applies to storage classes `Const`, `ConstVolatile`, `ExportToFile`, and `Volatile`.

Property	Value Returned
GetFunction	Character vector or string scalar that names a get function that a data element calls in the generated code. Applies to storage class GetSet.
HeaderFile	Character vector or string scalar that names a C header file that contains declarations for global data read by data elements and external code. Applies to storage classes Const, ConstVolatile, Define, ExportToFile, GetSet, ImportedDefine, ImportFromFile, and Volatile.
MemorySection	Character vector or string scalar that names a memory section for a model defined in the Embedded Coder Dictionary.
Owner	Character vector or string scalar that names the model that owns global data, which is used by other models in the same model hierarchy. The code generated for the owner model includes the global data definition. Applies to storage classes Const, ConstVolatile, ExportToFile, and Volatile.
PerserveDimensions	Boolean flag that indicates whether to preserve dimensions of a data element that is represented in generated code as a multidimensional array. Applies to storage classes Const, ConstVolatile, ExportToFile, FileScope, ImportFromFile, Localizable, and Volatile.
SetFunction	Character vector or string scalar that names a set function, which a data element calls in the generated code. Applies to storage class GetSet.
StorageClass	One of these values: Auto, Bitfield, CompileFlag, Const, ConstVolatile, Define, Dictionary default, ExportedGlobal, ExportToFile, FileScope, GetSet, ImportedDefine, ImportedExtern, ImportedExternPointer, ImportFromFile, Localizable, Model default, Struct, Volatile
StructName	Character vector or string scalar that names that names a structure for a data element in the generated code. Applies to storage classes Bitfield and Struct.

Data Types: char | string

See Also

coder.mapping.api.CodeMapping | coder.mapping.api.get | setDataDefault

Topics

"C Code Generation Configuration for Model Interface Elements"
"Configure Root-Level Inport Blocks for C Code Generation"
"Configure Root-Level Outport Blocks for C Code Generation"
"Configure Signal Data for C Code Generation"
"Configure Parameters for C Code Generation"
"Configure Block States for C Code Generation"
"Configure Data Stores for C Code Generation"
"Configure Default C Code Generation for Categories of Data Elements and Functions"

Introduced in R2020b

getDataStore

Get code configuration from code mappings for local or shared local data store

Syntax

```
propertyValue = getDataStore(myCodeMappingObj,dataStore,property)
```

Description

`propertyValue = getDataStore(myCodeMappingObj,dataStore,property)` returns the value of a code mapping property for the specified local or shared local data store. Use this function to return the storage class or the value of a storage class property configured for a local or shared local data store in a model.

Examples

Get Storage Class Configured for Local Data Store

From the model code mappings for model `myConfigModel`, get the name of the storage class that is configured for local data store `mode`.

```
cm = coder.mapping.api.get('myConfigModel');
scMode = getDataStore(cm,'mode','StorageClass');
```

Get Code Identifier Configured for Local Data Store

From the model code mappings for model `myConfigModel`, get the code identifier configured for the local data store `mode`.

```
cm = coder.mapping.api.get('myConfigModel');
idDSMmode = getDataStore(cm,'mode','Identifier');
```

Input Arguments

`myCodeMappingObj` — Code mapping object
`CodeMapping` object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

Example: `myCM`

`dataStore` — Block path, block handle, or name of data store
character vector | string scalar | block handle

Path of the Data Store Memory block for which to return the code mapping information, specified as a character vector or string scalar. Alternatively, you can specify a block handle or the name of the data

store. If you specify the name of a data store and that name is not unique within the model, Simulink returns an error that instructs you to specify the block path or handle.

Example: `blockHandle`

Data Types: `char` | `string` | `block_handle`

`property` — Code mapping property value to return
`StorageClass` | `Identifier` | `DefinitionFile` | `GetFunction` | `HeaderFile` | `Owner` | `PreserveDimensions` | `SetFunction` | `StructName` | storage class property name

Code mapping property for which to return a value. Specify one of these property names or a property name for a storage class defined in the Embedded Coder Dictionary associated with the model.

Information to Return	Property Name
Name of storage class	`StorageClass`
Name of variable for data store in the generated code	`Identifier`
Name of source definition file that contains definitions for global data that is read by the data store and external code	`DefinitionFile`
Name of `get` function called by code generated for the data store	`GetFunction`
Name of source header file that contains declarations for global data that is read by the data store and external code	`HeaderFile`
Name of memory section that contains data read by the state	`MemorySection`
Name of model for which the code generator places the definition for data store shared by multiple models in a model hierarchy	`Owner`
Boolean value indicating whether code generator preserves dimensions of an data store that is represented as a multidimensional array	`PerserveDimensions`
Name of `set` function called by code generated for data store	`SetFunction`
Name of structure in generated code for data store	`StructName`

Example: `'StorageClass'`

Output Arguments

`propertyValue` — Name of storage class or value of storage class property
character vector

Name of the storage class or value of the specified storage class property configured for the specified data store.

Data Types: char

See Also

coder.mapping.api.CodeMapping | coder.mapping.api.get | find | getDataDefault | setDataDefault | setDataStore

Topics

"Configure Data Stores for C Code Generation"
"C Code Generation Configuration for Model Interface Elements"

Introduced in R2020b

getFunction

Get code configuration from code mappings for model function

Syntax

```
propertyValue = getFunction(myCodeMappingObj,function,property)
```

Description

`propertyValue = getFunction(myCodeMappingObj,function,property)` returns the value of a property for the specified model function. Use this function to return the function customization template or memory section configured for a model function. For single-tasking periodic functions for which you previously set an argument specification and for Simulink functions, use this function to return the argument specification.

Examples

Get Function Name Configured for Initialize Function

For model `myConfigModel`, get the function name that is configured for the model initialize function from the model code mappings.

```
cm = coder.mapping.api.get('myConfigModel');
initFunctionName = getFunction(cm,'Initialize','FunctionName');
```

Get Memory Section Configured for Periodic Single-Tasking Function

For model `myConfigModel`, get the memory section that is configured for the model periodic single-tasking function from the model code mappings.

```
cm = coder.mapping.api.get('myConfigModel');
periodicFunctionMemSec = getFunction(cm,'Periodic','MemorySection');
```

Get Function Customization Template Configured for Periodic Multitasking Function for Sample Time D2

For model `myConfigModel`, get the function customization template that is configured for the model periodic multitasking function that corresponds to sample time D2 from the model code mappings.

```
cm = coder.mapping.api.get('myConfigModel');
periodicD2FunctionTemp = getFunction(cm,'Periodic:D2','FunctionCustomizationTemplate');
```

Get Argument Specification Configured for Simulink Function

For model myConfigModel, get the function argument specification (names, port type, qualifiers, and order) that is configured for the model Simulink function mySLFunc from the model code mappings.

```
cm = coder.mapping.api.get('myConfigModel');
mySLFuncArgs = getFunction(cm,'SimulinkFunction:mySLFunc','Arguments');
```

Input Arguments

myCodeMappingObj — Code mapping object
CodeMapping object

Code mapping object (model code mappings) returned by a call to function coder.mapping.api.get.

Example: myCM

function — Model function
Initialize | Terminate | Periodic:slIdentifier | Partition:slIdentifier | PeriodicUpdate:slIdentifier | PartitionUpdate:slIdentifier | Reset:slIdentifier | ExportedFunction:slIdentifier | SimulinkFunction:slIdentifier

Model function for which to return a code mapping property value. Specify one of the values listed in this table. If model configuration parameter **Single output/update function** is cleared, you can specify the update version of a partition, periodic multi-tasking, or periodic singletasking function.

Type of Model Function	Value
Exported function	ExportedFunction:*slIdentifier*, where *slIdentifier* is the name of the function-call Inport block in the model
Initialize function	Initialize
Partition function	Partition:*slIdentifier*, where *slIdentifier* is a partition that was created explicitly from a block in the model and shown in the Simulink Schedule Editor (for example, P1)
Partition update function	PartitionUpdate:*slIdentifier*, is a partition that was created explicitly from a block in the model and shown in the Simulink Schedule Editor (for example, P1)
Periodic multitasking function	Periodic:*slIdentifier*, where *slIdentifier* is an annotation that corresponds to the sample time period for a periodic or continuous rate of a multi-tasking model (for example, D1)
Periodic multitasking update function	PeriodicUpdate:*slIdentifier*, where *slIdentifier* is an annotation that corresponds to the sample time period for a periodic or continuous rate of a multi-tasking model (for example, D1)
Periodic single-tasking function	Periodic

Type of Model Function	Value
Periodic single-tasking update function	`PeriodicUpdate`
Reset function	`Reset:`*`slIdentifier`*, where *`slIdentifier`* is the name of the reset function in the model
Simulink function	`SimulinkFunction:`*`slIdentifier`*, where *`slIdentifier`* is the name of the Simulink function in the model
Terminate function	`Terminate`

For information about model partitioning, see “Create Partitions”.

Example: `'Periodic:D1'`

`property` — Code mapping property value to return
`FunctionCustomizationTemplate` | `MemorySection` | `FunctionName` | `Arguments`

Code mapping property value to return. Specify one of the property names listed in this table.

Information to Return	Property Name
Function customization template setting for the specified function	`FunctionCustomizationTemplate`
Memory section associated with the specified function	`MemorySection`
Name to use for the function in the generated code	`FunctionName`
For periodic, single-tasking functions and Simulink functions, a string that shows the names, type qualifiers, and order of arguments as they will appear in the generated code	`Arguments`

Example: `'FunctionCustomizationTemplate'`

Output Arguments

`propertyValue` — Name of function customization template, memory section, function, or argument specification
character vector | string scalar

Name of the function customization template, memory section, function, or argument specification returned as a character vector or string scalar.

Data Types: `char` | `string`

See Also

`coder.mapping.api.CodeMapping` | `coder.mapping.api.get` | `getFunctionDefault` | `setFunction` | `setFunctionDefault`

Topics
“Configure Names for Individual C Entry-Point Functions”
“Configure Name and Arguments for Individual Step Functions”

“Configure Default C Code Generation for Categories of Data Elements and Functions”
“C Code Generation Configuration for Model Interface Elements”

Introduced in R2020b

getFunctionDefault

Get default function customization template or memory section for model functions category

Syntax

`propertyValue = getFunctionDefault(myCodeMappingObj,category,property)`

Description

`propertyValue = getFunctionDefault(myCodeMappingObj,category,property)` returns the value of the specified property for the specified function category.

Examples

Get Default Function Customization Template Setting for Execution Functions

For model `myConfigModel`, get the default function customization template for execution functions from the model code mappings.

```
cm = coder.mapping.api.get('myConfigModel');
defaultFuncTemplateExe = getFunctionDefault(cm,'Execution','FunctionCustomizationTemplate');
```

Get Default Memory Section Setting for Initialize and Terminate Functions

For model `myConfigModel`, get the default function customization template for initialize and terminate functions from the model code mappings.

```
cm = coder.mapping.api.get('myConfigModel');
defaultMemSecExe = getFunctionDefault(cm,'InitializeTerminate','MemorySection');
```

Input Arguments

`myCodeMappingObj` — Code mapping object
`CodeMapping` object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

Example: `myCM`

`category` — Model function category
`InitializeTerminate` | `Execution` | `SharedUtility`

Category of model entry-point functions for which to return the default function customization template or memory section.

Example: `'Execution'`

`property` — Function customization template or memory section
`FunctionCustomizationTemplate` | `MemorySection`

`FunctionCustomizationTemplate` or `MemorySection` for which to return a value.

Example: `'FunctionCustomizationTemplate'`

Output Arguments

`propertyValue` — Name of function customization template or memory section
character vector | string scalar

Name of the function customization template or memory section.

Data Types: `char` | `string`

See Also

`coder.mapping.api.CodeMapping` | `coder.mapping.api.get` | `getFunction` | `setFunction` | `setFunctionDefault`

Topics
"Configure Default C Code Generation for Categories of Data Elements and Functions"
"C Code Generation Configuration for Model Interface Elements"

Introduced in R2020b

getInport

Get code configuration from code mappings for root-level inport

Syntax

```
propertyValue = getInport(myCodeMappingObj,inportBlock,property)
```

Description

`propertyValue = getInport(myCodeMappingObj,inportBlock,property)` returns the value of a code mapping property for the specified root-level Inport block. Use this function to return the storage class or the value of a storage class property configured for a root-level inport in a model.

Examples

Get Storage Class Configured for Root-Level Inport

From the model code mappings for model `myConfigModel`, get the name of the storage class that is configured for root-level inport `In1`.

```
cm = coder.mapping.api.get('myConfigModel');
scIn1 = getInport(cm,'In1','StorageClass');
```

Get Code Identifier Configured for Root-Level Inport

From the model code mappings for model `myConfigModel`, get the code identifier configured for root-level inport `In1`.

```
cm = coder.mapping.api.get('myConfigModel');
idIn1 = getInport(cm,'In1','Identifier');
```

Input Arguments

`myCodeMappingObj` — Code mapping object
`CodeMapping` object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

Example: `myCM`

`inportBlock` — Name, path, or handle of root-level inport
character vector | string scalar | block handle

Name, path, or handle of the root-level inport for which to return the code mapping information.

Example: `'In1'`

Data Types: `char` | `string` | `block_handle`

`property` — Code mapping property value to return
`StorageClass` | `Identifier` | `DefinitionFile` | `GetFunction` | `HeaderFile` | `Owner` | `PreserveDimensions` | `SetFunction` | `StructName` | storage class property name

Code mapping property for which to return a value. Specify one of these property names or a property name for a storage class defined in the Embedded Coder Dictionary associated with the model.

Information to Return	Property Name
Name of storage class	`StorageClass`
Name of variable for inport in the generated code	`Identifier`
Name of source definition file that contains definitions for global data that is read by the inport and external code	`DefinitionFile`
Name of `get` function called by code generated for the inport	`GetFunction`
Name of source header file that contains declarations for global data that is read by the inport and external code	`HeaderFile`
Name of model for which the code generator places the definition for inport shared by multiple models in a model hierarchy	`Owner`
Boolean value indicating whether code generator preserves dimensions of an inport that is represented as a multidimensional array	`PreserveDimensions`
Name of `set` function called by code generated for inport	`SetFunction`
Name of structure in generated code for inport	`StructName`

Example: `'StorageClass'`

Output Arguments

`propertyValue` — Name of storage class or value of storage class property
character vector

Name of the storage class or value of the specified storage class property configured for the specified root-level inport.

Data Types: `char`

See Also

`coder.mapping.api.CodeMapping` | `coder.mapping.api.get` | `find` | `getDataDefault` | `setDataDefault` | `setInport`

Topics

"Configure Root-Level Inport Blocks for C Code Generation"
"C Code Generation Configuration for Model Interface Elements"

Introduced in R2020b

getModelParameter

Get code configuration from code mappings for model parameters

Syntax

```
propertyValue = getModelParameter(myCodeMappingObj,modelParameter,property)
```

Description

`propertyValue = getModelParameter(myCodeMappingObj,modelParameter,property)` returns the value of a code mapping property for the specified model workspace parameter or model parameter argument. Use this function to return the storage class or the value of a storage class property configured for the parameter or parameter argument.

Examples

Get Storage Class Configured for Model Parameter

From the model code mappings for model `myConfigModel`, get the name of the storage class that is configured for model parameter `K1`.

```
cm = coder.mapping.api.get('myConfigModel');
scK1 = getModelParameter(cm,'K1','StorageClass');
```

Get Code Identifier Configured for Model Parameter

From the model code mappings for model `myConfigModel`, get the code identifier configured for model parameter `Table1`.

```
cm = coder.mapping.api.get('myConfigModel');
idTable1 = getModelParameter(cm,'Table1','Identifier');
```

Get Storage Class and Code Identifier Configured for Model Parameter Arguments

From the model code mappings for model `myConfigModel`, get the storage class and code identifier configured for model parameter arguments `LOWER` and `UPPER`.

```
cm = coder.mapping.api.get('myConfigModel');
scLOWER = getModelParameter(cm,'LOWER','StorageClass');
scUPPER = getModelParameter(cm,'UPPER','StorageClass');
```

```
idLOWER = getModelParameter(cm','LOWER','Identifier');
idUPPER = getModelParameter(cm','UPPER','Identifier');
```

Input Arguments

myCodeMappingObj — Code mapping object
CodeMapping object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

Example: myCM

modelParameter — Name of model parameter or model parameter argument
character vector | string scalar

Name of the model workspace parameter or model parameter argument for which to return the code mapping information.

Example: 'Table1'

Data Types: char | string

property — Code mapping property value to return
StorageClass | Identifier | DefinitionFile | GetFunction | ,HeaderFile | Owner | PreserveDimensions | SetFunction | StructName | storage class property name

Code mapping property for which to return a value. Specify one of these property names or a property name for a storage class defined in the Embedded Coder Dictionary associated with the model.

Information to Return	Property Name
Name of storage class	StorageClass
Name of variable for the parameter or parameter argument in the generated code	Identifier
Name of source definition file that contains definitions for global data that is read by the parameter or parameter argument and external code	DefinitionFile
Name of `get` function called by code generated for the parameter or parameter argument	GetFunction
Name of source header file that contains declarations for global data that is read by the parameter or parameter argument and external code	HeaderFile
Name of model for which the code generator places the definition for the parameter or parameter argument shared by multiple models in a model hierarchy	Owner

Information to Return	Property Name
Boolean value indicating whether code generator preserves dimensions of a parameter or parameter argument that is represented as a multidimensional array	`PerserveDimensions`
Name of `set` function called by code generated for the parameter or parameter argument	`SetFunction`
Name of structure in generated code for the parameter or parameter argument	`StructName`

Example: `'StorageClass'`

Output Arguments

`propertyValue` — Name of storage class or value of storage class property
character vector

Name of the storage class or value of the specified storage class property configured for model parameter or parameter argument .

Data Types: `char`

See Also

`coder.mapping.api.CodeMapping` | `coder.mapping.api.get` | `find` | `getDataDefault` | `setDataDefault` | `setModelParameter`

Topics

"Configure Parameters for C Code Generation"
"C Code Generation Configuration for Model Interface Elements"

Introduced in R2020b

getOutport

Get code configuration from code mappings for root-level outport

Syntax

```
propertyValue = getOutport(myCodeMappingObj,outportBlock,property)
```

Description

`propertyValue = getOutport(myCodeMappingObj,outportBlock,property)` returns the value of a code mapping property for the specified root-level Outport block. Use this function to return the storage class or the value of a storage class property configured for a root-level outport in a model.

Examples

Get Storage Class Configured for Root-Level Outport

From the model code mappings for model `myConfigModel`, get the name of the storage class that is configured for root-level outport `Out1`.

```
cm = coder.mapping.api.get('myConfigModel');
scOut1 = getOutport(cm,'Out1','StorageClass');
```

Get Code Identifier Configured for Root-Level Outport

From the model code mappings for model `myConfigModel`, get the code identifier configured for root-level outport `Out1`.

```
cm = coder.mapping.api.get('myConfigModel');
idOut1 = getOutport(cm,'Out1','Identifier');
```

Input Arguments

`myCodeMappingObj` — Code mapping object
`CodeMapping` object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

Example: `myCM`

`outportBlock` — Name, path, or handle of root-level outport
character vector | string scalar | block handle

Name, path, or handle of the root-level outport for which to return the code mapping information.

Example: `'Out1'`

Data Types: char | string

property — Code mapping property value to return
StorageClass | Identifier | DefinitionFile | GetFunction | HeaderFile | Owner |
PreserveDimensions | SetFunction | StructName | storage class property name

Code mapping property for which to return a value. Specify one of these property names or a property name for a storage class defined in the Embedded Coder Dictionary associated with the model.

Information to Return	Property Name
Name of storage class	StorageClass
Name of variable for root-level outport in the generated code	Identifier
Name of source definition file that contains definitions for global data that is read by the root-level outport and external code	DefinitionFile
Name of get function called by code generated for the root-level outport	GetFunction
Name of source header file that contains declarations for global data that is read by the root-level outport and external code	HeaderFile
Name of model for which the code generator places the definition for root-level outport shared by multiple models in a model hierarchy	Owner
Boolean value indicating whether code generator preserves dimensions of a root-level outport that is represented as a multidimensional array	PerserveDimensions
Name of set function called by code generated for root-level outport	SetFunction
Name of structure in generated code for root-level outport	StructName

Example: 'StorageClass'

Output Arguments

propertyValue — Name of storage class or value of storage class property
character vector

Name of the storage class or value of the specified storage class property configured for the specified root-level outport.

Data Types: char

See Also

coder.mapping.api.CodeMapping | coder.mapping.api.get | find | getDataDefault |
setDataDefault | setOutport

Topics

"Configure Root-Level Outport Blocks for C Code Generation"
"C Code Generation Configuration for Model Interface Elements"

Introduced in R2020b

getSignal

Get code configuration from code mappings for block output signal

Syntax

```
propertyValue = getSignal(myCodeMappingObj,portHandle,property)
```

Description

`propertyValue = getSignal(myCodeMappingObj,portHandle,property)` returns the value of a code mapping property for the signal specified by a block output port handle. Use this function to return the name of the storage class or the value of a storage class property configured for a signal.

This function does not apply to signals that originate from root-level Inport blocks. For signals that originate from root-level Inport blocks, see `getInport`.

Examples

Get Storage Class Configured for Block Output Signal

From the model code mappings for model `myConfigModel`, get the name of the storage class that is configured for the output signal of lookup table block `Table1D`. After creating the object `cm` by calling function `coder.mapping.api.get`, get the handle to the output signals for the lookup table block. Get the storage class configured for the output port by calling `getSignal`.

```
cm = coder.mapping.api.get('myConfigModel');
lut1D_ports = get_param('myConfigModel/Table1D','PortHandles');
lut1D_outPort = lut1D_ports.Outport;
scTable1D = getSignal(cm,lut1D_outPort,'StorageClass');
```

Get Code Identifiers Configured for Block Output Signals

From the model code mappings for model `myConfigModel`, get the code identifiers that are configured for output signals of lookup table blocks `Table1D` and `Table2D`. After creating the object `cm` by calling function `coder.mapping.api.get`, get the handles to the output ports for the lookup table blocks. Get the code identifiers configured for the output ports by calling `getSignal`.

```
cm = coder.mapping.api.get('myConfigModel');
lut1D_ports = get_param('myConfigModel/Table1D','PortHandles');
lut2D_ports = get_param('myConfigModel/Table2D','PortHandles');
lut1D_outPort = lut1D_ports.Outport;
lut2D_outPort = lut2D_ports.Outport;
```

```
idTable1D = getSignal(cm,lut1D_outPort,'Identifier');
idTable2D = getSignal(cm,lut3D_outPort,'Identifier');
```

Input Arguments

myCodeMappingObj — Code mapping object
CodeMapping object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

Example: myCM

portHandle — Output port handle of signal source block
port handle

Block output signals for which to return signal code mapping information.

Example: portHandle

Data Types: port_handle

property — Code mapping property value to return
StorageClass | Identifier | DefinitionFile | GetFunction | HeaderFile | Owner | PreserveDimensions | SetFunction | StructName | storage class property name

Code mapping property for which to return a value. Specify one of these property names or a property name for a storage class defined in the Embedded Coder Dictionary associated with the model.

Information to Return	Property Name
Name of storage class	StorageClass
Name of variable for signal data in the generated code	Identifier
Name of source definition file that contains definitions for global data that is read by the signal data and external code	DefinitionFile
Name of `get` function called by code generated for the signal data	GetFunction
Name of source header file that contains declarations for global data that is read by the signal data and external code	HeaderFile
Name of model for which the code generator places the definition for signal data shared by multiple models in a model hierarchy	Owner
Boolean value indicating whether code generator preserves dimensions of signal data that is represented as a multidimensional array	PerserveDimensions
Name of `set` function called by code generated for the signal data	SetFunction

Information to Return	Property Name
Name of structure in generated code for signal data	`StructName`

Example: `'StorageClass'`

Output Arguments

`propertyValue` — Name of storage class or value of storage class property
character vector

Name of the storage class or value of the specified storage class property configured for the specified signal.

Data Types: `char`

See Also

`addSignal` | `coder.mapping.api.CodeMapping` | `coder.mapping.api.get` | `find` | `getDataDefault` | `removeSignal` | `setDataDefault` | `setSignal`

Topics
"Configure Signal Data for C Code Generation"
"C Code Generation Configuration for Model Interface Elements"

Introduced in R2020b

getState

Get code configuration from code mappings for block state

Syntax

```
propertyValue = getState(myCodeMappingObj,block,property)
```

Description

`propertyValue = getState(myCodeMappingObj,block,property)` returns the value of a code mapping property for the state of the specified block. Use this function to return the storage class or the value of a storage class property configured for a block state.

Examples

Get Storage Class Configured for Block State

From the model code mappings for model `myConfigModel`, get the name of the storage class that is configured for state `X` of Unit Delay block `Delay`.

```
cm = coder.mapping.api.get('myConfigModel');
scX = getState(cm,'myConfigModel/Delay','StorageClass');
```

Get Code Identifier Configured for State of Unit Delay Block

From the model code mappings for model `myConfigModel`, get the code identifier that is configured for state `X` of Unit Delay block `Delay`.

```
cm = coder.mapping.api.get('myConfigModel');
scX = getState(cm,blockHandle,'Identifier');
```

Input Arguments

`myCodeMappingObj` — Code mapping object
`CodeMapping` object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

Example: `myCM`

`block` — Path or handle of block
character vector | string scalar | block handle

Path of the block for which to return the state code mapping information, specified as a character vector or string scalar. Alternatively, you can specify a block handle.

Example: `blockHandle`

Data Types: char | string | block_handle

property — Code mapping property value to return
StorageClass | Identifier | DefinitionFile | GetFunction | HeaderFile | MemorySection | Owner | PreserveDimensions | SetFunction | StructName | storage class property name

Code mapping property for which to return a value. Specify one of these property names or a property name for a storage class defined in the Embedded Coder Dictionary associated with the model.

Information to Return	Property Name
Name of storage class	StorageClass
Name of variable for state in the generated code	Identifier
Name of source definition file that contains definitions for global data that is read by the state and external code	DefinitionFile
Name of get function called by code generated for the state	GetFunction
Name of source header file that contains declarations for global data that is read by the state and external code	HeaderFile
Name of memory section that contains data read by the state	MemorySection
Name of model for which the code generator places the definition for state shared by multiple models in a model hierarchy	Owner
Boolean value indicating whether code generator preserves dimensions of an state that is represented as a multidimensional array	PerserveDimensions
Name of set function called by code generated for state	SetFunction
Name of structure in generated code for state	StructName

Example: 'StorageClass'

Output Arguments

propertyValue — Name of storage class or value of storage class property
character vector

Name of the storage class or value of the specified storage class property configured for the specified block state, returned as a character vector.

Data Types: char

See Also

coder.mapping.api.CodeMapping | coder.mapping.api.get | find | getDataDefault | setDataDefault | setState

Topics

"Configure Block States for C Code Generation"

"C Code Generation Configuration for Model Interface Elements"

Introduced in R2020b

removeSignal

Remove block output signal from model code mappings

Syntax

`removeSignal(myCodeMappingObj,portHandle)`

Description

`removeSignal(myCodeMappingObj,portHandle)` removes signals specified by the block output port handles from the specified model code mappings.

This function does not apply to signals that originate from root-level Inport blocks.

Examples

Remove Block Output Signals from Model Code Mappings

From the model code mappings for model `myConfigModel`, remove the output signals of lookup table blocks `Table1D` and `Table2D`. After creating the object `cm` by calling function `coder.mapping.api.get`, get handles to the output ports for lookup table blocks. Remove the output signals from the code mappings by calling `removeSignal`.

```
cm = coder.mapping.api.get('myConfigModel');
lut1D_ports = get_param('myConfigModel/Table1D','PortHandles');
lut2D_ports = get_param('myConfigModel/Table2D','PortHandles');
lut1D_outPort = lut1D_ports.Outport;
lut2D_outPort = lut2D_ports.Outport;
removeSignal(cm,[lut1D_outPort,lut2D_outPort]);
```

Input Arguments

`myCodeMappingObj` — Code mapping object
`CodeMapping` object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

Example: `myCM`

`portHandle` — Output port handle of signal source block
port handle | cell array of port handles

Signal to remove from the code mappings, specified as a handle of an output port of the source block of the signal. To specify multiple port handles, use a cell array.

Example: `portHandle`

Data Types: `port_handle` | `cell`

See Also

addSignal | coder.mapping.api.CodeMapping | coder.mapping.api.get | find | getDataDefault | getSignal | removeSignal | setDataDefault | setSignal

Topics

"Configure Signal Data for C Code Generation"
"C Code Generation Configuration for Model Interface Elements"

Introduced in R2020b

setDataDefault

Set default storage class and storage class property values for model data category

Syntax

```
setDataDefault(myCodeMappingObj,category,Name,Value)
```

Description

`setDataDefault(myCodeMappingObj,category,Name,Value)` sets the default storage class and storage class property values in the code mappings for the specified category of model data.

Examples

Configure Default Representation for Model Workspace Parameters in Generated Code as Unstructured Global Variables

In the model code mappings for model `myConfigModel`, configure the default representation of model workspace parameters in generated code as unstructured global variables by setting the default storage class to `ExportedGlobal`.

```
cm = coder.mapping.api.get('myConfigModel');
setDataDefault(cm,'ModelParameters','StorageClass','ExportedGlobal');
```

Configure Code Generator to Write Root-Level Outport Data to External Files by Default

In the model code mappings for model `myConfigModel`, configure the code generator to write root-level outport data to separate global variables declared and defined in external files `myextheader.h` and `myextsrc.c`.

```
cm = coder.mapping.api.get('myConfigModel');
setDataDefault(cm,'Outports','StorageClass','ExportToFile',...
  'HeaderFile',myextheader.h','DefinitionFile','myextsrc.c');
```

Input Arguments

`myCodeMappingObj` — Code mapping object
`CodeMapping` object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

Example: `myCM`

`category` — Model element category
`Constants` | `ExternalParameterObjects` | `GlobalDataStores` | `Inports` | `InternalData` | `ModelParameters` | `ModelParameterArguments` | `Outports` | `SharedLocalDataStores`

Category of model data element for which to set the storage class and storage class properties.

Example: 'Inports'

Name-Value Pair Arguments

Specify comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments as Name1,Value1,…,NameN,ValueN. The order of the name and value pair arguments does not matter.

Example: 'StorageClass','ExportedGlobal'

StorageClass — Name of storage class
Auto | Bitfield | CompileFlag | Const | ConstVolatile | Define | Dictionary default | ExportedGlobal | ExportToFile | FileScope | GetSet | ImportedDefine | ImportedExtern | ImportedExternPointer | ImportFromFile | Localizable | Model default | Struct | Volatile | storage class name

Storage class to set for the specified data element category. The name of a predefined storage class or storage class that is defined in the Embedded Coder Dictionary associated with the model. Values that you can specify vary depending on the category that you specify. See "Choose Storage Class for Controlling Data Representation in Generated Code".

Example: 'StorageClass','ImportedExtern'

DefinitionFile — C source file
character vector | string scalar

File name for a C source file that contains definitions for global data read by data elements and external code. Applies to storage classes Const, ConstVolatile, ExportToFile, and Volatile.

Example: 'DefinitionFile','myDataDefs.c'

Data Types: char | string

GetFunction — Name of get function
character vector | string scalar

Name of a get function that a data element calls in the generated code. Applies to storage class GetSet.

Example: 'GefFunction','myDataGetFunction'

Data Types: char | string

HeaderFile — C header file
character vector | string scalar

File name for a C header file that contains declarations for global data read by data elements and external code. Applies to storage classes Const, ConstVolatile, Define, ExportToFile, GetSet, ImportedDefine, ImportFromFile, and Volatile.

Example: 'HeaderFile','myDataDecl.h'

Data Types: char | string

MemorySection — Name of memory section
character vector | string scalar

Name of a memory section that is defined in the Embedded Coder Dictionary associated with the model.

Example: 'MemorySection','myFastMem'

Data Types: char | string

Owner — Owner of global data
character vector | string scalar

Name of the model that owns global data, which is used by other models in the same model hierarchy. The code generated for the model that owns the data includes the global data definition. Applies to storage classes Const, ConstVolatile, ExportToFile, and Volatile.

Example: 'Owner','myModelA'

Data Types: char | string

PreserveDimensions — Boolean flag indicating whether to preserve dimensions of multidimensional arrays
True | False

When model configuration parameter **Array layout** is set to Row-major, a flag that indicates whether to preserve dimensions of a data element represented in generated code as a multidimensional array. Applies to storage classes Const, ConstVolatile, , ExportToFile, FileScope, ImportFromFile, Localizable, and Volatile.

Example: 'PreserveDimensions','True'

Data Types: logical

SetFunction — Name of set function
character string | string scalar

Name of a set function that a data element calls in the generated code. Applies to storage class GetSet.

Example: 'SetFunction','myDataSetFunction'

Data Types: char | string

StructName — Name of structure
character vector | string scalar

Name that the code generator uses to identify the structure for a data element in the generated code. Applies to storage classes Bitfield and Struct.

Example: 'StructName','myDataStruct'

storageClassPropertyName — Value of storage class property
depends on property definition

Storage class property defined in the Embedded Coder Dictionary. Values that you can specify vary depending on the storage class definition.

Data Types: char | string

See Also

coder.mapping.api.CodeMapping | coder.mapping.api.get | getDataDefault

Topics
"C Code Generation Configuration for Model Interface Elements"
"Configure Root-Level Inport Blocks for C Code Generation"
"Configure Root-Level Outport Blocks for C Code Generation"
"Configure Signal Data for C Code Generation"
"Configure Parameters for C Code Generation"
"Configure Block States for C Code Generation"
"Configure Data Stores for C Code Generation"
"Configure Default C Code Generation for Categories of Data Elements and Functions"

Introduced in R2020b

setDataStore

Configure local or shared local data store for code generation

Syntax

```
setDataStore(myCodeMappingObj,dataStore,Name,Value)
```

Description

`setDataStore(myCodeMappingObj,dataStore,Name,Value)` configures the specified local or shared local data store for code generation. Use this function to map a local or shared local data store to the storage class and storage class property settings that the code generator uses to produce C code for that data store.

Examples

Configure Storage Class for Local Data Store

In the model code mappings for model `myConfigModel`, set the storage class for local data store `mode` to `ExportedGlobal`.

```
cm = coder.mapping.api.get('myConfigModel');
setDataStore(cm,'mode','StorageClass','ExportedGlobal');
```

Configure Storage Class for Local and Shared Local Data Stores in Model to `Model default`

In the model code mappings for model `myConfigModel`, set the storage class for local and shared local data stores throughout the model to `Model default`.

```
cm = coder.mapping.api.get('myConfigModel');
dsmHandles = find(cm,'DataStores')
setDataStores(cm,dsmHandles,'StorageClass','Model default');
```

Configure Code Identifier for Local Data Store

In the model code mappings, for model `myConfigModel`, set the code identifier for local data store `mode` to `ds_mode`.

```
cm = coder.mapping.api.get('myConfigModel');
setDataStore(cm,'mode','Identifier','ds_mode');
```

Input Arguments

`myCodeMappingObj` — Code mapping object
`CodeMapping` object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

Example: `myCM`

`dataStore` — Block path, block handle, or name of data store
character vector | string scalar | block handle | array of character vectors | array of string scalars | array of block handles

Path of the Data Store Memory block for which to return the code mapping information, specified as a character vector or string scalar. Alternatively, you can specify a block handle or the name of the data store. If you specify the name of a data store and that name is not unique within the model, Simulink returns an error that instructs you to specify the block path or handle. To specify multiple data stores, use an array.

Example: `blockHandle`

Data Types: `char` | `string` | `block_handle` | `array`

Name-Value Pair Arguments

Specify comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments as `Name1,Value1,…,NameN,ValueN`. The order of the name and value pair arguments does not matter.

Example: 'StorageClass' 'ExportedGlobal'

`StorageClass` — Name of storage class
`Auto` | `Bitfield` | `Dictionary default` | `ExportedGlobal` | `ExportToFile` | `FileScope` | `GetSet` | `ImportedExtern` | `ImportedExternPointer` | `ImportFromFile` | `Localizable` | `Model default` | `Struct` | `Volatile` | storage class name

Storage class to set for the specified data store. The name of a predefined storage class or storage class that is defined in the Embedded Coder Dictionary associated with the model. See “Configure Data Stores for C Code Generation”.

`Identifier` — Name of variable
character vector | string scalar

Name for the variable that represents the local data store in the generated code.

Data Types: `char` | `string`

`DefinitionFile` — C source file
character vector | string scalar

File name for a C source file that contains definitions for global data read by data elements and external code. Applies to storage classes `ExportToFile` and `Volatile`.

Data Types: `char` | `string`

`GetFunction` — Name of get function
character vector | string scalar

Name of a `get` function that a data element calls in the generated code. Applies to storage class `GetSet`.

Data Types: `char` | `string`

`HeaderFile` — C header file
character vector | string scalar

File name for a C header file that contains declarations for global data read by data elements and external code. Applies to storage classes `ExportToFile`, `GetSet`, `ImportFromFile`, and `Volatile`.

Data Types: `char` | `string`

`Owner` — Owner of global data
character vector | string scalar

Name of the model that owns global data that is used by other models in the same model hierarchy. The code generated for the model that owns the data includes the global data definition. Applies to storage classes `ExportToFile` and `Volatile`.

Data Types: `char` | `string`

`PreserveDimensions` — Boolean flag indicating whether to preserve dimensions of multidimensional arrays
`True` | `False`

When model configuration parameter **Array layout** is set to `Row-major`, a flag that indicates whether to preserve dimensions of a data element represented in generated code as a multidimensional array. Applies to storage classes `ExportToFile`, `ImportFromFile`, `Localizable`, and `Volatile`.

Data Types: `logical`

`SetFunction` — Name of set function
character string | string scalar

Name of a `set` function that a data element calls in the generated code. Applies to storage class `GetSet`.

Data Types: `char` | `string`

`StructName` — Name of structure
character vector | string scalar

Name that the code generator uses to identify the structure for a data element in the generated code. Applies to storage classes `Bitfield` and `Struct`.

Data Types: `char` | `string`

`storageClassPropertyName` — Value of storage class property
depends on property definition

Storage class property defined in the Embedded Coder Dictionary. Values that you can specify vary depending on the storage class definition.

See Also

`coder.mapping.api.CodeMapping` | `coder.mapping.api.get` | `find` | `getDataDefault` | `getDataStore` | `setDataDefault`

Topics
“Configure Data Stores for C Code Generation”
“C Code Generation Configuration for Model Interface Elements”

Introduced in R2020b

setFunction

Set code mapping information for model function

Syntax

```
setFunction(myCodeMappingObj,function,Name,Value)
```

Description

`setFunction(myCodeMappingObj,function,Name,Value)` sets code mapping information for the specified model function. Use this function to set the function customization template, memory section, or function name for a model function. For single-tasking periodic functions and Simulink functions, you can use this function to set the argument specification, including argument names, type qualifiers, and argument order.

Examples

Configure Function Name for Model Initialize Function

In the model code mappings for model `myConfigModel`, configure the name of the generated C initialize function as `myInitFunction`.

```
cm = coder.mapping.api.get('myConfigModel');
setFunction(cm,'Initialize','FunctionName','myInitFunction');
```

Configure Memory Section for Periodic Single-Tasking Function

In the model code mappings for model `myInitFunction`, configure the memory section for the periodic single-tasking function as `None`.

```
cm = coder.mapping.api.get('myInitFunction');
setFunction(cm,'Periodic','MemorySection','None');
```

Configure Function Customization Template for Periodic Multitasking Function for Sample Time D2

In the model code mappings for model `myInitFunction`, configure the function customization template for the periodic multitasking function for sample time D2 as `FastFcn`.

```
cm = coder.mapping.api.get('myInitFunction');
setFunction(cm,'Periodic:D2','FunctionCustomizationTemplate','FastFcn');
```

Configure Argument Specification for Simulink Function

In the model code mappings for model `myInitFunction`, configure the argument specification for Simulink function `mySLFunc` as `y=(u1, const *u2)`.

```
cm = coder.mapping.api.get('myInitFunction');
setFunction(cm,'mySLFunc','Arguments','y=(u1, const *u2)');
```

Input Arguments

`myCodeMappingObj` — Code mapping object

`CodeMapping` object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

Example: `myCM`

`function` — Model function

`Initialize` | `Terminate` | `Periodic:slIdentifier` | `Partition:slIdentifier` | `PeriodicUpdate:slIdentifier` | `PartitionUpdate:slIdentifier` | `Reset:slIdentifier` | `ExportedFunction:slIdentifier` | `SimulinkFunction:slIdentifier`

Model function for which to set code mapping property value. Specify one of the values listed in this table.

Type of Model Function	Value
Exported function	`ExportedFunction:`*`slIdentifier`*, where *`slIdentifier`* is the name of the function-call Inport block in the model
Initialize function	`Initialize`
Partition function	`Partition:`*`slIdentifier`*, where *`slIdentifier`* is a partition name for an exported function or a function for the model that you explicitly partition in the Simulink Schedule Editor. For example, `P1`.
Partition update function (model configuration parameter **Single output/update function** is cleared)	`PartitionUpdate:`*`slIdentifier`*, is a partition name for an exported function or a function for the model that you explicitly partition in the Simulink Schedule Editor (for example, `P1`)
Periodic, multitasking function	`Periodic:`*`slIdentifier`*, where *`slIdentifier`* is an annotation that corresponds the sample time period associated with a function for a periodic partition of a multi-tasking model (for example, `D1`)
Periodic, multitasking update function (model configuration parameter **Single output/update function** is cleared)	`PeriodicUpdate:`*`slIdentifier`*, where *`slIdentifier`* is an annotation that corresponds the sample time period associated with a function for a periodic partition of a multi-tasking model (for example, `D1`)

Type of Model Function	Value
Periodic, single-tasking function	`Periodic`
Periodic, single-tasking update function a (model configuration parameter **Single output/update function** is cleared)	`PeriodicUpdate`
Reset function	`Reset:`*`slIdentifier`*, where *`slIdentifier`* is the name of the reset function in the model
Simulink function	`SimulinkFunction:`*`slIdentifier`*, where *`slIdentifier`* is the name of the Simulink function in the model
Terminate function	`Terminate`

Model function for which to return a code mapping property value. Specify one of the values listed in this table. If model configuration parameter **Single output/update function** is cleared, you can specify the update version of a partition, periodic multi-tasking, or periodic singletasking function.

Type of Model Function	Value
Exported function	`ExportedFunction:`*`slIdentifier`*, where *`slIdentifier`* is the name of the function-call Inport block in the model
Initialize function	`Initialize`
Partition function	`Partition:`*`slIdentifier`*, where *`slIdentifier`* is a partition that was created explicitly from a block in the model and shown in the Simulink Schedule Editor (for example, `P1`)
Partition update function	`PartitionUpdate:`*`slIdentifier`*, is a partition that was created explicitly from a block in the model and shown in the Simulink Schedule Editor (for example, `P1`)
Periodic multitasking function	`Periodic:`*`slIdentifier`*, where *`slIdentifier`* is an annotation that corresponds to the sample time period for a periodic or continuous rate of a multi-tasking model (for example, `D1`)
Periodic multitasking update function	`PeriodicUpdate:`*`slIdentifier`*, where *`slIdentifier`* is an annotation that corresponds to the sample time period for a periodic or continuous rate of a multi-tasking model (for example, `D1`)
Periodic single-tasking function	`Periodic`
Periodic single-tasking update function	`PeriodicUpdate`
Reset function	`Reset:`*`slIdentifier`*, where *`slIdentifier`* is the name of the reset function in the model
Simulink function	`SimulinkFunction:`*`slIdentifier`*, where *`slIdentifier`* is the name of the Simulink function in the model
Terminate function	`Terminate`

For information about model partitioning, see “Create Partitions”.

Example: `'Periodic:D1'`

Name-Value Pair Arguments

Specify comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments as `Name1,Value1,…,NameN,ValueN`. The order of the name and value pair arguments does not matter.

Example: 'FunctionCustomizationTemplate' 'exFastFunction'

`FunctionCustomizationTemplate` — Name of function customization template
character vector | string scalar

Name of a function customization template defined in the Embedded Coder Dictionary associated with the model, specified as a character vector or string scalar. If you set the default function customization template for a category of functions to `Default`, you can specify a memory section for the functions category.

Data Types: `char` | `string`

`MemorySection` — Name of memory section
character vector | string scalar

Name of a memory section that is defined in the Embedded Coder Dictionary associated with the model, specified as a character vector or string scalar.

Data Types: `char` | `string`

`FunctionName` — Name of function
character vector | string scalar

Name for the entry-point function in the generated C code, specified as a character vector or string scalar.

Data Types: `char` | `string`

`Arguments` — Argument specification
character vector | string scalar

Argument specification for the entry-point function in the generated C code, specified as a character vector or string scalar. The specification is a function prototype that shows argument names, type qualifiers, and argument order (for example, `y=(u1, const *u2)` .

Data Types: `char` | `string`

See Also
`coder.mapping.api.CodeMapping` | `coder.mapping.api.get` | `getFunction` | `getFunctionDefault` | `setFunctionDefault`

Topics
“Configure Names for Individual C Entry-Point Functions”
“Configure Name and Arguments for Individual Step Functions”
“Configure Default C Code Generation for Categories of Data Elements and Functions”
“C Code Generation Configuration for Model Interface Elements”

Introduced in R2020b

setFunctionDefault

Set default function customization template and memory section for model functions category

Syntax

`setFunctionDefault(myCodeMappingObj,category,Name,Value)`

Description

`setFunctionDefault(myCodeMappingObj,category,Name,Value)` sets the default function customization template and memory section for the specified category of model entry-point functions.

Examples

Configure Default Memory Section for Model Execution Functions

For model `myConfigModel`, configure the code generator to use memory section `functionFastMem` for generating code for model execution functions in the model code mappings.

```
cm = coder.mapping.api.get('myConfigModel');
setFunctionDefault(cm,'Execution','MemorySection','FunctionFastMem');
```

Input Arguments

`myCodeMappingObj` — Code mapping object
`CodeMapping` object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

Example: `myCM`

`category` — Model function category
`InitializeTerminate` | `Execution` | `SharedUtility`

Category of model entry-point functions for which to set the function customization template and memory section.

Example: `'Execution'`

Name-Value Pair Arguments

Specify comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments as `Name1,Value1,…,NameN,ValueN`. The order of the name and value pair arguments does not matter.

Example: 'FunctionCustomizationTemplate' 'exFastFunction'

`FunctionCustomizationTemplate` — Name of function customization template
character vector | string scalar

Name of a function customization template defined in the Embedded Coder Dictionary associated with the model. If you set the default function customization template for a category of functions to `Default`, you can specify a memory section for the category of functions.

Data Types: `char` | `string`

`MemorySection` — Name of memory section
character vector | string scalar

Name of a memory section that is defined in the Embedded Coder Dictionary associated with the model.

Data Types: `char` | `string`

See Also

`coder.mapping.api.CodeMapping` | `coder.mapping.api.get` | `getFunction` | `getFunctionDefault` | `setFunction`

Topics

"Configure Default C Code Generation for Categories of Data Elements and Functions"
"C Code Generation Configuration for Model Interface Elements"

Introduced in R2020b

setInport

Configure root-level inports for code generation

Syntax

```
setInport(myCodeMappingObj,inport,Name,Value)
```

Description

`setInport(myCodeMappingObj,inport,Name,Value)` configures specified root-level Inport blocks for code generation. Use this function to map specified root-level inports to the storage class and storage class property settings that the code generator uses to produce C code for the inports.

Examples

Configure Storage Class for Root-Level Inports

In the model code mappings for model `myConfigModel`, set the storage class for root-level Inport block `In1` to `ExportedGlobal`.

```
cm = coder.mapping.api.get('myConfigModel');
setInport(cm,'In1','StorageClass','ExportedGlobal');
```

Configure Storage Class for Root-Level Inports in Model to `Model default`

In the model code mappings for model `myConfigModel`, set the storage class for root-level inports throughout the model to `Model default`.

```
cm = coder.mapping.api.get('myConfigModel');
inBlockHandles  = find(cm,'Inports')
setInport(cm,inBlockHandles,'StorageClass','Model default');
```

Configure Storage Class and Storage Class Properties for Root Inport Block

In the model code mappings for model `myConfigModel`, set the storage class for root-level inport `In1` to `InportFromFile`. Set the code identifier to `input1` and the header file to `exInDataMem.h`.

```
cm = coder.mapping.api.get('myConfigModel');
setInport(cm,'In1','StorageClass','ImportFromFile',...
   'Identifier','input1','HeaderFile','exInDataMem.h');
```

Input Arguments

`myCodeMappingObj` — Code mapping object
`CodeMapping` object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

Example: `myCM`

`inport` — Name, path, or handle of root-level inport
character vector | string scalar | block handle | cell array of character vectors | cell array of string scalars | cell array of handles

Name, path, or handle of root-level inport to configure. To specify multiple inports, use a cell array.

Example: `'In1'`

Data Types: `char` | `string` | `cell`

Name-Value Pair Arguments

Specify comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments as `Name1,Value1,…,NameN,ValueN`. The order of the name and value pair arguments does not matter.

Example: 'StorageClass' 'ExportedGlobal'

`StorageClass` — Name of storage class
`Auto` | `Bitfield` | `Dictionary default` | `ExportedGlobal` | `ExportToFile` | `GetSet` | `ImportedExtern` | `ImportedExternPointer` | `ImportFromFile` | `Localizable` | `Model default` | `Struct` | `Volatile` | storage class name

Storage class to set for the specified root Inport block. The name of a predefined storage class or storage class that is defined in the Embedded Coder Dictionary associated with the model. See “Configure Root-Level Inport Blocks for C Code Generation”.

`Identifier` — Name of variable
character vector | string scalar

Name for the variable that represents the inport in the generated code.

Data Types: `char` | `string`

`DefinitionFile` — C source file
character vector | string scalar

File name for a C source file that contains definitions for global data read by data elements and external code. Applies to storage classes `ExportToFile` and `Volatile`.

Data Types: `char` | `string`

`GetFunction` — Name of get function
character vector | string scalar

Name of a `get` function that a data element calls in the generated code. Applies to storage class `GetSet`.

Data Types: `char` | `string`

`HeaderFile` — C header file
character vector | string scalar

File name for a C header file that contains declarations for global data read by data elements and external code. Applies to storage classes `ExportToFile`, `GetSet`, `ImportFromFile`, and `Volatile`.

Data Types: `char` | `string`

`Owner` — Owner of global data
character vector | string scalar

Name of the model that owns global data, which is used by other models in the same model hierarchy. The code generated for the model that owns the data includes the global data definition. Applies to storage classes `ExportToFile` and `Volatile`.

Data Types: `char` | `string`

`PreserveDimensions` — Boolean flag indicating whether to preserve dimensions of multidimensional arrays
`True` | `False`

When model configuration parameter **Array layout** is set to `Row-major`, a flag that indicates whether to preserve dimensions of a data element represented in generated code as a multidimensional array. Applies to storage classes `ExportToFile`, `ImportFromFile`, `Localizable`, and `Volatile`.

Data Types: `logical`

`SetFunction` — Name of set function
character string | string scalar

Name of a `set` function that a data element calls in the generated code. Applies to storage class `GetSet`.

Data Types: `char` | `string`

`StructName` — Name of structure
character vector | string scalar

Name that the code generator uses to identify the structure for a data element in the generated code. Applies to storage classes `Bitfield` and `Struct`.

Data Types: `char` | `string`

`storageClassPropertyName` — Value of storage class property
depends on property definition

Storage class property defined in the Embedded Coder Dictionary. Values that you can specify vary depending on the storage class definition.

See Also

`coder.mapping.api.CodeMapping` | `coder.mapping.api.get` | `find` | `getDataDefault` | `getInport` | `setDataDefault`

Topics

“Configure Root-Level Inport Blocks for C Code Generation”
“C Code Generation Configuration for Model Interface Elements”

Introduced in R2020b

setModelParameter

Configure model parameter for code generation

Syntax

`setModelParameter(myCodeMappingObj,modelParameter,Name,Value)`

Description

`setModelParameter(myCodeMappingObj,modelParameter,Name,Value)` configures the specified model parameter or model parameter argument for code generation. Use this function to map the specified model parameter or model parameter argument to the storage class and storage class property settings that the code generator uses to produce C code for the parameter or parameter argument.

Examples

Configure Storage Class for Model Parameter

In the model code mappings for model `myConfigModel`, set the storage class for model parameter `K1` to `ExportedGlobal`.

```
cm = coder.mapping.api.get('myConfigModel');
setModelParameter(cm,'K1','StorageClass','ExportedGlobal');
```

Configure Storage Class for Model Parameters in Model to `Model default`

In the model code mappings for model `myConfigModel`, set the storage class for model parameters throughout the model to `Model default`.

```
cm = coder.mapping.api.get('myConfigModel');
paramHandles = find(cm,'ModelParameters')
setModelParameters(cm,paramHandles,'StorageClass','Model default');
```

Configure Storage Class and Code Identifier for Model Parameters

In the model code mappings for model `myConfigModel`, set the storage class for model parameters `Table1` and `Table2` to `ExportedGlobal`. Set the identifier for the variables representing the parameters in the generated code to `mp_Table1` and `mp_Table2`.

```
cm = coder.mapping.api.get('myConfigModel');
setModelParameter(cm,'Table1','StorageClass','ExportedGlobal',...
   'Identifier','mp_Table1');
setModelParameter(cm,'Table2','StorageClass','ExportedGlobal',...
   'Identifier','mp_Table2');
```

Configure Storage Class and Code Identifiers for Model Parameter Arguments

In the model code mappings for model `myConfigModel`, set the storage class for model parameter arguments `LOWER` and `UPPER` to `ExportedGlobal`. Set the identifiers for the variables representing the parameter arguments in the generated code to `arg_LOWER` and `arg_UPPER`.

```
cm = coder.mapping.api.get('myConfigModel');
setModelParameter(cm,'LOWER','StorageClass','ExportedGlobal',...
   'Identifier','arg_LOWER');
setModelParameter(cm,'UPPER','StorageClass','ExportedGlobal',...
   'Identifier','arg_UPPER');
```

Input Arguments

`myCodeMappingObj` — Code mapping object
`CodeMapping` object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

Example: `myCM`

`modelParameter` — Name of model parameter or model parameter argument
character vector | string | cell array of character vectors | cell array of string scalars

Name of the model workspace parameter or model parameter argument to configure.

Example: `'Table1'`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments as `Name1,Value1,…,NameN,ValueN`. The order of the name and value pair arguments does not matter.

Example: 'StorageClass' 'ExportedGlobal'

`StorageClass` — Name of storage class
`Auto` | `Bitfield` | `CompilerFlag` | `Const` | `ConstVolatile` | `Define` | `Dictionary default` | `ExportedGlobal` | `ExportToFile` | `FileScope` | `GetSet` | `ImportedDefine` | `ImportedExtern` | `ImportedExternPointer` | `ImportFromFile` | `Localizable` | `Model default` | `Struct` | `Volatile` | storage class name

Storage class to set for the specified model parameter or model parameter argument. The name of a predefined storage class or storage class that is defined in the Embedded Coder Dictionary associated with the model. See “Configure Parameters for C Code Generation”.

`Identifier` — Name of variable
character vector | string scalar

Name for the variable that represents the model parameter or model parameter argument in the generated code.

Data Types: `char` | `string`

DefinitionFile — C source file
character vector | string scalar

File name for a C source file that contains definitions for global data read by data elements and external code. Applies to storage classes ExportToFile and Volatile.

Data Types: char | string

GetFunction — Name of get function
character vector | string scalar

Name of a get function that a data element calls in the generated code. Applies to storage class GetSet.

Data Types: char | string

HeaderFile — C header file
character vector | string scalar

File name for a C header file that contains declarations for global data read by data elements and external code. Applies to storage classes ExportToFile, GetSet, ImportFromFile, and Volatile.

Data Types: char | string

Owner — Owner of global data
character vector | string scalar

Name of the model that owns global data used by other models in the same model hierarchy. The code generated for the model that owns the data includes the global data definition. Applies to storage classes ExportToFile and Volatile.

Data Types: char | string

PreserveDimensions — Boolean flag indicating whether to preserve dimensions of multidimensional arrays
True | False

When model configuration parameter **Array layout** is set to Row-major, a flag that indicates whether to preserve dimensions of a data element that is represented in generated code as a multidimensional array. Applies to storage classes ExportToFile, ImportFromFile, Localizable, and Volatile.

Data Types: logical

SetFunction — Name of set function
character string | string scalar

Name of a set function that a data element calls in the generated code. Applies to storage class GetSet.

Data Types: char | string

StructName — Name of structure
character vector | string scalar

Name that the code generator uses to identify the structure for a data element in the generated code. Applies to storage classes Bitfield and Struct.

Data Types: `char` | `string`

`storageClassPropertyName` — Value of storage class property
depends on property definition

Storage class property defined in the Embedded Coder Dictionary. Values that you can specify vary depending on the storage class definition.

See Also

`coder.mapping.api.CodeMapping` | `coder.mapping.api.get` | `find` | `getDataDefault` | `getModelParameter` | `setDataDefault`

Topics
"Configure Parameters for C Code Generation"
"C Code Generation Configuration for Model Interface Elements"

Introduced in R2020b

setOutport

Configure root-level outport for code generation

Syntax

`setOutport(myCodeMappingObj,outport,Name,Value)`

Description

`setOutport(myCodeMappingObj,outport,Name,Value)` configures specified root-level Outport blocks for code generation. Use this function to map specified root-level outports to the storage class and storage class property settings that the code generator uses to produce C code for the outports.

Examples

Configure Storage Class for Root-Level Outports

In the model code mappings for model `myConfigModel`, set the storage class for root-level outport `Out1` to `ExportedGlobal`.

```
cm = coder.mapping.api.get('myConfigModel');
setOutport(cm,'Out1','StorageClass','ExportedGlobal');
```

Configure Storage Class for Root-Level Outports in Model to `Model default`

In the model code mappings for model `myConfigModel`, set the storage class for root Outport blocks throughout the model to `Model default`.

```
cm = coder.mapping.api.get('myConfigModel');
outBlockHandles = find(cm,'Outports')
setOutport(cm,outBlockHandles,'StorageClass','Model default');
```

Configure Storage Class and Storage Class Properties for Outport

In the model code mappings for model `myConfigModel`, set the storage class for root-level outport `Out1` to `ExportToFile`. Set the code identifier to `output1`, the definition file to `exOutSys.c`, and the header file to `exOutSys.h`.

```
cm = coder.mapping.api.get('myConfigModel');
setOutport(cm,'Out1','StorageClass','ExportToFile',...
```

```
'Identifier','output1','DefinitionFile','exOutSys.c',...
'HeaderFile','exOutSys.h');
```

Input Arguments

myCodeMappingObj — Code mapping object
CodeMapping object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

Example: myCM

outport — Name, path, or handle of root-level outport
character vector | string scalar | block handle | cell array of character vectors | cell array of string scalars | cell array of handles

Name, path, or handle of root-level outport to configure. To specify multiple outports, use a cell array.

Example: 'Out1'

Data Types: char | string | cell

Name-Value Pair Arguments

Specify comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments as `Name1,Value1,…,NameN,ValueN`. The order of the name and value pair arguments does not matter.

Example: 'StorageClass' 'ExportedGlobal'

StorageClass — Name of storage class
Auto | Bitfield | Dictionary default | ExportedGlobal | ExportToFile | GetSet | ImportedExtern | ImportedExternPointer | ImportFromFile | Localizable | Model default | Struct | Volatile | storage class name

Storage class to set for the specified root Outport block. The name of a predefined storage class or storage class that is defined in the Embedded Coder Dictionary associated with the model. See "Configure Root-Level Outport Blocks for C Code Generation".

Identifier — Name of variable
character vector | string scalar

Name for the variable that represents the outport in the generated code.

Data Types: char | string

DefinitionFile — C source file
character vector | string scalar

File name for a C source file that contains definitions for global data read by data elements and external code. Applies to storage classes `ExportToFile` and `Volatile`.

Data Types: char | string

GetFunction — Name of get function
character vector | string scalar

Name of a get function that a data element calls in the generated code. Applies to storage class GetSet.

Data Types: char | string

HeaderFile — C header file
character vector | string scalar

File name for a C header file that contains declarations for global data read by data elements and external code. Applies to storage classes ExportToFile, GetSet, ImportFromFile, and Volatile.

Data Types: char | string

Owner — Owner of global data
character vector | string scalar

Name of the model that owns global data used by other models in the same model hierarchy. The code generated for the model that owns the data includes the global data definition. Applies to storage classes ExportToFile and Volatile.

Data Types: char | string

PreserveDimensions — Boolean flag indicating whether to preserve dimensions of multidimensional arrays
True | False

When model configuration parameter **Array layout** is set to Row-major, a flag that indicates whether to preserve dimensions of a data element represented in generated code as a multidimensional array. Applies to storage classes ExportToFile, ImportFromFile, Localizable, and Volatile.

Data Types: logical

SetFunction — Name of set function
character string | string scalar

Name of a set function that a data element calls in the generated code. Applies to storage class GetSet.

Data Types: char | string

StructName — Name of structure
character vector | string scalar

Name that the code generator uses to identify the structure for a data element in the generated code. Applies to storage classes Bitfield and Struct.

Data Types: char | string

storageClassPropertyName — Value of storage class property
depends on property definition

Storage class property defined in the Embedded Coder Dictionary. Values that you can specify vary depending on the storage class definition.

See Also

`coder.mapping.api.CodeMapping` | `coder.mapping.api.get` | `find` | `getDataDefault` | `getOutport` | `setDataDefault`

Topics

"Configure Root-Level Outport Blocks for C Code Generation"
"C Code Generation Configuration for Model Interface Elements"

Introduced in R2020b

setSignal

Configure block signal data for code generation

Syntax

```
setSignal(myCodeMappingObj,portHandle,Name,Value)
```

Description

`setSignal(myCodeMappingObj,portHandle,Name,Value)` configures signals specified by block output ports for code generation. Use this function to map specified block output ports to the storage class and storage class property settings that the code generator uses to produce C code for the corresponding signal data.

This function does not apply to signals that originate from root-level Inport blocks. For signals that originate from root-level Inport blocks, see `setInport`.

Examples

Configure Storage Class for Block Output Signals

In the model code mappings for model `myConfigModel`, set the storage class for output signals of lookup table blocks `Table1D` and `Table2D` to `ExportedGlobal`. After creating the object `cm` by calling function `coder.mapping.api.get`, get handles to the output ports for lookup table blocks. Set the storage class for the output signals by calling `setSignal`.

```
cm = coder.mapping.api.get('myConfigModel');
lut1D_ports = get_param('myConfigModel/Table1D','PortHandles');
lut2D_ports = get_param('myConfigModel/Table2D','PortHandles');
lut1D_outPort = lut1D_ports.Outport;
lut2D_outPort = lut2D_ports.Outport;
setSignal(cm,[lut1D_outPort,lut2D_outPort],'StorageClass','ExportedGlobal');
```

Configure Storage Class for Signal Data in Model Code Mappings to `Model default`

In the model code mappings for model `myConfigModel`, set the storage class for block output signals to `Model default`. After creating the object `cm` by calling function `coder.mapping.api.get`, get the port handles of the signal data in the code mappings. Set the storage class for the signals by calling `setSignal`.

```
cm = coder.mapping.api.get('myConfigModel');
portHandles = find(cm,'Signals')
setSignal(cm,portHandles,'StorageClass','Model default');
```

Configure Code Identifiers for Block Output Signals

In the model code mappings for model `myConfigModel`, set the code identifiers for output signals of lookup table blocks `Table1D` and `Table2D` to `dout_Table1D` and `dout_Table2D`. After creating the

object cm by calling function coder.mapping.api.get, get handles to the output ports for lookup table blocks. Set the code identifiers for the output signals by calling setSignal.

```
cm = coder.mapping.api.get('myConfigModel');
lut1D_ports = get_param('myConfigModel/Table1D','PortHandles');
lut2D_ports = get_param('myConfigModel/Table2D','PortHandles');
lut1D_outPort = lut1D_ports.Outport;
lut2D_outPort = lut2D_ports.Outport;
setSignal(cm,lut1D_outPort,'Identifier','dout_Table1D');
setSignal(cm,lut2D_outPort,'Identifier','dout_Table2D');
```

Input Arguments

myCodeMappingObj — Code mapping object

CodeMapping object

Code mapping object (model code mappings) returned by a call to function coder.mapping.api.get.

Example: myCM

portHandle — Output port handle of signal source block

port handle | array of port handles

Signal to add to the code mappings, specified as a handle of an output port of the signal's source block. To specify multiple port handles, use an array.

Example: portHandle

Data Types: port_handle | array

Name-Value Pair Arguments

Specify comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments as Name1,Value1,...,NameN,ValueN. The order of the name and value pair arguments does not matter.

Example: 'StorageClass' 'ExportedGlobal'

StorageClass — Name of storage class

Auto | Bitfield | Dictionary default | ExportedGlobal | ExportToFile | FileScope | GetSet | ImportedExtern | ImportedExternPointer | ImportFromFile | Localizable | Model default | Struct | Volatile | storage class name

Storage class to set for the specified signals. The name of a predefined storage class or storage class that is defined in the Embedded Coder Dictionary associated with the model. See “Configure Signal Data for C Code Generation”.

Identifier — Name of variable

character vector | string scalar

Name for the variable that represents the signal data in the generated code.

Data Types: char | string

DefinitionFile — C source file
character vector | string scalar

File name for a C source file that contains definitions for global data read by data elements and external code. Applies to storage classes ExportToFile and Volatile.

Data Types: char | string

GetFunction — Name of get function
character vector | string scalar

Name of a get function that a data element calls in the generated code. Applies to storage class GetSet.

Data Types: char | string

HeaderFile — C header file
character vector | string scalar

File name for a C header file that contains declarations for global data read by data elements and external code. Applies to storage classes ExportToFile, GetSet, ImportFromFile, and Volatile.

Data Types: char | string

Owner — Owner of global data
character vector | string scalar

Name of the model that owns global data used by other models in the same model hierarchy. The code generated for the model that owns the data includes the global data definition. Applies to storage classes ExportToFile and Volatile.

Data Types: char | string

PreserveDimensions — Boolean flag indicating whether to preserve dimensions of multidimensional arrays
True | False

When model configuration parameter **Array layout** is set to Row-major, a flag that indicates whether to preserve dimensions of a data element represented in generated code as a multidimensional array. Applies to storage classes ExportToFile, ImportFromFile, Localizable, and Volatile.

Data Types: logical

SetFunction — Name of set function
character string | string scalar

Name of a set function that a data element calls in the generated code. Applies to storage class GetSet.

Data Types: char | string

StructName — Name of structure
character vector | string scalar

Name that the code generator uses to identify the structure for a data element in the generated code. Applies to storage classes Bitfield and Struct.

`storageClassPropertyName` — Value of storage class property
depends on property definition

Storage class property defined in the Embedded Coder Dictionary. Values that you can specify vary depending on the storage class definition.

Data Types: `char` | `string`

See Also

`addSignal` | `coder.mapping.api.CodeMapping` | `coder.mapping.api.get` | `find` | `getDataDefault` | `getSignal` | `removeSignal` | `setDataDefault`

Topics

"Configure Signal Data for C Code Generation"
"C Code Generation Configuration for Model Interface Elements"

Introduced in R2020b

setState

Configure block states for code generation

Syntax

```
setState(myCodeMappingObj,block,Name,Value)
```

Description

`setState(myCodeMappingObj,block,Name,Value)` configures specified block states for code generation. Use this function to map specified block states to the storage class and storage class property settings that the code generator uses to produce C code for the states.

Examples

Configure Storage Class for Block State

In the model code mappings for model `myConfigModel`, set the storage class for the state `X` of Unit Delay block `Delay` to `ExportedGlobal`.

```
cm = coder.mapping.api.get('myConfigModel');
setState(cm,'myConfigModel/Delay','StorageClass','ExportedGlobal');
```

Configure Storage Class for Block States in Model to `Model default`

In the model code mappings for model `myConfigModel`, configure the storage class for block states throughout the model to `Model default`.

```
cm = coder.mapping.api.get('myConfigModel');
blockHandles = find(cm,'States')
setState(cm,blockHandles,'StorageClass','Model default');
```

Configure Code Identifier for Block State

In the model code mappings for model `myConfigModel`, configure the code identifier for the state `X` of Unit Delay block `Delay` to `dstate_X`.

```
cm = coder.mapping.api.get('myConfigModel');
setState(cm,blockHandle,'Identifier','dstate_X');
```

Input Arguments

`myCodeMappingObj` — Code mapping object
`CodeMapping` object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

Example: `myCM`

`block` — Path or handle of block
character vector | string scalar | block handle | cell array of character vectors | cell array of string scalars | cell array of block handles

Path or handle of the block containing the state to configure. To specify multiple block states, use a cell array.

Example: `blockHandle`

Data Types: `char` | `string` | `block_handle` | `cell`

Name-Value Pair Arguments

Specify comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments as `Name1,Value1,…,NameN,ValueN`. The order of the name and value pair arguments does not matter.

Example: 'StorageClass' 'ExportedGlobal'

`StorageClass` — Name of storage class
`Auto` | `Bitfield` | `Dictionary default` | `ExportedGlobal` | `ExportToFile` | `FileScope` | `GetSet` | `ImportedExtern` | `ImportedExternPointer` | `ImportFromFile` | `Localizable` | `Model default` | `Struct` | `Volatile` | storage class name

Storage class to set for the specified block state. The name of a predefined storage class or storage class that is defined in the Embedded Coder Dictionary associated with the model. See “Configure Block States for C Code Generation”.

`Identifier` — Name of variable
character vector | string scalar

Name for the variable that represents the block state in the generated code.

Data Types: `char` | `string`

`DefinitionFile` — C source file
character vector | string scalar

File name for a C source file that contains definitions for global data read by data elements and external code. Applies to storage classes `ExportToFile` and `Volatile`.

Data Types: `char` | `string`

`GetFunction` — Name of get function
character vector | string scalar

Name of a `get` function that a data element calls in the generated code. Applies to storage class `GetSet`.

Data Types: `char` | `string`

HeaderFile — C header file
character vector | string scalar

File name for a C header file that contains declarations for global data read by data elements and external code. Applies to storage classes ExportToFile, GetSet, ImportFromFile, and Volatile.

Data Types: char | string

Owner — Owner of global data
character vector | string scalar

Name of the model that owns global data used by other models in the same model hierarchy. The code generated for the model that owns the data includes the global data definition. Applies to storage classes ExportToFile and Volatile.

Data Types: char | string

PreserveDimensions — Boolean flag indicating whether to preserve dimensions of multidimensional arrays
True | False

When model configuration parameter **Array layout** is set to Row-major, a flag that indicates whether to preserve dimensions of a data element represented in generated code as a multidimensional array. Applies to storage classes ExportToFile, ImportFromFile, Localizable, and Volatile.

Data Types: logical

SetFunction — Name of set function
character string | string scalar

Name of a set function that a data element calls in the generated code. Applies to storage class GetSet.

Data Types: char | string

StructName — Name of structure
character vector | string scalar

Name that the code generator uses to identify the structure for a data element in the generated code. Applies to storage classes Bitfield and Struct.

Data Types: char | string

storageClassPropertyName — Value of storage class property
depends on property definition

Storage class property defined in the Embedded Coder Dictionary. Values that you can specify vary depending on the storage class definition.

See Also

coder.mapping.api.CodeMapping | coder.mapping.api.get | find | getDataDefault | getState | setDataDefault

Topics
"Configure Block States for C Code Generation"

"C Code Generation Configuration for Model Interface Elements"

Introduced in R2020b

coder.mapping.utils.create

Create code mappings object for configuring data and function interface for C and C++ code generation

Syntax

```
myCodeMappingObj = coder.mapping.utils.create(model)
myCodeMappingObj = coder.mapping.utils.create(dictionary)
myCodeMappingObj = coder.mapping.utils.create(model,configObj)
```

Description

`myCodeMappingObj = coder.mapping.utils.create(model)` creates code mappings environment for the specified model and returns the mappings as object `myCodeMappingObj`. Code mappings associate model data elements and functions with configurations for C or C++ code generation. If code mappings exist for the specified model, the function returns those code mappings.

`myCodeMappingObj = coder.mapping.utils.create(dictionary)` creates C code mappings environment for the specified data dictionary and returns the mappings as object `myCodeMappingObj`. Code mappings associate data elements and functions with configurations for C or C++ code generation. If code mappings exist for the specified data dictionary, the function returns those code mappings.

`myCodeMappingObj = coder.mapping.utils.create(model,configObj)` imports default memory section and shared utility naming rule configurations from configuration set `configObj` while creating C code mappings for the specified model. See “Migration of Memory Section and Shared Utility Settings from Configuration Parameters to Code Mappings”.

Examples

Create Code Mappings for Model

For model `myConfigModel`, create C code mappings by calling `coder.mapping.utils.create`. Then, get the code mappings with a call to `coder.mapping.api.get`.

```
myCodeMappingObj = coder.mapping.utils.create('myConfigModel');
myCodeMappingObj = coder.mapping.api.get('myConfigModel');
```

Import Configurations from Configuration Set While Creating Code Mappings for Model

For model `myConfigModel`, import memory section and shared utility naming rule configurations from configuration set `myConfigSet` while creating C code mappings.

```
myCodeMappingObj = coder.mapping.utils.create('myConfigModel','myLegacyConfigSet');
```

After calling this function, call `coder.mapping.api.get` to get the code mapping object.

Input Arguments

`model` — Name of model
handle | character vector | string scalar

Model file for which to create and return a code mappings object, specified as a handle or a character vector or string scalar representing the model name. The model must be loaded (for example, by using `load_system`) or open. Omit the `.slx` file extension.

Example: `'myConfigModel'`

Data Types: `char` | `string` | `model_handle`

`dictionary` — Name of data dictionary
character vector | string scalar

Data dictionary for which to return code mappings object, specified as a character vector or string scalar representing the dictionary name.

Example: `'exCodeDefs.sldd'`

Data Types: `char` | `string`

`configObj` — Configuration object
handle | character vector | string scalar

Model configuration object from which to import memory section and shared utility naming rule configurations while creating code mappings for `model`, specified as a handle, character vector or string scalar. Specify a configuration set object to preserve memory section definitions or shared utility naming rules applied to a model in a version of Embedded Coder prior to R2018a.

Example: `'my_legacyConfigset'`

Output Arguments

`myCodeMappingObj` — Code mapping object
`CodeMapping` object | `CodeMappingCPP` object | `CoderDictionary` object

The model or dictionary code mappings, returned as a `CodeMapping` object, a `CodeMappingCPP` object, or a `CoderDictionary` object.

Output	Input Object
`coder.mapping.api.CodeMapping`	Simulink model configured for C code generation
`coder.mapping.api.CodeMappingCPP`	Simulink model configured for C++ code generation
`coder.mapping.api.CoderDictionary`	Simulink data dictionary

See Also
`coder.mapping.api.CodeMapping` | `coder.mapping.api.CodeMappingCPP` | `coder.mapping.api.CoderDictionary` | `coder.mapping.api.get`

Topics
"C Code Generation Configuration for Model Interface Elements"
"Programmatically Configure C++ Interface"

Introduced in R2020b

coder.mapping.api.CoderDictionary

Query and set the code settings of dictionary defaults in an Embedded Coder dictionary within a Simulink data dictionary

Description

A coder dictionary code mappings object and its related functions enable you to configure C code generation settings for dictionary defaults in an Embedded Coder dictionary within a Simulink data dictionary. For model data categories, code mappings associate data categories with configurations that consist of a storage class and storage class properties. For functions, code mappings associate function categories with configurations that consist of a function customization template. Reduce the effort of preparing a model for code generation by specifying default configurations for categories of data elements and functions across a model.

Creation

Syntax

```
myCoderDictionaryObj = coder.mapping.api.get(dictionary)
myCoderDictionaryObj = coder.mapping.utils.create(dictionary)
```

Description

`myCoderDictionaryObj = coder.mapping.api.get(dictionary)` returns the active code mappings for the specified dictionary as object `myCoderDictionaryObj`. Code mappings associate data elements and functions in the data dictionary with configurations for code generation.

If code mappings do not exist, Simulink returns an error. You can create a code mappings object with a call to `coder.mapping.utils.create`.

`myCoderDictionaryObj = coder.mapping.utils.create(dictionary)` creates a code mappings environment for the specified data dictionary and returns the mappings as object `myCoderDictionaryObj`. Code mappings associate data elements and functions with configurations for C or C++ code generation. If code mappings exist for the specified data dictionary, the function returns those code mappings.

Input Arguments

`dictionary` — Name of data dictionary
character vector | string scalar

Data dictionary for which to return code mappings object, specified as a character vector or string scalar representing the dictionary name.

Example: `'exCodeDefs.sldd'`

Data Types: `char` | `string`

Object Functions

setDataDefault	Set default code settings for data category
getDataDefault	Get default code settings for data category
setFunctionDefault	Set default function customization template and memory section for model functions category
getFunctionDefault	Get default function customization template or memory section for model functions category

Examples

Create Environment to Configure Code Mappings for Data Dictionary

For the data dictionary `exCodeDefs.sldd`, create the environment for configuring the data and functions for code generation. After calling this function, use calls to other functions listed under Object Functions to configure aspects of code generation for the interface elements.

```
coder.mapping.utils.create('exCodeDefs.sldd');
```

See Also

`coder.mapping.api.CodeMapping` | `coder.mapping.api.CodeMappingCPP` | `coder.mapping.utils.create`

Topics

"C Code Generation Configuration for Model Interface Elements"
"Programmatically Configure C++ Interface"

Introduced in R2021a

setDataDefault

Set default code settings for data category

Syntax

`setDataDefault(myCoderDictionaryObj,category,Name,Value)`

Description

`setDataDefault(myCoderDictionaryObj,category,Name,Value)` sets the default storage class and storage class property values in the code mappings for the specified category of model data.

Examples

Configure default code settings for a data category in a data dictionary

Use the `coder.mapping.api.get` function to access the `CoderDictionary` object associated with the data dictionary.

```
cm = coder.mapping.api.get('codeDefinitions.sldd');
```

To see the storage class of root-level inports for the dictionary, use the `getDataDefault` function.

```
value = getDataDefault(cm, 'Inports', 'StorageClass')
```

```
value =

    'Default'
```

The dictionary uses the default storage class for inports.

To configure the storage class, use the `setDataDefault` function.

```
setDataDefault(cm, 'Inports', 'StorageClass', 'ExportedGlobal')
```

To verify that the storage class of inports is now set to `ExportedGlobal`, use the `getDataDefault` function.

```
value = getDataDefault(cm, 'Inports', 'StorageClass')
```

```
value =

    'ExportedGlobal'
```

Input Arguments

`myCoderDictionaryObj` — Coder dictionary object
`CoderDictionary` object

Coder dictionary object returned by a call to function `coder.mapping.api.get`.

category — Model data element category
Constants | ExternalParameterObjects | GlobalDataStores | Inports | InternalData | ModelParameters | ModelParameterArguments | Outports | SharedLocalDataStores

Category of data elements to return a property value for.

Example: 'Inports'

Name-Value Pair Arguments

Specify comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments as Name1,Value1,…,NameN,ValueN. The order of the name and value pair arguments does not matter.

Example: 'StorageClass','ExportedGlobal'

StorageClass — Name of storage class
Auto | Bitfield | CompileFlag | Const | ConstVolatile | Define | Dictionary default | ExportedGlobal | ExportToFile | FileScope | GetSet | ImportedDefine | ImportedExtern | ImportedExternPointer | ImportFromFile | Localizable | Model default | Struct | Volatile | storage class name

Storage class to set for the specified data element category. The name of a predefined storage class or storage class that is defined in the Embedded Coder Dictionary associated with the model. Values that you can specify vary depending on the category that you specify. See “Choose Storage Class for Controlling Data Representation in Generated Code”.

Example: 'StorageClass','ImportedExtern'

DefinitionFile — C source file
character vector | string scalar

File name for a C source file that contains definitions for global data read by data elements and external code. Applies to storage classes Const, ConstVolatile, ExportToFile, and Volatile.

Example: 'DefinitionFile','myDataDefs.c'

Data Types: char | string

GetFunction — Name of get function
character vector | string scalar

Name of a get function that a data element calls in the generated code. Applies to storage class GetSet.

Example: 'GefFunction','myDataGetFunction'

Data Types: char | string

HeaderFile — C header file
character vector | string scalar

File name for a C header file that contains declarations for global data read by data elements and external code. Applies to storage classes Const, ConstVolatile, Define, ExportToFile, GetSet, ImportedDefine, ImportFromFile, and Volatile.

Example: 'HeaderFile','myDataDecl.h'

Data Types: char | string

MemorySection — Name of memory section
character vector | string scalar

Name of a memory section that is defined in the Embedded Coder Dictionary associated with the model.

Example: 'MemorySection','myFastMem'

Data Types: char | string

Owner — Owner of global data
character vector | string scalar

Name of the model that owns global data, which is used by other models in the same model hierarchy. The code generated for the model that owns the data includes the global data definition. Applies to storage classes Const, ConstVolatile, ExportToFile, and Volatile.

Example: 'Owner','myModelA'

Data Types: char | string

PreserveDimensions — Boolean flag indicating whether to preserve dimensions of multidimensional arrays
True | False

When model configuration parameter **Array layout** is set to Row-major, a flag that indicates whether to preserve dimensions of a data element represented in generated code as a multidimensional array. Applies to storage classes Const, ConstVolatile, , ExportToFile, FileScope, ImportFromFile, Localizable, and Volatile.

Example: 'PreserveDimensions','True'

Data Types: logical

SetFunction — Name of set function
character string | string scalar

Name of a set function that a data element calls in the generated code. Applies to storage class GetSet.

Example: 'SetFunction','myDataSetFunction'

Data Types: char | string

StructName — Name of structure
character vector | string scalar

Name that the code generator uses to identify the structure for a data element in the generated code. Applies to storage classes Bitfield and Struct.

Example: 'StructName','myDataStruct'

storageClassPropertyName — Value of storage class property
depends on property definition

Storage class property defined in the Embedded Coder Dictionary. Values that you can specify vary depending on the storage class definition.

Data Types: char | string

See Also

coder.mapping.api.CoderDictionary | getDataDefault | getFunctionDefault | setFunctionDefault

Introduced in R2021a

getDataDefault

Get default code settings for data category

Syntax

```
value = getDataDefault(myCoderDictionaryObj, category, property)
```

Description

`value = getDataDefault(myCoderDictionaryObj, category, property)` returns the value from the code mappings of the specified property for the specified data category.

Examples

Configure default code settings for a data category in a data dictionary

Use the `coder.mapping.api.get` function to access the `CoderDictionary` object associated with the data dictionary.

```
cm = coder.mapping.api.get('codeDefinitions.sldd');
```

To see the storage class of root-level inports for the dictionary, use the `getDataDefault` function.

```
value = getDataDefault(cm, 'Inports', 'StorageClass')
```

```
value =

    'Default'
```

The dictionary uses the default storage class for inports.

To configure the storage class, use the `setDataDefault` function.

```
setDataDefault(cm, 'Inports', 'StorageClass', 'ExportedGlobal')
```

To verify that the storage class of inports is now set to `ExportedGlobal`, use the `getDataDefault` function.

```
value = getDataDefault(cm, 'Inports', 'StorageClass')
```

```
value =

    'ExportedGlobal'
```

Input Arguments

`myCoderDictionaryObj` — Coder dictionary object
`CoderDictionary` object

Coder dictionary object returned by a call to function `coder.mapping.api.get`.

`category` — Model data element category
`Constants` | `ExternalParameterObjects` | `GlobalDataStores` | `Inports` | `InternalData` | `ModelParameters` | `ModelParameterArguments` | `Outports` | `SharedLocalDataStores`

Category of data elements to return a property value for.

Example: `'Inports'`

`property` — Code mapping property value to return
`StorageClass` | `Identifier` | `DefinitionFile` | `GetFunction` | `HeaderFile` | `MemorySection` | `Owner` | `PreserveDimensions` | `SetFunction` | `StructName` | storage class property name

Code mapping property that you return a value for. Specify one of these property names or a property name for a storage class defined in the Embedded Coder Dictionary .

Information to Return	Property Name
Name of storage class	`StorageClass`
Name of variable for data element in the generated code	`Identifier`
Name of source definition file that contains definitions for global data that is read by the data element and external code	`DefinitionFile`
Name of `get` function called by code generated for the data element	`GetFunction`
Name of source header file that contains declarations for global data that is read by the model data element and external code	`HeaderFile`
Character vector or string scalar that names a memory section for a model defined in the Embedded Coder Dictionary.	`MemorySection`
Name of model for which the code generator places the definition for data element shared by multiple models in a model hierarchy	`Owner`
Boolean value indicating whether code generator preserves dimensions of data that is represented as a multidimensional array	`PerserveDimensions`
Name of `set` function called by code generated for data element	`SetFunction`
Name of structure in generated code for data element	`StructName`

Example: `'Identifier'`

Output Arguments

`value` — Code mapping property value of category
character vector

The code mapping property value of the specified category, returned as a character vector.

See Also

`coder.mapping.api.CoderDictionary` | `getFunctionDefault` | `setDataDefault` | `setFunctionDefault`

Introduced in R2021a

setFunctionDefault

Set default function customization template and memory section for model functions category

Syntax

`setFunctionDefault(myCoderDictionaryObj,category,Name,Value)`

Description

`setFunctionDefault(myCoderDictionaryObj,category,Name,Value)` sets the default function customization template and memory section for the specified category of model entry-point functions.

Examples

Configure the default memory section for a data category

Use the `coder.mapping.api.get` function to access the `CoderDictionary` object associated with the data dictionary.

```
cm = coder.mapping.api.get('codeDefinitions.sldd');
```

To see the memory section for `Execution` functions in the dictionary, use the `getFunctionDefault` function.

```
value = getFunctionDefault(cm, 'Execution', 'MemorySection')
```

```
value =

    'None'
```

To configure the memory section for the category, use the `setFunctionDefault` function.

```
setFunctionDefault(cm, 'Execution', 'MemorySection', 'functionFastMem')
```

To verify that the memory section for the `Execution` category is now set to `functionFastMem`, use the `getFunctionDefault` function.

```
value = getFunctionDefault(cm, 'Execution', 'MemorySection')
```

```
value =

    'functionFastMem'
```

Input Arguments

`myCoderDictionaryObj` — Coder dictionary object
`CoderDictionary` object

Coder dictionary object returned by a call to function `coder.mapping.api.get`.

`category` — Model function category
`InitializeTerminate` | `Execution` | `SharedUtility`

Category of model entry-point functions for which to set the function customization template and memory section.

Example: `'Execution'`

Name-Value Pair Arguments

Specify comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments as `Name1,Value1,…,NameN,ValueN`. The order of the name and value pair arguments does not matter.

Example: 'FunctionCustomizationTemplate' 'exFastFunction'

`FunctionCustomizationTemplate` — Name of function customization template
character vector | string scalar

Name of a function customization template defined in the Embedded Coder Dictionary associated with the model. If you set the default function customization template for a category of functions to `Default`, you can specify a memory section for the category of functions.

Data Types: `char` | `string`

`MemorySection` — Name of memory section
`None` | `MemConst` | `MemVolatile` | `MemConstVolatile` | `internalDataMem` | `functionFastMem` | `functionSlowMem`

Name of a memory section that is defined in the Embedded Coder Dictionary associated with the model.

Data Types: `char` | `string`

See Also

`coder.mapping.api.CoderDictionary` | `getDataDefault` | `getFunctionDefault` | `setDataDefault`

Introduced in R2021a

getFunctionDefault

Get default function customization template or memory section for model functions category

Syntax

```
propertyValue = getFunctionDefault(myCoderDictionaryObj,category,property)
```

Description

`propertyValue = getFunctionDefault(myCoderDictionaryObj,category,property)` returns the value of the specified property for the specified function category.

Examples

Configure the default memory section for a data category

Use the `coder.mapping.api.get` function to access the `CoderDictionary` object associated with the data dictionary.

```
cm = coder.mapping.api.get('codeDefinitions.sldd');
```

To see the memory section for `Execution` functions in the dictionary, use the `getFunctionDefault` function.

```
value = getFunctionDefault(cm, 'Execution', 'MemorySection')
```

```
value =

    'None'
```

To configure the memory section for the category, use the `setFunctionDefault` function.

```
setFunctionDefault(cm, 'Execution', 'MemorySection', 'functionFastMem')
```

To verify that the memory section for the `Execution` category is now set to `functionFastMem`, use the `getFunctionDefault` function.

```
value = getFunctionDefault(cm, 'Execution', 'MemorySection')
```

```
value =

    'functionFastMem'
```

Input Arguments

`myCoderDictionaryObj` — Coder dictionary object
`CoderDictionary` object

Coder dictionary object returned by a call to function `coder.mapping.api.get`.

`category` — Model function category
`InitializeTerminate` | `Execution` | `SharedUtility`

Category of model entry-point functions for which to set the function customization template and memory section.

Example: `'Execution'`

`property` — Function customization template or memory section
`FunctionCustomizationTemplate` | `MemorySection`

`FunctionCustomizationTemplate` or `MemorySection` for which to return a value.

Example: `'FunctionCustomizationTemplate'`

Output Arguments

`propertyValue` — Name of function customization template or memory section
character vector | string scalar

Name of the function customization template or memory section.

Data Types: `char` | `string`

See Also

`coder.mapping.api.CoderDictionary` | `getDataDefault` | `setDataDefault` | `setFunctionDefault`

Introduced in R2021a

coder.codedescriptor.CodeDescriptor class

Package: `coder.codedescriptor`

Return information about generated code

Description

Create a `coder.codedescriptor.CodeDescriptor` object to access all the methods defined within the code descriptor API. The `coder.codedescriptor.CodeDescriptor` object describes the data interfaces, function interfaces, global data stores, local and global parameters in the generated code.

Creation

`codeDescObj = coder.getCodeDescriptor(model)` creates a `coder.codedescriptor.CodeDescriptor` object for the specified model.

`codeDescObj = coder.getCodeDescriptor(folder)` creates a `coder.codedescriptor.CodeDescriptor` object for the model in the build folder specified in `folder`.

Properties

`ModelName` — Name of the model
character vector (default)

Name of the model for which the code descriptor object is invoked.

Example: `'rtwdemo_comments'`

`BuildFolder` — Build folder
character vector (default)

Path of the build folder where the model is built.

Example: `'C:\Users\Desktop\Work\rtwdemo_comments_ert_rtw'`

Methods

Public Methods

getAllDataInterfaceTypes	Return data interface types
getAllFunctionInterfaceTypes	Return function interface types
getArrayLayout	Return array layout of the generated code
getDataInterfaceTypes	Return data interface types in the generated code
getDataInterfaces	Return information of the specified data interface
getFunctionInterfaceTypes	Return function interface types in the generated code
getFunctionInterfaces	Return information of the specified function interface

getReferencedModelCodeDescriptor	Return coder.codedescriptor.CodeDescriptor object for the specified referenced model
getReferencedModelNames	Return names of the referenced models

Examples

Create `coder.codedescriptor.CodeDescriptor` Object

1 Build the model.

```
slbuild('rtwdemo_comments')
```

2 Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

```
codeDescObj = coder.getCodeDescriptor('rtwdemo_comments')
```

```
    ModelName: 'rtwdemo_comments'
     BuildDir: 'C:\Users\Desktop\Work\rtwdemo_comments_ert_rtw'
```

3 Return a list of all available function interface types.

```
allFunctionInterfaceTypes = getAllFunctionInterfaceTypes(codeDescObj)
```

```
    {'Initialize'}
    {'Output'    }
    {'Update'    }
    {'Terminate' }
```

See Also

`getCodeDescriptor` | `coder.descriptor.DataInterface` | `coder.descriptor.FunctionInterface`

Topics

"Get Code Description of Generated Code"

Introduced in R2018a

getAllDataInterfaceTypes

Class: `coder.codedescriptor.CodeDescriptor`
Package: `coder.codedescriptor`

Return data interface types

Syntax

`allDataInterfaceTypes = getAllDataInterfaceTypes(codeDescObj)`

Description

`allDataInterfaceTypes = getAllDataInterfaceTypes(codeDescObj)` returns a list of the data interface types. This list is not specific to any model.

Input Arguments

`codeDescObj` — Code Descriptor object
`coder.codedescriptor.CodeDescriptor` object

`coder.codedescriptor.CodeDescriptor` object for which you want to retrieve the information about generated code.

Output Arguments

`allDataInterfaceTypes` — Data interface types available
cell array of character vectors

A list of available data interface types.

Examples

Create a `coder.codedescriptor.CodeDescriptor` object for the required model that is built, then list the available data interface types.

1. Build the model.

   ```
   slbuild('rtwdemo_comments')
   ```
2. Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

   ```
   codeDescObj = coder.getCodeDescriptor('rtwdemo_comments')
   ```
3. Return a list of available data interface types.

   ```
   allDataInterfaceTypes = getAllDataInterfaceTypes(codeDescObj)
   ```

   `allDataInterfaceTypes` has these values:

   ```
       {'Inports'              }
       {'Outports'             }
       {'Parameters'           }
   ```

```
{'GlobalDataStores'     }
{'SharedLocalDataStores'}
{'ExternalParameterObjects'     }
{'ModelParameters'      }
{'InternalData'         }
```

In a model, there can be `ExternalParameterObjects` and/or `LocalParameters`. The data interface type `Parameters` consist of a consolidated list of both types of parameters.

See Also

`coder.codedescriptor.CodeDescriptor` | `coder.descriptor.DataInterface` | `getDataInterfaceTypes` | `getDataInterfaces` | `getCodeDescriptor`

Topics

"Get Code Description of Generated Code"

Introduced in R2018a

getAllFunctionInterfaceTypes

Class: coder.codedescriptor.CodeDescriptor
Package: coder.codedescriptor

Return function interface types

Syntax

```
allFunctionInterfaceTypes = getAllFunctionInterfaceTypes(codeDescObj)
```

Description

allFunctionInterfaceTypes = getAllFunctionInterfaceTypes(codeDescObj) returns a list of the function interface types. The returned list is not specific to any model.

Input Arguments

codeDescObj — Code Descriptor object
coder.codedescriptor.CodeDescriptor object

coder.codedescriptor.CodeDescriptor object for which you want to retrieve the information about generated code.

Output Arguments

allFunctionInterfaceTypes — Function interface types available
cell array of character vectors

A list of the available function interface types.

Examples

Create a coder.codedescriptor.CodeDescriptor object for the required model which is built, then list the available function interface types.

1. Build the model.

```
slbuild('rtwdemo_comments')
```

2. Create a coder.codedescriptor.CodeDescriptor object for the required model.

```
codeDescObj = coder.getCodeDescriptor('rtwdemo_comments')
```

3. Return a list of available function interface types.

```
allFunctionInterfaceTypes = getAllFunctionInterfaceTypes(codeDescObj)
```

allFunctionInterfaceTypes has these values:

```
{'Allocation'}
{'Initialize'}
{'Output'    }
```

```
{'Update'    }
{'Terminate' }
```

See Also
coder.codedescriptor.CodeDescriptor | getFunctionInterfaceTypes | getFunctionInterfaces | getCodeDescriptor | coder.descriptor.FunctionInterface

Topics
"Get Code Description of Generated Code"
"Configure C Code Generation for Model Entry-Point Functions"

Introduced in R2018a

getArrayLayout

Class: `coder.codedescriptor.CodeDescriptor`
Package: `coder.codedescriptor`

Return array layout of the generated code

Syntax

```
arrayLayout = getArrayLayout(codeDescObj)
```

Description

`arrayLayout = getArrayLayout(codeDescObj)` returns the array layout of the model for which the code is generated.

Input Arguments

`codeDescObj` — Code Descriptor object
`coder.codedescriptor.CodeDescriptor` object

`coder.codedescriptor.CodeDescriptor` object for which you want to retrieve the information about generated code.

Output Arguments

`arrayLayout` — Array layout of the generated code
character vectors

Array layout specified for the model by using the model configuration parameter **Array layout**.

Examples

Create a `coder.codedescriptor.CodeDescriptor` object for the model that is built, then list the array layout of the generated code.

1. Open a model.

```
rtwdemo_comments
```

2. Specify the model configuration parameter **Array layout** as `Row-major`. Alternatively, in the command window, use these commands:

```
set_param('rtwdemo_comments', 'ArrayLayout','Row-major');
```

3. Build the model.

```
slbuild('rtwdemo_comments')
```

4. Create a `coder.codedescriptor.CodeDescriptor` object for the model.

```
codeDescObj = coder.getCodeDescriptor('rtwdemo_comments')
```

5. Return the array layout of the generated code.

```
arrayLayout = getArrayLayout(codeDescObj)
```

`arrayLayout` has this value:

```
'Row-major'
```

See Also

`coder.codedescriptor.CodeDescriptor` | `getCodeDescriptor`

Topics

"Get Code Description of Generated Code"
"Code Generation of Matrices and Arrays"

Introduced in R2018b

getDataInterfaces

Class: `coder.codedescriptor.CodeDescriptor`
Package: `coder.codedescriptor`

Return information of the specified data interface

Syntax

```
dataInterface = getDataInterfaces(codeDescObj, dataInterfaceName)
```

Description

`dataInterface = getDataInterfaces(codeDescObj, dataInterfaceName)` returns the type of data, SID, graphical name, timing, implementation, and variant information on the data interface that `dataInterfaceName` specifies.

Input Arguments

`codeDescObj` — Code Descriptor object
`coder.codedescriptor.CodeDescriptor` object

`coder.codedescriptor.CodeDescriptor` object for which you want to retrieve the information about generated code.

`dataInterfaceName` — Name of data interface
`Inports` | `Outports` | `Parameters` | `GlobalDataStores` | `SharedLocalDataStores` | `ExternalParameterObjects` | `ModelParameters` | `InternalData`

`dataInterfaceName` specifies the name of a data interface. To get a list of all the data interfaces in the generated code, call `getDataInterfaceTypes()`.

Data Types: `string`

Output Arguments

`dataInterface` — `coder.descriptor.DataInterface` object with properties of specified data interface type
`coder.descriptor.DataInterface` object | array of `coder.descriptor.DataInterface` objects

The `coder.descriptor.DataInterface` object describes information about the specified data interface such as type of data, SID, graphical name, timing, implementation, and variant information.

Examples

1 Build the model.

```
slbuild('rtwdemo_comments')
```

2 Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

```
codeDescObj = coder.getCodeDescriptor('rtwdemo_comments')
```

3 Return a list of all data interface types in the generated code.

```
dataInterfaceTypes = getDataInterfaceTypes(codeDescObj)
```

`dataInterfaceTypes` has these values:

```
    {'Inports'         }
    {'Outports'        }
    {'Parameters'      }
    {'ExternalParameterObjects'}
```

4 Return properties of Inport blocks in the generated code.

```
dataInterface = getDataInterfaces(codeDescObj, 'Inports')
```

`dataInterface` is an array of `coder.descriptor.DataInterface` objects. Obtain the details of the first Inport block of the model by accessing the first location in the array.

```
dataInterface(1)
```

The first `coder.descriptor.DataInterface` object with properties is returned.

```
              Type: [1×1 coder.descriptor.types.Double]
               SID: 'rtwdemo_comments:1'
     GraphicalName: 'In1'
       VariantInfo: [0×0 coder.descriptor.VariantInfo]
    Implementation: [1×1 coder.descriptor.StructExpression]
            Timing: [1×1 coder.descriptor.TimingInterface]
```

See Also

`coder.codedescriptor.CodeDescriptor` | `getAllDataInterfaceTypes` | `getDataInterfaceTypes` | `coder.descriptor.DataInterface`

Topics

"Get Code Description of Generated Code"

Introduced in R2018a

getDataInterfaceTypes

Class: coder.codedescriptor.CodeDescriptor
Package: coder.codedescriptor

Return data interface types in the generated code

Syntax

```
dataInterfaceTypes = getDataInterfaceTypes(codeDescObj)
```

Description

`dataInterfaceTypes = getDataInterfaceTypes(codeDescObj)` returns a list of the data interface types in the generated code. To get a list of the available data interfaces, call `getAllDataInterfaceTypes()`.

Input Arguments

codeDescObj — Code Descriptor object
coder.codedescriptor.CodeDescriptor object

`coder.codedescriptor.CodeDescriptor` object for which you want to retrieve the information about generated code.

Output Arguments

dataInterfaceTypes — Data interface types in the generated code
cell array of character vectors

A list of the data interface types in the generated code.

Examples

1. Build the model.

   ```
   slbuild('rtwdemo_counter')
   ```
2. Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

   ```
   codeDescObj = coder.getCodeDescriptor('rtwdemo_counter')
   ```
3. Return a list of data interface types in the generated code.

   ```
   dataInterfaceTypes = getDataInterfaceTypes(codeDescObj)
   ```

   `dataInterfaceTypes` has these values for model `rtwdemo_counter`:

   ```
       {'Inports'     }
       {'Outports'    }
       {'InternalData'    }
   ```

See Also

`coder.codedescriptor.CodeDescriptor` | `getAllDataInterfaceTypes` | `getDataInterfaces` | `getCodeDescriptor`

Topics

"Get Code Description of Generated Code"

Introduced in R2018a

getFunctionInterfaces

Class: coder.codedescriptor.CodeDescriptor
Package: coder.codedescriptor

Return information of the specified function interface

Syntax

```
functionInterface = getFunctionInterfaces(codeDescObj, functionInterfaceName)
```

Description

functionInterface = getFunctionInterfaces(codeDescObj, functionInterfaceName) returns the function prototype, input arguments, return arguments, variant conditions, and timing information of the function interface that functionInterfaceName specifies.

Input Arguments

codeDescObj — Code Descriptor object
coder.codedescriptor.CodeDescriptor object

coder.codedescriptor.CodeDescriptor object for which you want to retrieve the information about generated code.

functionInterfaceName — Name of function interface
Allocation | Initialize | Output | Update | Terminate

functionInterfaceName specifies the name of a function interface. A list of all the function interfaces in the generated code is returned by getFunctionInterfaceTypes().

Data Types: string

Output Arguments

functionInterface — coder.descriptor.FunctionInterface object with properties of specified function interface type
coder.descriptor.FunctionInterface object | array of coder.descriptor.FunctionInterface objects

The coder.descriptor.FunctionInterface object describes information about the specified function interface such as function prototype, input arguments, return arguments, variant conditions, and timing information.

Examples

1. Build the model.

   ```
   slbuild('rtwdemo_comments')
   ```
2. Create a coder.codedescriptor.CodeDescriptor object for the required model.

```
codeDescObj = coder.getCodeDescriptor('rtwdemo_comments')
```

3 Return a list of all function interface types in the generated code.

```
functionInterfaceTypes = getFunctionInterfaceTypes(codeDescObj)
```

These are the function interface types in the generated code of model `rtwdemo_comments`:

```
    {'Initialize'}
    {'Output'    }
```

4 Return properties of a specified function interface in the generated code.

```
functionInterface = getFunctionInterfaces(codeDescObj, 'Output')
```

`functionInterface` is a `coder.descriptor.FunctionInterface` object.

```
       Prototype: [1×1 coder.descriptor.types.Prototype]
    ActualReturn: [0×0 coder.descriptor.DataInterface]
     VariantInfo: [0×0 coder.descriptor.VariantInfo]
          Timing: [1×1 coder.descriptor.TimingInterface]
      ActualArgs: [1x0 coder.descriptor.DataInterface List]
```

See Also

`coder.codedescriptor.CodeDescriptor` | `getAllFunctionInterfaceTypes` | `getFunctionInterfaceTypes` | `coder.descriptor.FunctionInterface`

Topics

"Get Code Description of Generated Code"

Introduced in R2018a

getFunctionInterfaceTypes

Class: `coder.codedescriptor.CodeDescriptor`
Package: `coder.codedescriptor`

Return function interface types in the generated code

Syntax

```
functionInterfaceTypes = getFunctionInterfaceTypes(codeDescObj)
```

Description

`functionInterfaceTypes = getFunctionInterfaceTypes(codeDescObj)` returns a list of the function interface types in the generated code. To get a list of the available function interfaces, call `getAllFunctionInterfaceTypes()`.

Input Arguments

`codeDescObj` — Code Descriptor object
`coder.codedescriptor.CodeDescriptor` object

`coder.codedescriptor.CodeDescriptor` object for which you want to retrieve the information about generated code.

Output Arguments

`functionInterfaceTypes` — Function interface types in the generated code
cell array of character vectors

A list of the data interface types in the generated code.

Examples

1. Build the model.

```
slbuild('rtwdemo_counter')
```

2. Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

```
codeDescObj = coder.getCodeDescriptor('rtwdemo_counter')
```

3. Return a list of function interface types in the generated code.

```
functionInterfaceTypes = getFunctionInterfaceTypes(codeDescObj)
```

`functionInterfaceTypes` has these values for model `rtwdemo_counter`:

```
{'Initialize'}
{'Output'    }
```

See Also

`coder.codedescriptor.CodeDescriptor` | `getAllFunctionInterfaceTypes` | `getFunctionInterfaces` | `getCodeDescriptor`

Topics

"Get Code Description of Generated Code"

Introduced in R2018a

getReferencedModelCodeDescriptor

Class: coder.codedescriptor.CodeDescriptor
Package: coder.codedescriptor

Return coder.codedescriptor.CodeDescriptor object for the specified referenced model

Syntax

```
refCodeDescriptor = getReferencedModelCodeDescriptor(codeDescObj,
refModelName)
```

Description

refCodeDescriptor = getReferencedModelCodeDescriptor(codeDescObj, refModelName) returns the coder.codedescriptor.CodeDescriptor object for the referenced model specified in refModelName.

Input Arguments

codeDescObj — Code Descriptor object
coder.codedescriptor.CodeDescriptor object

coder.codedescriptor.CodeDescriptor object for which you want to retrieve the information about generated code.

refModelName — Name of referenced model
string

refModelName can take any name from the list of referenced models returned by getReferencedModelNames().

Output Arguments

refCodeDescriptor — coder.codedescriptor.CodeDescriptor object for the specified referenced model
coder.codedescriptor.CodeDescriptor object

coder.codedescriptor.CodeDescriptor object for the specified referenced model.

Examples

1. Build the model.

```
slbuild('rtwdemo_async_mdlreftop')
```

2. Create a coder.codedescriptor.CodeDescriptor object for the required model.

```
codeDescObj = coder.getCodeDescriptor('rtwdemo_async_mdlreftop')
```

3. Return a list of referenced models.

```
refModels = getReferencedModelNames(codeDescObj)
```

`refModels` contains the list of referenced models for `rtwdemo_async_mdlreftop`.

```
{'rtwdemo_async_mdlrefbot'}
```

Obtain the `coder.codedescriptor.CodeDescriptor` object for any of the referenced models.

```
refCodeDescriptorObj = getReferencedModelCodeDescriptor(codeDescObj, 'rtwdemo_async_mdlrefbot')
```

`refCodeDescriptorObj` is the `coder.codedescriptor.CodeDescriptor` object for `rtwdemo_async_mdlrefbot` model.

```
    ModelName: 'rtwdemo_async_mdlrefbot'
     BuildDir: 'C:\Users\Desktop\Work\slprj\tornado\rtwdemo_async_mdlrefbot'
```

See Also

`coder.codedescriptor.CodeDescriptor` | `getReferencedModelNames` | `getCodeDescriptor`

Topics

"Get Code Description of Generated Code"

Introduced in R2018a

getReferencedModelNames

Class: `coder.codedescriptor.CodeDescriptor`
Package: `coder.codedescriptor`

Return names of the referenced models

Syntax

`refModels = getReferencedModelNames(codeDescObj)`

Description

`refModels = getReferencedModelNames(codeDescObj)` returns a list of referenced models for a `coder.codedescriptor.CodeDescriptor` object.

Input Arguments

`codeDescObj` — Code Descriptor object
`coder.codedescriptor.CodeDescriptor` object

`coder.codedescriptor.CodeDescriptor` object for which you want to retrieve the information about generated code.

Output Arguments

`refModels` — Names of referenced models
cell array of character vectors

A list of referenced models.

Examples

1. Build the model.

```
slbuild('rtwdemo_async_mdlreftop')
```

2. Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

```
codeDescObj = coder.getCodeDescriptor('rtwdemo_async_mdlreftop')
```

3. Return a list of referenced models.

```
refModels = getReferencedModelNames(codeDescObj)
```

`refModels` has the list of referenced models.

```
{'rtwdemo_async_mdlrefbot'}
```

See Also

`coder.codedescriptor.CodeDescriptor` | `getReferencedModelCodeDescriptor`

Topics
"Get Code Description of Generated Code"

Introduced in R2018a

isLookupTableDataInterface

Determine whether object is a `coder.descriptor.LookupTableDataInterface` object

Syntax

```
lookupTableDataInterface = isLookupTableDataInterface(parameterObj)
```

Description

`lookupTableDataInterface = isLookupTableDataInterface(parameterObj)` returns a logical value indicating whether the object is a `coder.descriptor.LookupTableDataInterface` object.

Input Arguments

`parameterObj` — `coder.descriptor.LookupTableDataInterface` object
`coder.descriptor.LookupTableDataInterface` object

`coder.descriptor.LookupTableDataInterface` object that represents a Lookup Table block in the model.

Data Types: `string`

Output Arguments

`lookupTableDataInterface` — logical value
`1` | `0`

Logical value indicating whether the object is a `coder.descriptor.LookupTableDataInterface`.

Data Types: `logical`

Examples

Determine if the object is a `coder.descriptor.LookupTableDataInterface` object

1. Build the model.

   ```
   slbuild('rtwdemo_asap2')
   ```
2. Create a `coder.codedescriptor.CodeDescriptor` object for the model.

   ```
   codeDescObj = coder.getCodeDescriptor('rtwdemo_asap2')
   ```
3. Return properties of the Lookup Table parameters in the model.

   ```
   params = getDataInterfaces(codeDescObj, 'Parameters')
   ```

   The `params` variable is an array of `coder.descriptor.DataInterface`, `coder.descriptor.LookupTableDataInterface`, and `coder.descriptor.BreakpointDataInterface` objects.

4 Obtain the details of the model Lookup Table block by accessing the sixth location in the array.

```
parameterObj = params(6)
```

5 Determine if the object stored in `parameterObj` variable is a `coder.descriptor.LookupTableDataInterface` object.

```
lookupTableDataInterface = isLookupTableDataInterface(parameterObj)

lookupTableDataInterface =

  logical

  1
```

The code generator returns a logical value of `1` if `parameterObj` is a `coder.descriptor.LookupTableDataInterface` object. Otherwise, the code generator returns a logical value of `0`.

See Also

`coder.descriptor.LookupTableDataInterface` | `isBreakpointDataInterface`

Introduced in R2020a

isBreakpointDataInterface

Determine whether object is a `coder.descriptor.BreakpointDataInterface` object

Syntax

```
breakpointTableDataInterface = isBreakpointDataInterface(parameterObj)
```

Description

`breakpointTableDataInterface = isBreakpointDataInterface(parameterObj)` returns a logical value indicating whether the object is a `coder.descriptor.BreakpointDataInterface` object or not.

Input Arguments

`parameterObj` — `coder.descriptor.BreakpointDataInterface` object
`coder.descriptor.BreakpointDataInterface`

`coder.descriptor.BreakpointDataInterface` object that represents a breakpoint set in the model.

Data Types: `string`

Output Arguments

`breakpointTableDataInterface` — logical value
`1` | `0`

Logical value indicating whether the object is a `coder.descriptor.BreakpointDataInterface`.

Data Types: `logical`

Examples

Determine if the object is a `coder.descriptor.BreakpointDataInterface` object

1. Build the model.

   ```
   slbuild('rtwdemo_asap2')
   ```
2. Create a `coder.codedescriptor.CodeDescriptor` object for the model.

   ```
   codeDescObj = coder.getCodeDescriptor('rtwdemo_asap2')
   ```
3. Return properties of the breakpoint set data in the model.

   ```
   params = getDataInterfaces(codeDescObj, 'Parameters')
   ```

   The `params` variable is an array of `coder.descriptor.DataInterface`, `coder.descriptor.LookupTableDataInterface`, and `coder.descriptor.BreakpointDataInterface` objects.

4. Obtain the details of the breakpoint set attached to the model Lookup Table block by accessing the first location in the array.

   ```
   parameterObj = params(6).Breakpoints(1)
   ```
5. Determine if the object stored in `parameterObj` variable is a `coder.descriptor.BreakpointDataInterface` object.

   ```
   breakpointDataInterface = isBreakpointDataInterface(parameterObj)

   lookupTableDataInterface =

     logical

     1
   ```

   The code generator returns a logical value of `1` indicating if `parameterObj` is a `coder.descriptor.BreakpointDataInterface` object. Otherwise, the code generator returns a logical value of `0`.

See Also

`isLookupTableDataInterface` | `coder.descriptor.BreakpointDataInterface`

Introduced in R2020a

getAllParameters

Return all associated `coder.descriptor.LookupTableDataInterface` and `coder.descriptor.BreakpointDataInterface` objects

Syntax

```
dataInterface = getAllParameters(parameterObj)
```

Description

`dataInterface = getAllParameters(parameterObj)` returns all associated `coder.descriptor.LookupTableDataInterface` and `coder.descriptor.BreakpointDataInterface` objects.

Input Arguments

`parameterObj` — `coder.descriptor.LookupTableDataInterface` object
`coder.descriptor.LookupTableDataInterface` object

The `coder.descriptor.LookupTableDataInterface` object that represents a Lookup Table block in the model.

Data Types: `string`

Output Arguments

`dataInterface` — array of `coder.descriptor.LookupTableDataInterface` and/or `coder.descriptor.BreakpointDataInterface` objects
`coder.descriptor.LookupTableDataInterface` object | array of `coder.descriptor.LookupTableDataInterface` and/or `coder.descriptor.BreakpointDataInterface` objects

The `coder.descriptor.LookupTableDataInterface` object represents a Lookup Table block in the model. The `coder.descriptor.BreakpointDataInterface` object represents the breakpoint set data associated with the Lookup Table block.

Examples

Return all associated `coder.descriptor.LookupTableDataInterface` and `coder.descriptor.BreakpointDataInterface` objects

1. Build the model.

   ```
   slbuild('rtwdemo_asap2')
   ```
2. Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

   ```
   codeDescObj = coder.getCodeDescriptor('rtwdemo_asap2')
   ```
3. Return properties of the Lookup Table parameters in the model.

```
params = getDataInterfaces(codeDescObj, 'Parameters')
```

The `params` variable is an array of `coder.descriptor.DataInterface`, `coder.descriptor.LookupTableDataInterface`, and `coder.descriptor.BreakpointDataInterface` objects.

4. Obtain the details of the model Lookup Table block by accessing the sixth location in the array.

   ```
   parameterObj = params(6)
   ```
5. Retrieve all the associated `coder.descriptor.LookupTableDataInterface` and `coder.descriptor.BreakpointDataInterface` objects by using the `parameterObj`.

   ```
   dataInterface = getAllParameters(parameterObj)
   ```

   The code generator returns an array of `coder.descriptor.LookupTableDataInterface` and `coder.descriptor.BreakpointDataInterface` objects.

   ```
   1×3 heterogeneous DataInterface (LookupTableDataInterface, BreakpointDataInterface) array with properties:

       Type
       SID
       GraphicalName
       VariantInfo
       Implementation
       Timing
       Unit
       Range
   ```

See Also

`coder.descriptor.LookupTableDataInterface` | `coder.descriptor.BreakpointDataInterface` | `coder.descriptor.DataInterface`

Introduced in R2020a

coder.descriptor.DataInterface class

Package: coder.descriptor

Return information about different types of data interfaces

Description

The coder.descriptor.DataInterface object describes various properties for a specified data interface in the generated code. The different types of data interfaces are:

- Root-level inports and outports: An interface between the model and external models or systems, for exchanging data.
- Parameters: Local and global parameters that describe the data for the block, lookup table, and the associated breakpoint set data.
- Data Stores: A repository to store global and shared data that can be written and read.
- Internal data: Internal data structures including DWork vectors, block I/O, and zero-crossings.

If your model has a Stateflow chart that uses machine-parented data, the code generator generates a DWork structure in the generated code. When you use the getDataInterfaces method, you cannot access these structures as InternalData.

Creation

dataInterface = getDataInterfaces(codeDescObj, dataInterfaceName) creates a coder.descriptor.DataInterface object. The codeDescObj object is the coder.codedescriptor.CodeDescriptor object created for the model by using the getCodeDescriptor function.

Input Arguments

dataInterfaceName — Name of data interface
Inports | Outports | Parameters | GlobalDataStores | SharedLocalDataStores | ExternalParameterObjects | ModelParameters | InternalData

Name of the specified data interface.

Example: 'Inports'

Data Types: string

Properties

Type — Type of data
coder.descriptor.types object

The data type associated with the data such as integer, double, matrix, and its properties.

SID — Simulink identifier
character vector

The Simulink identifier (SID) is a unique number within the model that Simulink assigns to the block.

GraphicalName — Name of graphical entity
character vector

The name of the associated graphical entity.

VariantInfo — Variant conditions in the model
coder.descriptor.VariantInfo object

The variant conditions in the model that interact with the data interface.

Implementation — Description of implementation of data
coder.descriptor.DataImplementation object

The description of how the data in the generated code is implemented. This property describes characteristics such as data type and size. In addition, it describes how the data is accessed or declared in the code. The property describes if the data is declared as a variable or structure member.

Timing — Data access rate in run-time environment
coder.descriptor.TimingInterface object

The rate at which data is accessed in a run-time environment.

Unit — Physical unit as attribute on signals
character vector

Specified physical units as attributes on signals at the boundaries of model components.

Range — Range of output value
coder.descriptor.Range object

The range of valid values for the block output signals.

Limitations

A bitfield data structure is generated if you select these configuration parameters:

- **Pack Boolean data into bitfields**
- **Use bitset for storing state configuration**
- **Use bitset for storing Boolean data**

If the `coder.descriptor.DataInterface` represents a bitfield data structure, the Implementation property of the `coder.descriptor.DataInterface` object is empty.

Examples

Get All Data Interface Types

1 Build the model.

```
slbuild('rtwdemo_comments')
```

2. Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

   ```
   codeDescObj = coder.getCodeDescriptor('rtwdemo_comments')
   ```
3. Return a list of all data interface types in the generated code.

   ```
   dataInterfaceTypes = getDataInterfaceTypes(codeDescObj)
   ```

   ```
       {'Inports'          }
       {'Outports'         }
       {'Parameters'       }
       {'ExternalParameterObjects'}
       {'InternalData'     }
   ```
4. Return properties of a specified data interface in the generated code.

   ```
   dataInterface = getDataInterfaces(codeDescObj, 'Inports')
   ```

   `dataInterface` is an array of `coder.descriptor.DataInterface` objects. Obtain the details of the first Inport block of the model by accessing the first location in the array.

   ```
   dataInterface(1)
   ```

   The first `coder.descriptor.DataInterface` object with properties is returned.

   ```
                 Type: [1×1 coder.descriptor.types.Double]
                  SID: 'rtwdemo_comments:1'
         GraphicalName: 'In1'
           VariantInfo: [0×0 coder.descriptor.VariantInfo]
        Implementation: [1×1 coder.descriptor.StructExpression]
                Timing: [1×1 coder.descriptor.TimingInterface]
   ```

See Also

`coder.codedescriptor.CodeDescriptor` | `getAllDataInterfaceTypes` | `getDataInterfaceTypes` | `getDataInterfaces`

Topics

"Get Code Description of Generated Code"

Introduced in R2018a

coder.descriptor.FunctionInterface class

Package: `coder.descriptor`

Return information about entry-point functions

Description

The function interfaces are the entry-point functions in the generated code. The `coder.descriptor.FunctionInterface` object describes various properties for a specified function interface. The different types of function interfaces are:

- Allocation: Contains memory allocation code based on the target of the model. See `model_initialize`.
- Initialize: Contains initialization code for the model and is called once at the start of your application code. See `model_initialize`.
- Output: Contains the output code for the blocks in the model. See `model_step`.
- Update: Contains the update code for the blocks in the model. See `model_step`.
- Terminate: Contains the termination code for the model and is called as part of a system shutdown. See `model_terminate`.

Creation

`functionInterface = getFunctionInterfaces(codeDescObj, functionInterfaceName)` creates a `coder.descriptor.FunctionInterface` object. `codeDescObj` is the `coder.codedescriptor.CodeDescriptor` object created for the model by using the `getCodeDescriptor` function.

Input Arguments

`functionInterfaceName` — Name of function interface
`Allocation` | `Initialize` | `Output` | `Update` | `Terminate`

Name of the specified function interface

Example: `'Output'`

Data Types: `string`

Properties

`Prototype` — Description of function prototype
`coder.descriptor.types` object

The description of the function prototype including function return value, name, arguments, header, and source files.

`ActualReturn` — Return arguments from the function
`coder.descriptor.DataInterface` object

The data that the function returns as a return argument. When there is no data returned from the function, this field is empty.

VariantInfo — Variant conditions in the model
`coder.descriptor.VariantInfo` object

The variant conditions in the model that interact with the function interface.

Timing — Function access rate in run-time environment
`coder.descriptor.TimingInterface` object

The rate at which function is accessed in a run-time environment.

ActualArgs — Input arguments to the function
`coder.descriptor.DataInterfaceList` object

The data passed as arguments to the function. When there is no data passed as an argument to the function, this field is empty.

Examples

Get All Function Interface Types

1. Build the model.

```
slbuild('rtwdemo_comments')
```

2. Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

```
codeDescObj = coder.getCodeDescriptor('rtwdemo_comments')
```

3. Return a list of all function interface types in the generated code.

```
functionInterfaceTypes = getFunctionInterfaceTypes(codeDescObj)

    {'Initialize'}
    {'Output'    }
```

4. Return properties of a specified function interface in the generated code.

```
functionInterface = getFunctionInterfaces(codeDescObj, 'Output')

        Prototype: [1×1 coder.descriptor.types.Prototype]
     ActualReturn: [0×0 coder.descriptor.DataInterface]
      VariantInfo: [0×0 coder.descriptor.VariantInfo]
           Timing: [1×1 coder.descriptor.TimingInterface]
       ActualArgs: [1×0 coder.descriptor.DataInterface List]
```

See Also

`coder.codedescriptor.CodeDescriptor` | `getAllFunctionInterfaceTypes` | `getFunctionInterfaceTypes` | `getFunctionInterfaces`

Topics
"Get Code Description of Generated Code"

Introduced in R2018a

coder.descriptor.LookupTableDataInterface class

Package: `coder.descriptor`
Superclasses: `coder.descriptor.DataInterface`

Return information about Lookup Table blocks that have tunable parameters

Description

The `coder.descriptor.LookupTableDataInterface` object describes various properties for these Lookup Table blocks that have tunable parameters in the generated code:

- 1-D Lookup Table
- 2-D Lookup Table
- n-D Lookup Table
- Interpolation Using Prelookup
- Direct Lookup Table (n-D)
- Sine
- Cosine

Creation

`params = getDataInterfaces(codeDescObj, dataInterfaceName)` creates a `coder.descriptor.LookupTableDataInterface` object if the model has a Lookup Table block that has tunable parameters. The `codeDescObj` object is the `coder.codedescriptor.CodeDescriptor` object created for the model by using the `getCodeDescriptor` function.

Input Arguments

`dataInterfaceName` — Name of data interface
`Parameters`

Specify the `Parameters` data interface type.

Example: `Parameters`

Properties

Type — Type of data
`coder.descriptor.types` object

The data type associated with the data such as `integer`, `double`, `matrix`, and its properties.

`SID` — Simulink identifier
character vector

The Simulink identifier (SID) is a unique number within the model that Simulink assigns to the block.

`GraphicalName` — Name of the tunable parameter for the table data
character vector

The name of the associated tunable parameter for the table data.

`VariantInfo` — Variant conditions in the model
`coder.descriptor.VariantInfo` object

The variant conditions in the model that interact with the data interface.

`Implementation` — Description of implementation of data
`coder.descriptor.DataImplementation` object

Description of how the data in the generated code is implemented. This property describes characteristics such as data type and size. It also describes how the data is accessed or declared in the code. The property describes if the data is declared as a variable or structure member.

`Timing` — Data access rate in run-time environment
`coder.descriptor.TimingInterface` object

The rate at which data is accessed in a run-time environment.

`Unit` — Physical unit as attribute on signals
character vector

Specified physical units as attributes on signals at the boundaries of model components.

`Range` — Range of output value
`coder.descriptor.Range` object

The range of valid values for the block output signals.

`SupportTunableSize` — Tunability of table size
`1` (default) | `0`

Value that represents whether table is enabled for tunability of the effective size of the table, represented as `0` or `1`.

Data Types: `logical`

`BreakpointSpecification` — Source of breakpoint set information in ASAP2 specification
`'Explicit values'` (default) | `'Reference'` | `'Even spacing'`

Source of the breakpoint set information, specified as `'Explicit values'` (default), `'Even spacing'`, or `'Reference'`. The breakpoint specification is mapped as:

For more information on ASAP2 lookup tables, see “Define ASAP2 Information for Lookup Tables”.

Data Types: `char`

`Output` — Data interface for the output of Lookup Table block
`coder.descriptor.DataInterface` object

Return value of the lookup table operation.

Breakpoints — Breakpoint set data
coder.descriptor.BreakpointDataInterface object

Vector of `coder.descriptor.BreakpointDataInterface` objects that are used in the Lookup Table block. These objects contain the breakpoint set data.

Methods

Public Methods

isLookupTableDataInterface	Determine whether object is a coder.descriptor.LookupTableDataInterface object
getAllParameters	Return all associated coder.descriptor.LookupTableDataInterface and coder.descriptor.BreakpointDataInterface objects

Examples

Get Lookup Table block information

1. Build the model.

   ```
   slbuild('rtwdemo_asap2')
   ```
2. Create a `coder.codedescriptor.CodeDescriptor` object for the model.

   ```
   codeDescObj = coder.getCodeDescriptor('rtwdemo_asap2')
   ```
3. Return properties of the Lookup Table parameters in the model.

   ```
   params = getDataInterfaces(codeDescObj, 'Parameters')
   ```

   The `params` variable is an array of `coder.descriptor.DataInterface` and `coder.descriptor.LookupTableDataInterface` objects. The model `rtwdemo_asap2` contains three Lookup Table blocks. Only two of them have tunable breakpoint set data. The code generator creates only two corresponding `coder.descriptor.LookupTableDataInterface` objects.

   Obtain the details of the Standard_Axis block by accessing the sixth location in the array.

   ```
   params(6)
   ```

   The `coder.descriptor.LookupTableDataInterface` object with properties is returned.

   ```
                    Type: [1×1 coder.descriptor.types.Type]
                     SID: 'rtwdemo_asap2:14'
           GraphicalName: 'tabledata'
             VariantInfo: [1×0 coder.descriptor.VariantInfo]
          Implementation: [1×1 coder.descriptor.DataImplementation]
                  Timing: [1×0 coder.descriptor.TimingInterface]
                    Unit: ''
                   Range: [1×0 coder.descriptor.Range]
      SupportTunableSize: 0
 BreakpointSpecification: 'Explicit values'
                  Output: [1×1 coder.descriptor.DataInterface]
             Breakpoints: [1×2 coder.descriptor.BreakpointDataInterface Sequence]
   ```

See Also

`coder.codedescriptor.CodeDescriptor` | `coder.descriptor.DataInterface` | `coder.descriptor.BreakpointDataInterface`

Introduced in R2020a

coder.descriptor.BreakpointDataInterface class

Package: `coder.descriptor`
Superclasses: `coder.descriptor.DataInterface`

Return information about tunable breakpoint set data for a lookup table that has tunable parameters

Description

The `coder.descriptor.BreakpointDataInterface` object describes various properties for breakpoint set data for these Lookup Table blocks that have tunable parameters in the generated code:

- 1-D Lookup Table
- 2-D Lookup Table
- n-D Lookup Table
- Interpolation Using Prelookup
- Direct Lookup Table (n-D)
- Sine
- Cosine

Creation

`params = getDataInterfaces(codeDescObj, dataInterfaceName)` creates a `coder.descriptor.BreakpointDataInterface` object for each dimension in the lookup table. The `codeDescObj` object is the `coder.codedescriptor.CodeDescriptor` object created for the model by using the `getCodeDescriptor` function.

The `coder.descriptor.LookupTableDataInterface` and `coder.descriptor.BreakpointDataInterface` objects are created if these conditions are true:

- Lookup table data is tunable.
- One of these conditions is true:
  - Breakpoint set data is tunable.
  - Breakpoint set data is nontunable and the block does not use a `Simulink.LookupTable` object.
  - The block uses a `Simulink.LookupTable` object.

Input Arguments

`dataInterfaceName` — Name of data interface
`Parameters`

Specify the `Parameters` data interface type.

Example: `Parameters`

Properties

Type — Type of data
coder.descriptor.types object

The data type associated with the data such as integer, double, matrix, and its properties.

SID — Simulink identifier
character vector

The Simulink identifier (SID) is a unique number within the model that Simulink assigns to a block.

GraphicalName — Name of the tunable parameter for the breakpoints
character vector

The name of the associated tunable parameter for the breakpoints.

VariantInfo — Variant conditions in the model
coder.descriptor.VariantInfo object

The variant conditions in the model that interact with the data interface.

Implementation — Description of implementation of data
coder.descriptor.DataImplementation object

Description of how the data in the generated code is implemented. This property describes characteristics such as data type and size. It also describes how the data is accessed or declared in the code. The property describes if the data is declared as a variable or structure member.

Timing — Data access rate in run-time environment
coder.descriptor.TimingInterface object

The rate at which data is accessed in a run-time environment.

Unit — Physical unit as attribute on signals
character vector

Specified physical units as attributes on signals at the boundaries of model components.

Range — Range of output value
coder.descriptor.Range object

The range of valid values for the block output signals.

OperatingPoint — Input value to the lookup table relative to each breakpoint
coder.descriptor.DataInterface object

To find the input value in the table, the operating point uses relative breakpoint set data.

SupportTunableSize — Option to generate code that enables tunability of table size
1 (default) | 0

Option to generate code that enables tunability of the effective size of the table, specified as 0 or 1.

Data Types: logical

`FixAxisMetadata` — Description of breakpoint set data

`coder.descriptor.FixAxisMetadata`

Description of breakpoint set data that is either evenly spaced or non-evenly spaced. The `coder.descriptor.FixAxisMetadata` object is created only if the lookup table data is tunable and the breakpoint set data is not tunable.

Methods

Public Methods

isBreakpointDataInterface	Determine whether object is a coder.descriptor.BreakpointDataInterface object

Examples

Get breakpoint data set information

1. Build the model.

   ```
   slbuild('rtwdemo_asap2')
   ```
2. Create a `coder.codedescriptor.CodeDescriptor` object for the model.

   ```
   codeDescObj = coder.getCodeDescriptor('rtwdemo_asap2')
   ```
3. Return properties of the Lookup Table parameters in the model.

   ```
   params = getDataInterfaces(codeDescObj, 'Parameters')
   ```

   The `params` variable is an array of `coder.descriptor.DataInterface` and `coder.descriptor.LookupTableDataInterface` objects. The model `rtwdemo_asap2` contains three Lookup Table blocks. Only two of them have tunable breakpoint set data. The code generator creates only two corresponding `coder.descriptor.LookupTableDataInterface` objects.

   Obtain the details of the Standard_Axis block by accessing the sixth location in the array.

   ```
   params(6)
   ```

   The `coder.descriptor.LookupTableDataInterface` object with properties is returned.

   ```
                      Type: [1×1 coder.descriptor.types.Type]
                       SID: 'rtwdemo_asap2:14'
             GraphicalName: 'tabledata'
               VariantInfo: [1×0 coder.descriptor.VariantInfo]
            Implementation: [1×1 coder.descriptor.DataImplementation]
                    Timing: [1×0 coder.descriptor.TimingInterface]
                      Unit: ''
                     Range: [1×0 coder.descriptor.Range]
        SupportTunableSize: 0
   BreakpointSpecification: 'Explicit values'
                    Output: [1×1 coder.descriptor.DataInterface]
               Breakpoints: [1×2 coder.descriptor.BreakpointDataInterface Sequence]
   ```
4. The Breakpoints property of the `coder.descriptor.LookupTableDataInterface` object holds a vector of `coder.descriptor.BreakpointDataInterface` objects. Obtain the details of the breakpoint set attached to the model Lookup Table block by accessing the first location in the array.

   ```
   params(6).Breakpoints(1)
   ```

The `coder.descriptor.BreakpointDataInterface` object with properties is returned.

```
                  Type: [1×1 coder.descriptor.types.Type]
                   SID: 'rtwdemo_asap2:14'
         GraphicalName: 'tabledata'
           VariantInfo: [1×0 coder.descriptor.VariantInfo]
        Implementation: [1×1 coder.descriptor.DataImplementation]
                Timing: [1×0 coder.descriptor.TimingInterface]
                  Unit: ''
                 Range: [1×0 coder.descriptor.Range]
        OperatingPoint: [1×1 coder.descriptor.DataInterface]
    SupportTunableSize: 0
       FixAxisMetadata: [1×0 coder.descriptor.FixAxisMetadata]
```

See Also

`coder.codedescriptor.CodeDescriptor` | `coder.descriptor.DataInterface` | `coder.descriptor.LookupTableDataInterface` | `coder.descriptor.FixAxisMetadata`

Introduced in R2020a

coder.descriptor.FixAxisMetadata class

Package: coder.descriptor

Abstract class to get breakpoint set data information

Description

Abstract base class to get breakpoint set data information. Based on the breakpoint set data, you can get either a coder.descriptor.EvenSpacingMetadata object or a coder.descriptor.NonEvenSpacingMetadata object. To get breakpoint set data information, use the coder.descriptor.BreakpointDataInterface object.

You can get a coder.descriptor.FixAxisMetadata object if the model meets these conditions:

- Table data is tunable.

  Table data is tunable if the Lookup Table block uses a Simulink.Parameter object that has a non-Auto storage class or the model configuration parameter **Default parameter behavior** is Tunable.
- Breakpoint set data is not tunable.

  Breakpoint set data is tunable if the Lookup Table block uses a Simulink.Parameter object that has a non-Auto storage class or the model configuration parameter **Default parameter behavior** is Tunable.

Creation

params = getDataInterfaces(codeDescObj, dataInterfaceName) creates a coder.descriptor.BreakpointDataInterface object for each dimension in the lookup table. The codeDescObj object is the coder.codedescriptor.CodeDescriptor object created for the model by using the getCodeDescriptor function.

The coder.descriptor.BreakpointDataInterface object has property FixAxisMetadata that contains a coder.descriptor.FixAxisMetadata object.

Input Arguments

dataInterfaceName — Name of data interface
Parameters

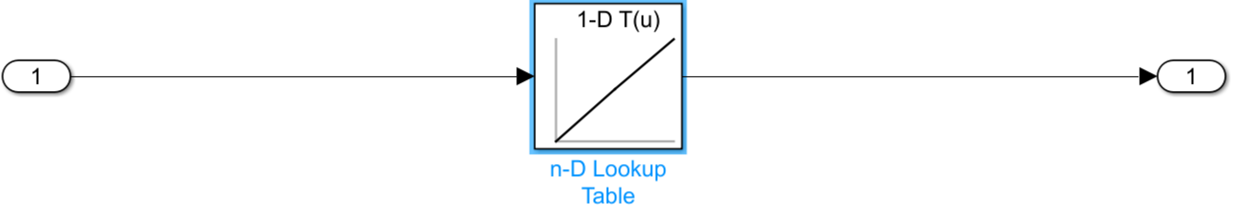
Specify the Parameters data interface type.

Example: Parameters

Examples

Get breakpoint set data information

Consider creating model codeDescDemo or a model with similar specifications.



The model contains an n-D Lookup Table. The n-D Lookup Table block takes table data from a model workspace variable named `tableData` that has a value of `[4 5 6]`. The `tableData` is a `Simulink.Parameter` object that has a `non-Auto` storage class. The breakpoint set data is specified as `[2 6 10]`.

Block Parameters: n-D Lookup Table

Lookup Table (n-D)

Perform n-dimensional interpolated table lookup including index searches. The table is a sampled representation of a function in N variables. Breakpoint sets relate the input values to positions in the table. The first dimension corresponds to the top (or left) input port.

Table and Breakpoints | Algorithm | Data Types

Number of table dimensions: 1

Data specification: Table and breakpoints

Breakpoints specification: Explicit values

	Source	Value
Table data:	Dialog	tableData
Breakpoints 1:	Dialog	[2 6 10]

Edit table and breakpoints...

OK Cancel Help Apply

The model configuration parameter **Default parameter behavior** is set to `Inlined`.

1 Build the model and create a `coder.codedescriptor.CodeDescriptor` object for the model.

```
codeDescObj = coder.getCodeDescriptor('codeDescDemo')
```

2 Retrieve properties of the Lookup Table block and breakpoint set in the generated code.

```
params = getDataInterfaces(codeDescObj, 'Parameters')
```

The `params` variable is an array of `coder.descriptor.LookupTableDataInterface` and `coder.descriptor.BreakpointDataInterface` objects.

```
  LookupTableDataInterface with properties:
                      Type: [1×1 coder.descriptor.types.Type]
                       SID: 'demoModel:22'
              GraphicalName: 'tableData'
                VariantInfo: [1×0 coder.descriptor.VariantInfo]
             Implementation: [1×1 coder.descriptor.DataImplementation]
                     Timing: [1×0 coder.descriptor.TimingInterface]
                       Unit: ''
                      Range: [1×1 coder.descriptor.Range]
         SupportTunableSize: 0
    BreakpointSpecification: 'Even spacing'
                     Output: [1×1 coder.descriptor.DataInterface]
                Breakpoints: [1×1 coder.descriptor.BreakpointDataInterface Sequence]
```

3. The `Breakpoints` property of the `coder.descriptor.LookupTableDataInterface` object holds a vector of `coder.descriptor.BreakpointDataInterface` objects. Obtain the details of the breakpoint set attached to the Lookup Table block by accessing the first location in the array.

```
params.Breakpoints(1)
```

```
  BreakpointDataInterface with properties:
                Type: [1×1 coder.descriptor.types.Type]
                 SID: 'demoModel:22'
       GraphicalName: 'n-D Lookup↵Table'
         VariantInfo: [1×0 coder.descriptor.VariantInfo]
      Implementation: [1×0 coder.descriptor.DataImplementation]
              Timing: [1×0 coder.descriptor.TimingInterface]
                Unit: ''
               Range: [1×1 coder.descriptor.Range]
      OperatingPoint: [1×1 coder.descriptor.DataInterface]
  SupportTunableSize: 0
     FixAxisMetadata: [1×1 coder.descriptor.FixAxisMetadata]
```

4. The new `coder.descriptor.FixAxisMetadata` object provides more information about whether the breakpoint set data is evenly spaced or not.

```
params.Breakpoints(1).FixAxisMetadata
```

The information is returned as a new `coder.descriptor.EvenSpacingMetadata` object that has these properties:

```
  EvenSpacingMetadata with properties:
    StartingValue: 2
        StepValue: 2
        NumPoints: 3
           IsPow2: 1
```

5. If the breakpoint set data value is changed to `[1 5 10]`, the information is returned as a new `coder.descriptor.NonEvenSpacingMetadata` object that has these properties:

```
  NonEvenSpacingMetadata with properties:
    AllPoints: [1×3 Real Sequence]
```

See Also

`coder.codedescriptor.CodeDescriptor` | `coder.descriptor.DataInterface` | `coder.descriptor.BreakpointDataInterface` | `coder.descriptor.EvenSpacingMetadata` | `coder.descriptor.NonEvenSpacingMetadata`

Introduced in R2020b

coder.descriptor.EvenSpacingMetadata class

Package: `coder.descriptor`
Superclasses: `coder.descriptor.FixAxisMetadata`

Return information about evenly spaced breakpoint set data

Description

The `coder.descriptor.EvenSpacingMetadata` object describes breakpoint set data that is evenly spaced, such as starting point, breakpoint step size, and number of points.

Creation

`params = getDataInterfaces(codeDescObj, dataInterfaceName)` creates a `coder.descriptor.BreakpointDataInterface` object for each dimension in the lookup table. The `codeDescObj` object is the `coder.codedescriptor.CodeDescriptor` object created for the model by using the `getCodeDescriptor` function.

The `coder.descriptor.BreakpointDataInterface` object has property `FixAxisMetadata` that contains a `coder.descriptor.FixAxisMetadata` object. The `coder.descriptor.FixAxisMetadata` contains a `coder.descriptor.EvenSpacingMetadata` object if the breakpoint set data is evenly spaced.

Input Arguments

`dataInterfaceName` — Name of data interface
`Parameters`

Specify the `Parameters` data interface type.

Example: `Parameters`

Properties

`StartingValue` — Starting value in the breakpoint set data
`character vector`

The first point in evenly spaced breakpoint set data.

`StepValue` — Spacing between evenly spaced breakpoints
`character vector`

The spacing between points in evenly spaced breakpoint set data. This value represents a power of `2` if the `IsPow2` returns `1`.

`NumPoints` — Total number of breakpoint values
`character vector`

Total number of points in evenly spaced breakpoint set data.

IsPow2 — Power of 2
1 | 0

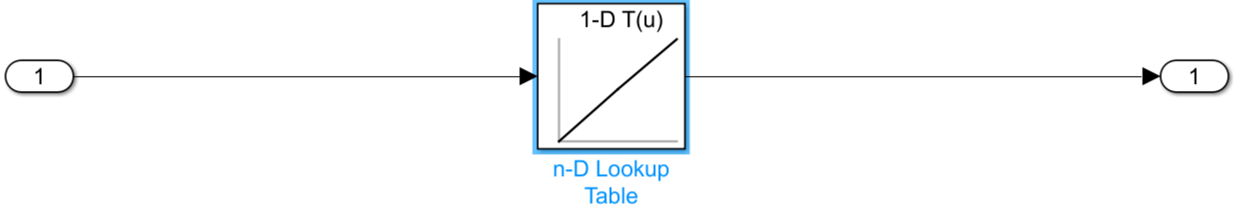
Returns 1 if the value in `StepValue` is a power of 2.

Data Types: `logical`

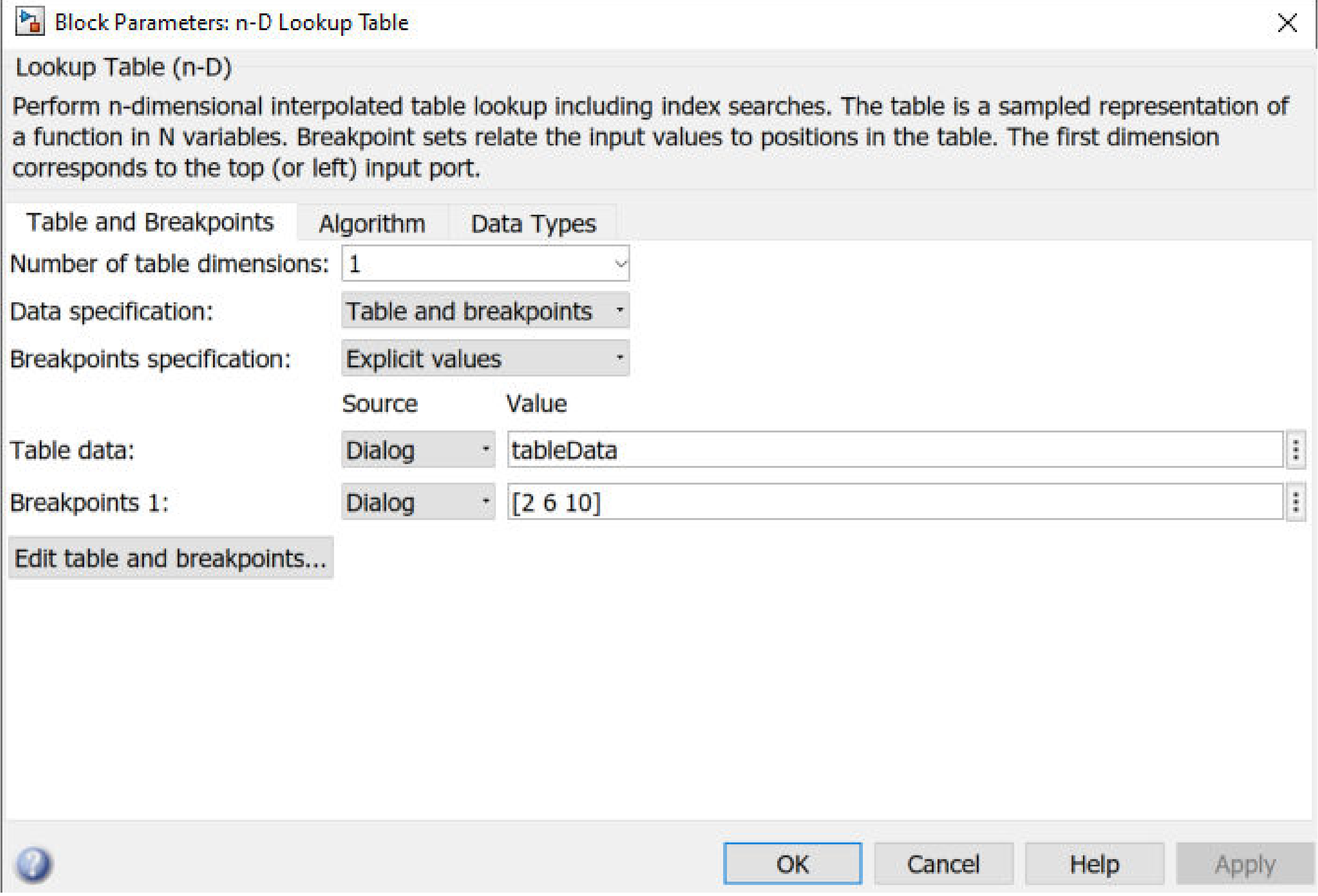
Examples

Get evenly spaced breakpoint set data information

Consider creating model `codeDescDemo` or a model with similar specifications.



The model contains a n-D Lookup Table. The n-D Lookup Table block takes table data from a model workspace variable named `tableData` with value `[4 5 6]`. The `tableData` is a `Simulink.Parameter` object with `non-Auto` storage class. The breakpoint set data is specified as `[2 6 10]`.



The model configuration parameter **Default parameter behavior** is set to `Inlined`.

1 Build the model and create a `coder.codedescriptor.CodeDescriptor` object for the model.

```
codeDescObj = coder.getCodeDescriptor('codeDescDemo')
```

2 Retrieve properties of the Lookup Table block and breakpoint set in the generated code.

```
params = getDataInterfaces(codeDescObj, 'Parameters')
```

The `params` variable is an array of `coder.descriptor.LookupTableDataInterface` and `coder.descriptor.BreakpointDataInterface` objects.

```
LookupTableDataInterface with properties:
                     Type: [1×1 coder.descriptor.types.Type]
                      SID: 'demoModel:22'
             GraphicalName: 'tableData'
              VariantInfo: [1×0 coder.descriptor.VariantInfo]
           Implementation: [1×1 coder.descriptor.DataImplementation]
                   Timing: [1×0 coder.descriptor.TimingInterface]
                     Unit: ''
                    Range: [1×1 coder.descriptor.Range]
       SupportTunableSize: 0
  BreakpointSpecification: 'Even spacing'
                   Output: [1×1 coder.descriptor.DataInterface]
              Breakpoints: [1×1 coder.descriptor.BreakpointDataInterface Sequence]
```

3 The `Breakpoints` property of the `coder.descriptor.LookupTableDataInterface` object holds a vector of `coder.descriptor.BreakpointDataInterface` objects. Obtain the details

of the breakpoint set attached to the Lookup Table block by accessing the first location in the array.

```
params.Breakpoints(1)
```

```
  BreakpointDataInterface with properties:
                Type: [1×1 coder.descriptor.types.Type]
                 SID: 'demoModel:22'
       GraphicalName: 'n-D Lookup↵Table'
         VariantInfo: [1×0 coder.descriptor.VariantInfo]
      Implementation: [1×0 coder.descriptor.DataImplementation]
              Timing: [1×0 coder.descriptor.TimingInterface]
                Unit: ''
               Range: [1×1 coder.descriptor.Range]
      OperatingPoint: [1×1 coder.descriptor.DataInterface]
   SupportTunableSize: 0
      FixAxisMetadata: [1×1 coder.descriptor.FixAxisMetadata]
```

4 The new `coder.descriptor.FixAxisMetadata` object provides more information about whether the breakpoint set data is evenly spaced or not.

```
params.Breakpoints(1).FixAxisMetadata
```

The information is returned as a new `coder.descriptor.EvenSpacingMetadata` object with these properties:

```
  EvenSpacingMetadata with properties:
    StartingValue: 2
        StepValue: 2
        NumPoints: 3
           IsPow2: 1
```

See Also

`coder.codedescriptor.CodeDescriptor` | `coder.descriptor.DataInterface` | `coder.descriptor.BreakpointDataInterface` | `coder.descriptor.NonEvenSpacingMetadata`

Introduced in R2020b

coder.descriptor.NonEvenSpacingMetadata class

Package: `coder.descriptor`
Superclasses: `coder.descriptor.FixAxisMetadata`

Return information about non-evenly spaced breakpoint set data

Description

The `coder.descriptor.NonEvenSpacingMetadata` object describes the points in breakpoint set data that are non-evenly spaced.

Creation

`params = getDataInterfaces(codeDescObj, dataInterfaceName)` creates a `coder.descriptor.BreakpointDataInterface` object for each dimension in the lookup table. The `codeDescObj` object is the `coder.codedescriptor.CodeDescriptor` object created for the model by using the `getCodeDescriptor` function.

The `coder.descriptor.BreakpointDataInterface` object has property `FixAxisMetadata` that contains a `coder.descriptor.FixAxisMetadata` object. The `coder.descriptor.FixAxisMetadata` further contains a `coder.descriptor.NonEvenSpacingMetadata` object if the breakpoint set data is non-evenly spaced.

Input Arguments

`dataInterfaceName` — Name of data interface
`Parameters`

Specify the `Parameters` data interface type.

Example: `Parameters`

Properties

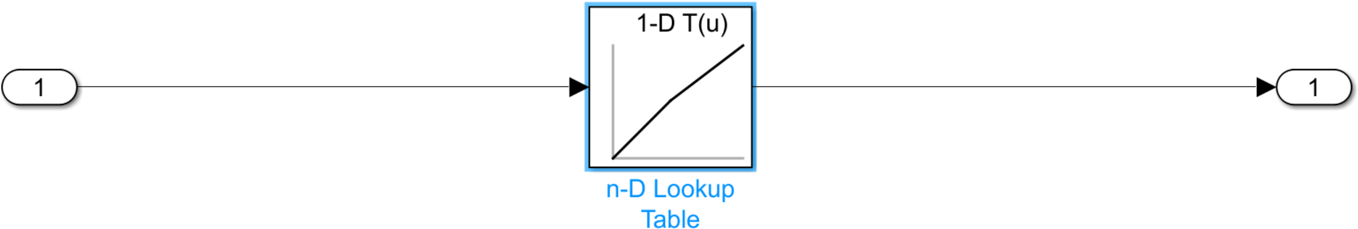
`AllPoints` — All values in the breakpoint set
`double vector`

All values that are in the non-evenly spaced breakpoint set data.

Examples

Get non-evenly spaced breakpoint set data information

Consider creating model `codeDescDemo` or a model with similar specifications.



The model contains an n-D Lookup Table. The n-D Lookup Table block takes table data from a model workspace variable named `tableData` that has a value of `[4 5 6]`. The `tableData` is a `Simulink.Parameter` object with a `non-Auto` storage class. The breakpoint set data is specified as `[1 5 10]`.

Block Parameters: n-D Lookup Table

Lookup Table (n-D)

Perform n-dimensional interpolated table lookup including index searches. The table is a sampled representation of a function in N variables. Breakpoint sets relate the input values to positions in the table. The first dimension corresponds to the top (or left) input port.

Table and Breakpoints | Algorithm | Data Types

Number of table dimensions: 1

Data specification: Table and breakpoints

Breakpoints specification: Explicit values

	Source	Value
Table data:	Dialog	tableData
Breakpoints 1:	Dialog	[1 5 10]

Edit table and breakpoints...

OK | Cancel | Help | Apply

The model configuration parameter **Default parameter behavior** is set to `Inlined`.

1. Build the model and create a `coder.codedescriptor.CodeDescriptor` object for the model.

```
codeDescObj = coder.getCodeDescriptor('codeDescDemo')
```

2. Retrieve properties of the Lookup Table block and breakpoint set in the generated code.

```
params = getDataInterfaces(codeDescObj, 'Parameters')
```

The `params` variable is an array of `coder.descriptor.LookupTableDataInterface` and `coder.descriptor.BreakpointDataInterface` objects.

```
  LookupTableDataInterface with properties:
                     Type: [1×1 coder.descriptor.types.Type]
                      SID: 'demoModel:22'
            GraphicalName: 'tableData'
              VariantInfo: [1×0 coder.descriptor.VariantInfo]
           Implementation: [1×1 coder.descriptor.DataImplementation]
                   Timing: [1×0 coder.descriptor.TimingInterface]
                     Unit: ''
                    Range: [1×1 coder.descriptor.Range]
       SupportTunableSize: 0
   BreakpointSpecification: 'Even spacing'
                   Output: [1×1 coder.descriptor.DataInterface]
              Breakpoints: [1×1 coder.descriptor.BreakpointDataInterface Sequence]
```

3. The `Breakpoints` property of the `coder.descriptor.LookupTableDataInterface` object holds a vector of `coder.descriptor.BreakpointDataInterface` objects. Obtain the details of the breakpoint set attached to the Lookup Table block by accessing the first location in the array.

```
params.Breakpoints(1)
```

```
  BreakpointDataInterface with properties:
                 Type: [1×1 coder.descriptor.types.Type]
                  SID: 'demoModel:22'
        GraphicalName: 'n-D Lookup↵Table'
          VariantInfo: [1×0 coder.descriptor.VariantInfo]
       Implementation: [1×0 coder.descriptor.DataImplementation]
               Timing: [1×0 coder.descriptor.TimingInterface]
                 Unit: ''
                Range: [1×1 coder.descriptor.Range]
       OperatingPoint: [1×1 coder.descriptor.DataInterface]
   SupportTunableSize: 0
      FixAxisMetadata: [1×1 coder.descriptor.FixAxisMetadata]
```

4. The new `coder.descriptor.FixAxisMetadata` object gives you more information about whether the breakpoint set data is evenly spaced or not.

```
params.Breakpoints(1).FixAxisMetadata
```

The information is returned as a new `coder.descriptor.NonEvenSpacingMetadata` object that has these properties:

```
  NonEvenSpacingMetadata with properties:
    AllPoints: [1×3 Real Sequence]
```

See Also

`coder.codedescriptor.CodeDescriptor` | `coder.descriptor.DataInterface` | `coder.descriptor.BreakpointDataInterface` | `coder.descriptor.EvenSpacingMetadata`

Introduced in R2020b

coder.report.close

Close HTML code generation report

Syntax

```
coder.report.close()
```

Description

`coder.report.close()` closes the HTML code generation report.

Examples

Close code generation report for a model

After opening a code generation report for `rtwdemo_counter`, close the report.

```
coder.report.close()
```

See Also

`coder.report.generate` | `coder.report.open`

Topics

"Reports for Code Generation"

Introduced in R2012a

coder.report.generate

Generate HTML code generation report

Syntax

```
coder.report.generate(model)
coder.report.generate(subsystem)
coder.report.generate(model,Name,Value)
```

Description

`coder.report.generate(model)` generates a code generation report for the `model`. The build folder for the model must be present in the current working folder.

`coder.report.generate(subsystem)` generates the code generation report for the `subsystem`. The build folder for the subsystem must be present in the current working folder.

`coder.report.generate(model,Name,Value)` generates the code generation report using the current model configuration and additional options specified by one or more `Name,Value` pair arguments. Possible values for the `Name,Value` arguments are parameters on the **Code Generation > Report** pane. Without modifying the model configuration, using the `Name,Value` arguments you can generate a report with a different report configuration.

Examples

Generate Code Generation Report for Model

Open the model `rtwdemo_counter`.

```
open rtwdemo_counter
```

Build the model. The model is configured to create and open a code generation report.

```
slbuild('rtwdemo_counter');
```

Close the code generation report.

```
coder.report.close;
```

Generate a code generation report.

```
coder.report.generate('rtwdemo_counter');
```

Generate Code Generation Report for Subsystem

Open the model `rtwdemo_counter`.

```
open rtwdemo_counter
```

Build the subsystem. The model is configured to create and open a code generation report.

```
slbuild('rtwdemo_counter/Amplifier');
```

Close the code generation report.

```
coder.report.close;
```

Generate a code generation report for the subsystem.

```
coder.report.generate('rtwdemo_counter/Amplifier');
```

Generate Code Generation Report to Include Static Code Metrics Report

Generate a code generation report to include a static code metrics report after the build process, without modifying the model.

Open the model `rtwdemo_hyperlinks`.

```
open rtwdemo_hyperlinks
```

Build the model. The model is configured to create and open a code generation report.

```
slbuild('rtwdemo_hyperlinks');
```

Close the code generation report.

```
coder.report.close;
```

Generate a code generation report that includes the static code metrics report.

```
coder.report.generate('rtwdemo_hyperlinks',...
'GenerateCodeMetricsReport','on');
```

The code generation report opens. In the left navigation pane, click **Static Code Metrics Report** to view the report.

Input Arguments

`model` — Model name
character vector

Model name specified as a character vector

Example: '`rtwdemo_counter`'

Data Types: `char`

`subsystem` — Subsystem name
character vector

Subsystem name specified as a character vector

Example: '`rtwdemo_counter/Amplifier`'

Data Types: `char`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Each `Name,Value` argument corresponds to a parameter on the Configuration Parameters **Code Generation > Report** pane. When the configuration parameter `GenerateReport` is `on`, the parameters are enabled. The `Name,Value` arguments are used only for generating the current report. The arguments will override, but not modify, the parameters in the model configuration. The following parameters require an Embedded Coder license.

Example: `'GenerateWebview','on','GenerateCodeMetricsReport','on'` includes a model Web view and static code metrics in the code generation report.

Navigation

`IncludeHyperlinkInReport` — Code-to-model hyperlinks
‘`off`’ | ‘`on`’

Code-to-model hyperlinks, specified as ‘`on`’ or ‘`off`’. Specify ‘`on`’ to include code-to-model hyperlinks in the code generation report. The hyperlinks link code to the corresponding blocks, Stateflow® objects, and MATLAB functions in the model diagram. For more information see “Code-to-model” on page 12-5.

Example: ‘`'IncludeHyperlinkInReport','on'`’

Data Types: `char`

`GenerateTraceInfo` — Model-to-code highlighting
‘`off`’ | ‘`on`’

Model-to-code highlighting, specified as ‘`on`’ or ‘`off`’. Specify ‘`on`’ to include model-to-code highlighting in the code generation report. For more information see “Model-to-code” on page 12-7.

Example: ‘`'GenerateTraceInfo','on'`’

Data Types: `char`

`GenerateWebview` — Model Web view
‘`off`’ | ‘`on`’

Model Web view, specified as ‘`on`’ or ‘`off`’. Specify ‘`on`’ to include the model Web view in the code generation report. For more information, see “Generate model Web view” on page 12-20.

Example: ‘`'GenerateWebview','on'`’

Data Types: `char`

Traceability Report Contents

`GenerateTraceReport` — Summary of eliminated and virtual blocks
‘`off`’ | ‘`on`’

Summary of eliminated and virtual blocks, specified as ‘`on`’ or ‘`off`’. Specify ‘`on`’ to include a summary of eliminated and virtual blocks in the code generation report. For more information, see “Eliminated / virtual blocks” on page 12-10.

Example: `'GenerateTraceReport','on'`

Data Types: `char`

`GenerateTraceReportSl` — Summary of Simulink blocks and the corresponding code location
'`off`' | '`on`'

Summary of the Simulink blocks and the corresponding code location, specified as '`on`' or '`off`'. Specify '`on`' to include a summary of the Simulink blocks and the corresponding code location in the code generation report. For more information, see "Traceable Simulink blocks" on page 12-12.

Example: `'GenerateTraceReportSl','on'`

Data Types: `char`

`GenerateTraceReportsSf` — Summary of Stateflow objects and the corresponding code location
'`off`' | '`on`'

Summary of the Stateflow objects and the corresponding code location, specified as '`on`' or '`off`'. Specify '`on`' to include a summary of Stateflow objects and the corresponding code location in the code generation report. For more information, see "Traceable Stateflow objects" on page 12-14.

Example: `'GenerateTraceReportSf','on'`

Data Types: `char`

`GenerateTraceReportEml` — Summary of MATLAB functions and the corresponding code location
'`off`' | '`on`'

Summary of the MATLAB functions and the corresponding code location, specified as '`on`' or '`off`'. Specify '`on`' to include a summary of the MATLAB objects and the corresponding code location in the code generation report. For more information, see "Traceable MATLAB functions" on page 12-16.

Example: `'GenerateTraceReportEml','on'`

Data Types: `char`

Metrics

`GenerateCodeMetricsReport` — Static code metrics
'`off`' | '`on`'

Static code metrics, specified as '`on`' or '`off`'. Specify '`on`' to include static code metrics in the code generation report. For more information, see "Generate static code metrics" on page 12-4.

Example: `'GenerateCodeMetricsReport','on'`

Data Types: `char`

See Also

`coder.report.close` | `coder.report.open`

Topics
"Reports for Code Generation"
"Generate a Code Generation Report"

"Generate Code Generation Report After Build Process"

Introduced in R2012a

coder.report.open

Open existing HTML code generation report

Syntax

```
coder.report.open(model)
coder.report.open(subsystem)
```

Description

`coder.report.open(model)` opens a code generation report for the `model`. The build folder for the model must be present in the current working folder.

`coder.report.open(subsystem)` opens a code generation report for the `subsystem`. The build folder for the subsystem must be present in the current working folder.

Examples

Open code generation report for a model

After generating code for `rtwdemo_counter`, open a code generation report for the model.

```
coder.report.open('rtwdemo_counter')
```

Open code generation report for a subsystem

Open a code generation report for the subsystem '`Amplifier`' in model '`rtwdemo_counter`'.

```
coder.report.open('rtwdemo_counter/Amplifier')
```

Input Arguments

`model` — Model name
character vector

Model name specified as a character vector

Example: '`rtwdemo_counter`'

Data Types: `char`

`subsystem` — Subsystem name
character vector

Subsystem name specified as a character vector

Example: '`rtwdemo_counter/Amplifier`'

Data Types: `char`

See Also

`coder.report.close` | `coder.report.generate`

Topics

"Reports for Code Generation"
"Open Code Generation Report"

Introduced in R2012a

coder.cdf.export

Generate CDF (Calibration Data Format) file according to ASAM AE CDF standards

Syntax

```
coder.cdf.export(modelName)
coder.cdf.export(modelName,Name,Value)
```

Description

`coder.cdf.export(modelName)` generates a CDF file for `modelName`.

`coder.cdf.export(modelName,Name,Value)` specifies additional options for CDF file creation with one or more Name, Value pair arguments. For example, you can specify a location where to save the CDF file.

Examples

Generate CDF File for Model

Generate a CDF file for the selected model and save it in the build folder of the model.

```
% Generate CDF file for model
coder.cdf.export('modelName')
```

Generate CDF File and Save it with a Custom Name

Generate a CDF file for the selected model and save it with the custom name specified.

```
% Export CDF file and save it as
coder.cdf.export('modelName','FileName','test_car')
```

Generate CDF File at Specified Location

Generate a CDF file for the selected model and save it in the specified folder.

```
% Export CDF file to specified path
coder.cdf.export('modelName','Folder','/home/temp/workspace/')
```

Generate CDF File with Specified Schema Type

Generate a CDF file of the specified schema type for the selected model.

```
% Export CDF file with dtd schema type
coder.cdf.export('modelName','SchemaType','DTD')
```

Input Arguments

`modelName` — name of the model
character vector | string scalar

Name of the model.

Example: `'MyModel','nav_app'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Folder','/home/applications'` generate a CDF file for the model and saves it in a specified folder.

`Folder` — Folder location for exported CDF file
character vector | string scalar

Full path to a folder in which to place an exported CDF file.

Example: `'Folder','/home/temp/prjct/'`

`FileName` — Custom name for the exported CDF file
character vector | string scalar

Name for the CDF file to save it as in the folder.

Example: `'FileName','test_car'`

`SchemaType` — Schema type for CDF file
DTD (default) | XSD

Schema type for the CDF file can be XSD (XML Schema Definition) or DTD (Document Type Definition).

Example: `'SchemaType','DTD'`

See Also
`coder.asap2.export`

Topics
"Generate ASAP2 and CDF Calibration Files"

Introduced in R2021a

coder.asap2.export

Generate ASAP2 (A2L) file according to ASAM MCD-2 MC standards

Syntax

```
coder.asap2.export(modelName)
coder.asap2.export(modelName,Name,Value)
```

Description

`coder.asap2.export(modelName)` generates an ASAP2 (A2L) file for `modelName`.

`coder.asap2.export(modelName,Name,Value)` specifies additional options for ASAP2 (A2L) creation with one or more Name, Value pair arguments. For example, you can specify a location where to save the A2L file. You can provide the symbol file of the model to replace ECU addresses in the A2L file.

Examples

Generate ASAP2 File for Model

Generate an A2L file for the selected model and save it in the build folder of the model.

```
% Generate A2L file for model
coder.asap2.export('modelName')
```

Generate A2L File and Save it with a Custom Name

Generate an A2L file for the selected model and save it with the custom name specified.

```
% Export A2L file and save it as
coder.asap2.export('modelName','FileName','test_car')
```

Generate ASAP2 File at Specified Location

Generate an A2L file for the selected model and save it in the specified folder.

```
% Export A2L file to specified path
coder.asap2.export('modelName','Folder','/home/temp/workspace/')
```

Generate ASAP2 File for Model by Using Symbol File

Generate an A2L file for the selected model with ECU addresses based on the ELF symbol file associated with the executable.

```
% Generate A2L file for model
coder.asap2.export('modelName','MapFile','model.elf')
```

Generate Specific Version of ASAP2 File for Model

Generate a specific version of the A2L file for the selected model. The description format of the data changes with respect to the version of the A2L file.

```
% Generate A2L file with version 1.71
coder.asap2.export('modelName','Version','1.71')
```

Exclude Comments from Generated A2L File

Generate an A2L file for the selected model and exclude the comments.

```
% Generate A2L file with comments excluded
coder.asap2.export('modelName','Comments',false)
```

Exclude A2ML and IF_DATA Sections from Generated A2L File

Generate an A2L file for the selected model and exclude the A2ML and IF_DATA sections.

```
% Generate A2L file with A2ML and IF_DATA excluded
coder.asap2.export('modelName','GenerateXCPInfo',false)
```

Generate ASAP2 File with Custom Model Class Instance Name

Specify the name of the model class instance. The objName is declared in the global namespace.

```
% Use custom specified name as object name in A2L file
coder.asap2.export('modelName','ModelClassInstanceName','objName')

% Specify the name of model class instance declared inside the namespace. Here instance customObj
% is declared in customNameSpace
coder.asap2.export('modelName','ModelClassInstanceName','customNamespace::customObj')
```

Generate ASAP2 File with Customized ASAP2 Fields

Create a custom base object and specify the fields. Customize the contents of the A2L file by using custom base object.

```
% Create custom base object and provide fields you want to modify
obj = coder.asap2.UserCustomizeBase;
obj.HeaderComment = 'Header comment';
obj.ModParComment = 'Mod Par comment';
obj.ModCommonComment = 'Mod Common comment';
obj.ASAP2FileName = 'File name';
obj.ByteOrder = 'BYTE_ORDER MSB_LAST';
```

```
% Generate A2L file with custom base created
coder.asap2.export('modelName','CustomizationObject',obj);
```

Input Arguments

modelName — name of the model

character vector | string scalar

Name of the model.

Example: 'MyModel','nav_app'

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'MapFile','model.elf' generates A2L file for the model with debug mapping information from the model.elf file.

Folder — Folder location to export A2L file

character vector | string scalar

Full path to a folder in which to place an exported A2L file.

Example: 'Folder','/home/temp/prjct/'

FileName — Custom name for the exported A2L file

character vector | string scalar

Name for the exported 2L file to save it in the folder.

Example: 'FileName','test_car'

MapFile — Name of symbol file for model

character vector | string scalar

Name of the model symbol file that contains symbols of generated code. For example, the addresses of variables used in generated code.

Example: 'MapFile','model.elf'

Version — Version of A2L file

1.71 (default) | 1.31 | 1.61

A2L file format based on ASAM MCD-2 MC standard defined by ASAM. There are multiple versions of the ASAM MCD-2 MC standard. Specify the version of A2L that you want.

Example: 'Version','1.61' or 'Version','1.31'

Comments — Include comments in A2L file

true (default) | false

Generate the A2L file by including or excluding comments.

Example: 'Comments',true

`GenerateXCPInfo` — Include A2ML and IF_DATA in A2L file
true (default) | false

Generate the A2L file by including or excluding A2ML and IF_DATA sections.

Example: `'GenerateXCPInfo',true`

`ModelClassInstanceName` — Specify class instance and path names
character vector | string scalar

Custom model instance name in an A2L file. This argument is applicable only for Adaptive AUTOSAR models.

Example: `'ModelClassInstanceName','customObj'` or `'ModelClassInstanceName','customNameSpace::customObj'`

`IndentFile` — Follow indentation in A2L file
false (default) | true

Generate an A2L file by following indentation.

Example: `'IndentFile',true`

`CustomizationObject` — Customize ASAP2 fields
`coder.asap2.UserCustomizeBase` object (default)

Create a user base and customize the ASAP2 fields such as:

- ASAP2FileName
- ByteOrder
- HeaderComment
- ModParComment
- ModCommonComment

Example: `'CustomizationObject',obj`

See Also

`coder.cdf.export`

Topics
"Generate ASAP2 and CDF Calibration Files"

Introduced in R2021a

extmodeBackgroundRun

Perform external mode background activity

Syntax

```
errorCode = extmodeBackgroundRun();
```

Description

`errorCode = extmodeBackgroundRun();` performs external mode background activity, for example, retrieving packets from the network, running the packets protocol layer, and sending packets to the development computer.

Do not invoke the function in a thread with real-time constraints.

Use this function with other external mode functions to enable communication between Simulink and the target application during an external mode simulation.

Examples

Set Up External Mode Communication

For a pseudo-code example that shows how you can provide external mode communication by using the function with related functions, see "External Mode Abstraction Layer".

Output Arguments

`errorCode` — Error detection
extmodeErrorCode_T enumeration

Error code, returned as an `extmodeErrorCode_T` enumeration with one of these values:

- `EXTMODE_SUCCESS` (0) –– No error detected.
- `EXTMODE_BUSY` (-6) –– Resource busy detected, try later
- `EXTMODE_INV_MSG_FORMAT` (-7) –– Invalid message format detected by external mode communication protocol.
- `EXTMODE_INV_SIZE` (-8) –– Invalid size detected by the external mode communication protocol.
- `EXTMODE_NOT_INITIALIZED` (-9) –– External mode not initialized yet.
- `EXTMODE_NO_MEMORY` (-10) –– No memory available on the target hardware.
- `EXTMODE_ERROR` (-12) –– External mode generic error detected.
- `EXTMODE_PKT_CHECKSUM_ERROR` (-13) –– Checksum inconsistency detected by external mode communication protocol.
- `EXTMODE_PKT_RX_TIMEOUT_ERROR` (-14) –– Timeout error detected during the reception of a packet.
- `EXTMODE_PKT_TX_TIMEOUT_ERROR` (-15) –– Timeout error detected during the transmission of a packet.

See Also

`extmodeEvent` | `extmodeGetFinalSimulationTime` | `extmodeInit` | `extmodeParseArgs` | `extmodeReset` | `extmodeSetFinalSimulationTime` | `extmodeSimulationComplete` | `extmodeStopRequested` | `extmodeWaitForHostRequest`

Topics

"External Mode Simulation by Using XCP Communication"
"Customize XCP Slave Software"

Introduced in R2018a

extmodeEvent

External mode event trigger

Syntax

`errorCode = extmodeEvent(eventId, simulationTime)`

Description

`errorCode = extmodeEvent(eventId, simulationTime)` informs the external mode abstraction layer of the occurrence of an event.

`eventId` is the sample time ID of the model, for example, 0 for base rate, 1 for first subrate, and so on.

The function:

- Samples all signals associated with a given sample time.
- Stores signal values in a new packet buffer.
- Passes the packet buffer to the underlying transport layer for subsequent transmission to the development computer.

Use this function with other external mode functions to enable communication between Simulink and the target application during an external mode simulation.

For correct sampling of signal values, run the function immediately after `model_step()` for the corresponding sample time ID. You can invoke the function with different sample time IDs in separate threads because the function is thread-safe.

The `extmodeBackgroundRun` function performs the transmission of signal values to the development computer.

Examples

Set Up External Mode Communication

For a pseudo-code example that shows how you can provide external mode communication by using the function with related functions, see “External Mode Abstraction Layer”.

Input Arguments

`eventId` — Event ID
uint16_T

Sample time ID of the model, which is 0 for base rate, 1 for first subrate, 2 for second subrate, and so on.

`simulationTime` — Simulation time
real_T

Time when event occurs.

Output Arguments

errorCode — Error detection
extmodeErrorCode_T enumeration

Error code, returned as an extmodeErrorCode_T enumeration with one of these values:

- EXTMODE_SUCCESS (0) –– No error detected.
- EXTMODE_INV_ARG (-1) –– Arguments invalid.
- EXTMODE_NOT_INITIALIZED (-9) –– External mode not initialized yet.
- EXTMODE_NO_MEMORY (-10) –– No memory available on the target hardware.

See Also

extmodeBackgroundRun | extmodeGetFinalSimulationTime | extmodeInit | extmodeParseArgs | extmodeReset | extmodeSetFinalSimulationTime | extmodeSimulationComplete | extmodeStopRequested | extmodeWaitForHostRequest

Topics
"External Mode Simulation by Using XCP Communication"
"Customize XCP Slave Software"

Introduced in R2018a

extmodeGetFinalSimulationTime

Get final simulation time for external mode platform abstraction layer

Syntax

```
errorCode = extmodeGetFinalSimulationTime(finalTime);
```

Description

`errorCode = extmodeGetFinalSimulationTime(finalTime);` gets the model's final simulation time for the external mode platform abstraction layer. The function is a complementary function for `extmodeSetFinalSimulationTime`.

Output Arguments

`finalTime` — Final simulation time
real_T pointer

Final simulation time of model.

`errorCode` — Error detection
extmodeErrorCode_T enumeration

Error code, returned as an `extmodeErrorCode_T` enumeration with one of these values:

- `EXTMODE_SUCCESS` (0) –– No error detected.
- `EXTMODE_INV_ARG` (-1) –– Arguments invalid.
- `EXTMODE_NOT_INITIALIZED` (-9) –– External mode not initialized yet.

See Also

`extmodeBackgroundRun` | `extmodeEvent` | `extmodeInit` | `extmodeParseArgs` | `extmodeReset` | `extmodeSetFinalSimulationTime` | `extmodeSimulationComplete` | `extmodeStopRequested` | `extmodeWaitForHostRequest`

Topics
"External Mode Simulation by Using XCP Communication"
"Customize XCP Slave Software"

Introduced in R2018a

extmodeInit

Initialize external mode target connectivity

Syntax

```
errorCode = extmodeInit(extmodeInfo, finalTime);
```

Description

`errorCode = extmodeInit(extmodeInfo, finalTime);` initializes the external mode target connectivity, including the underlying communication stack.

Use this function with other external mode functions to enable communication between Simulink and the target application during an external mode simulation.

Examples

Set Up External Mode Communication

For a pseudo-code example that shows how you can provide external mode communication by using the function with related functions, see "External Mode Abstraction Layer".

Input Arguments

`extmodeInfo` — External mode information structure
RTWExtModeInfo structure

Model structure that contains information for the external mode simulation. `RTWExtModeInfo` is defined in *`matlabroot`*`/simulink/include/rtw_extmode.h`.

`finalTime` — Final simulation time
real_T pointer

If the model's final simulation time in the external mode abstraction layer is initialized, then `finalTime` is an output and the pointer location is updated with the initialized value. You might initialize the final simulation time through the `'-tf '` option detected by `extmodeParseArgs()` or `extmodeSetFinalSimulationTime()`

If the model's final simulation time in the external mode abstraction layer is not initialized, then `finalTime` is an input and the model's final simulation time in external mode is updated accordingly.

Output Arguments

`errorCode` — Error detection
extmodeErrorCode_T enumeration

Error code, returned as an `extmodeErrorCode_T` enumeration with one of these values:

- `EXTMODE_SUCCESS` (0) –– No error detected.
- `EXTMODE_INV_ARG` (-1) –– Arguments invalid.
- `EXTMODE_ERROR` (-12) –– External mode generic error detected.

See Also

`extmodeBackgroundRun` | `extmodeEvent` | `extmodeGetFinalSimulationTime` | `extmodeParseArgs` | `extmodeReset` | `extmodeSetFinalSimulationTime` | `extmodeSimulationComplete` | `extmodeStopRequested` | `extmodeWaitForHostRequest`

Topics

"External Mode Simulation by Using XCP Communication"
"Customize XCP Slave Software"

Introduced in R2018a

extmodeParseArgs

Extract values of configuration parameters supported by external mode abstraction layer

Syntax

```
errorCode = extmodeParseArgs(argCount, argValues);
```

Description

`errorCode = extmodeParseArgs(argCount, argValues);` extracts the values of the configuration parameters that are supported by the external mode abstraction layer. The function parses the array of strings passed as input arguments. The array of strings is from the command-line arguments of the executable file running on the target hardware.

The external mode abstraction layer interprets only two options and passes the other arguments to `rtIOStreamOpen` for the initialization of the communication driver.

Use this function with other external mode functions to enable communication between Simulink and the target application during an external mode simulation.

If your target hardware does not support the parsing of command-line arguments, define the preprocessor macro `EXTMODE_DISABLE_ARGS_PROCESSING`. See information about parsing command-line arguments in "Other Platform Abstraction Layer Functionality".

Examples

Set Up External Mode Communication

For a pseudo-code example that shows how you can provide external mode communication by using the function with related functions, see "External Mode Abstraction Layer".

Input Arguments

`argCount` — Number of arguments
int_T scalar

Number of elements in `argValues` array.

`argValues` — Command-line arguments
array of null-terminated strings

Command-line arguments of the executable file running on the target hardware. The external mode abstraction layer interprets only these options:

- `'-w'` – Enables the `extmodeWaitForStartRequest()` function, which waits for a model start request from Simulink in external mode. If you do not specify this option, the `extmodeWaitForStartRequest()` function has no effect.
- `'-tf` *`finalSimulationTime`*`'` – *`finalSimulationTime`* overrides the Simulink configuration parameter, `StopTime`.

If the command contains more options, they are passed to `rtIOStreamOpen` as configuration parameters for the communication driver.

Output Arguments

`errorCode` — Error detection
extmodeErrorCode_T enumeration

Error code, returned as an `extmodeErrorCode_T` enumeration with one of these values:

- `EXTMODE_SUCCESS` (0) –– No error detected.
- `EXTMODE_INV_ARG` (-1) –– Arguments invalid.

See Also

`extmodeBackgroundRun` | `extmodeEvent` | `extmodeGetFinalSimulationTime` | `extmodeInit` | `extmodeReset` | `extmodeSetFinalSimulationTime` | `extmodeSimulationComplete` | `extmodeStopRequested` | `extmodeWaitForHostRequest`

Topics
"External Mode Simulation by Using XCP Communication"
"Customize XCP Slave Software"

Introduced in R2018a

extmodeReset

Reset external mode target connectivity

Syntax

```
errorCode = extmodeReset();
```

Description

`errorCode = extmodeReset();` restores the external mode abstraction layer, including the communication stack, to the initial, default state.

Use this function with other external mode functions to enable communication between Simulink and the target application during an external mode simulation.

Examples

Set Up External Mode Communication

For a pseudo-code example that shows how you can provide external mode communication by using the function with related functions, see “External Mode Abstraction Layer”.

Output Arguments

`errorCode` — Error detection
extmodeErrorCode_T enumeration

Error code, returned as an `extmodeErrorCode_T` enumeration with one of these values:

- `EXTMODE_SUCCESS` (0) –– No error detected.
- `EXTMODE_ERROR` (-12) –– External mode generic error detected.

See Also

`extmodeBackgroundRun` | `extmodeEvent` | `extmodeGetFinalSimulationTime` | `extmodeInit` | `extmodeParseArgs` | `extmodeSetFinalSimulationTime` | `extmodeSimulationComplete` | `extmodeStopRequested` | `extmodeWaitForHostRequest`

Topics

“External Mode Simulation by Using XCP Communication”
“Customize XCP Slave Software”

Introduced in R2018a

extmodeSetFinalSimulationTime

Set final simulation time in external mode platform abstraction layer

Syntax

```
errorCode = extmodeSetFinalSimulationTime(finalTime);
```

Description

`errorCode = extmodeSetFinalSimulationTime(finalTime);` sets the final simulation time of the model in the external mode platform abstraction layer.

In the `main` function of your external mode target application, before `extmodeInit`, you can call `extmodeSetFinalSimulationTime` to set the final simulation time if:

- You do not want to use `extmodeParseArgs`.
- Your target hardware does not support parsing of command-line arguments but you want to override `StopTime` from the target application.

`extmodeGetFinalSimulationTime` and `extmodeSetFinalSimulationTime` are complementary functions.

Input Arguments

`finalTime` — Final simulation time
real_T

Final simulation time of model.

Output Arguments

`errorCode` — Error detection
extmodeErrorCode_T enumeration

Error code, returned as an `extmodeErrorCode_T` enumeration with one of these values:

- `EXTMODE_SUCCESS` (0) –– No error detected.
- `EXTMODE_INV_ARG` (-1) –– Arguments invalid.

See Also

`extmodeBackgroundRun` | `extmodeEvent` | `extmodeGetFinalSimulationTime` | `extmodeInit` | `extmodeParseArgs` | `extmodeReset` | `extmodeSimulationComplete` | `extmodeStopRequested` | `extmodeWaitForHostRequest`

Topics
"External Mode Simulation by Using XCP Communication"
"Customize XCP Slave Software"

Introduced in R2018a

extmodeSimulationComplete

Check that external mode simulation is complete

Syntax

```
simComplete = extmodeSimulationComplete();
```

Description

`simComplete = extmodeSimulationComplete();` during an external mode simulation, checks whether the model simulation time has reached the final simulation time specified by the command-line `'-tf'` option or the Simulink configuration parameter, `StopTime`.

Use this function with other external mode functions to enable communication between Simulink and the target application during an external mode simulation.

Examples

Set Up External Mode Communication

For a pseudo-code example that shows how you can provide external mode communication by using the function with related functions, see “External Mode Abstraction Layer”.

Output Arguments

`simComplete` — Simulation complete
true | false

`true` if model simulation time has reached the specified final simulation time. Otherwise, returns `false`.

See Also

`extmodeBackgroundRun` | `extmodeEvent` | `extmodeGetFinalSimulationTime` | `extmodeInit` | `extmodeParseArgs` | `extmodeReset` | `extmodeSetFinalSimulationTime` | `extmodeStopRequested` | `extmodeWaitForHostRequest`

Topics
“External Mode Simulation by Using XCP Communication”
“Customize XCP Slave Software”

Introduced in R2018a

extmodeStopRequested

Check whether request to stop external mode simulation is received from model

Syntax

```
stopRequest = extmodeStopRequested();
```

Description

`stopRequest = extmodeStopRequested();` checks whether a request to stop the external mode simulation is received from the Simulink model on the development computer.

Use this function with other external mode functions to enable communication between Simulink and the target application during an external mode simulation.

Examples

Set Up External Mode Communication

For a pseudo-code example that shows how you can provide external mode communication by using the function with related functions, see "External Mode Abstraction Layer".

Output Arguments

`stopRequest` — Stop request
true | false

`true` if request to stop external mode simulation is received. Otherwise, returns `false`.

See Also

`extmodeBackgroundRun` | `extmodeEvent` | `extmodeGetFinalSimulationTime` | `extmodeInit` | `extmodeParseArgs` | `extmodeReset` | `extmodeSetFinalSimulationTime` | `extmodeSimulationComplete` | `extmodeWaitForHostRequest`

Topics
"External Mode Simulation by Using XCP Communication"
"Customize XCP Slave Software"

Introduced in R2018a

extmodeWaitForHostRequest

Wait for request from development computer to start or stop external mode simulation

Syntax

```
errorCode = extmodeWaitForHostRequest(timeoutInMicroseconds);
```

Description

`errorCode = extmodeWaitForHostRequest(timeoutInMicroseconds);` waits for a start or stop request from the development computer and times out when the timeout value is reached.

Use this function with other external mode functions to enable communication between Simulink and the target application during an external mode simulation. Use the function during initialization because the function is a blocking function.

Examples

Set Up External Mode Communication

For a pseudo-code example that shows how you can provide external mode communication by using the function with related functions, see "External Mode Abstraction Layer".

Input Arguments

`timeoutInMicroseconds` — Timeout
uint32_T

Specifies the timeout value. If the value is set to `EXTMODE_WAIT_FOREVER`, the function waits indefinitely. If `'-w'` is not extracted by `extmodeParseArgs()`, the function has no effect.

Output Arguments

`errorCode` — Error detection
extmodeErrorCode_T enumeration

Error code, returned as an `extmodeErrorCode_T` enumeration with one of these values:

- `EXTMODE_SUCCESS` (0) –– No error detected.
- `EXTMODE_INV_ARG` (-1) –– Arguments invalid.
- `EXTMODE_TIMEOUT_ERROR` (-100) –– External mode timeout error detected.

See Also

`extmodeBackgroundRun` | `extmodeEvent` | `extmodeGetFinalSimulationTime` | `extmodeInit` | `extmodeParseArgs` | `extmodeReset` | `extmodeSetFinalSimulationTime` | `extmodeSimulationComplete` | `extmodeStopRequested`

Topics
"External Mode Simulation by Using XCP Communication"
"Customize XCP Slave Software"

Introduced in R2018a

findBuildArg

Find a specific build argument in build information

Syntax

```
[identifier,value] = findBuildArg(buildinfo,buildArgName)
```

Description

`[identifier,value] = findBuildArg(buildinfo,buildArgName)` searches for a build argument from the build information.

If the build argument is present in the build information, the function returns the name and value.

Examples

Find Build Argument in Build Information

Find a build argument and its value stored in build information `myBuildInfo`. Then, view the argument identifier and value.

```
load buildInfo.mat
myBuildInfo = buildInfo;
myBuildArgExtmodeStaticAlloc = 'EXTMODE_STATIC_ALLOC';
[buildArgId buildArgValue] = findBuildArg(buildInfo, ...
   myBuildArgExtmodeStaticAlloc);
```

```
>> buildArgId

buildArgId =

    'EXTMODE_STATIC_ALLOC'

>> buildArgValue

buildArgValue =

    '0'
```

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo`
object

`buildArgName` — Name of build argument to find in build information
character vector | string scalar

To get the build argument identifiers from the build information, use the `getBuildArgs` function.

Output Arguments

`identifier` — Name of the build argument
character vector | string scalar

`value` — Value of the build argument
character vector | string scalar

See Also
`getBuildArgs`

Topics
"Customize Post-Code-Generation Build Processing"

Introduced in R2014a

findIncludeFiles

Find and add include (header) files to build information

Syntax

`findIncludeFiles(buildinfo,extPatterns)`

Description

`findIncludeFiles(buildinfo,extPatterns)` searches for and adds include files to the build information.

Use the `findIncludeFiles` function to:

- Search for include files in source and include paths from the build information.
- Apply the optional *`extPatterns`* argument to specify file name extension patterns for search.
- Add the found files with their full paths to the build information.
- Delete duplicate include file entries from the build information.

To ensure that `findIncludeFiles` finds header files, add their paths to `buildInfo` by using the `addIncludePaths` function.

Examples

Find and Add Include Files to Build Information

Find include files with file name extension `.h` that are in the build information, `myBuildInfo`. Add the full paths for these files to the build information. View the include files from the build information.

```
myBuildInfo = RTW.BuildInfo;
addSourcePaths(myBuildInfo,{fullfile(pwd,...
   'mycustomheaders')},'myheaders');
findIncludeFiles(myBuildInfo);
headerfiles = getIncludeFiles(myBuildInfo,true,false);
```

```
>> headerfiles

headerfiles =

    'W:\work\mycustomheaders\myheader.h'
```

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo`
object

Object provides information for compiling and linking generated code.

`extPatterns` — Patterns of file name extensions that specify files for the search
`'*.h'` (default) | cell array of character vectors | string array

To specify files for the search, the character vectors or strings in the *`extPatterns`* argument:

- Must start with an asterisk immediately followed by a period (*.)
- Can include a combination of alphanumeric and underscore (_) characters

Example: `'*.h' '*.hpp' '*.x*'`

See Also

`addIncludeFiles` | `getIncludeFiles` | `packNGo`

Topics

"Customize Post-Code-Generation Build Processing"

Introduced in R2006b

getBuildArgs

Get build arguments from build information

Syntax

```
[identifiers,values] = getBuildArgs(buildinfo,includeGroupIDs,
excludeGroupIDs)
```

Description

`[identifiers,values] = getBuildArgs(buildinfo,includeGroupIDs,
excludeGroupIDs)` returns build argument identifiers and values from build information.

The function requires the *`buildinfo`*, *`identifiers`*, and *`values`* arguments. You can use optional *`includeGroupIDs`* and *`excludeGroupIDs`* arguments. These optional arguments let you include or exclude groups selectively from the build arguments returned by the function.

If you choose to specify *`excludeGroupIDs`* and omit *`includeGroupIDs`*, specify a null character vector (`''`) for *`includeGroupIDs`*.

Examples

Get Build Arguments from Build Information

After you build a , the build information is available in the `buildInfo.mat` file. Retrieve the build arguments from the build information object.

```
load buildInfo.mat
[buildArgIds,buildArgValues] = getBuildArgs(buildInfo);
```

To get the value of a single build argument from the build information, you can use the `findBuildArg` function.

To view the build argument identifiers, enter:

```
buildArgIds
```

To view the build argument values, enter:

```
buildArgValues
```

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo`
object

`includeGroupIDs` — Group identifiers of build arguments to include in the return from the function
cell array of character vectors | string

To use the *includeGroupIDs* argument, view available build argument identifier groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

`excludeGroupIDs` — Group identifiers of build arguments to exclude from the return from the function
cell array of character vectors | string

To use the *excludeGroupIDs* argument, view available build argument identifier groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

Output Arguments

`identifiers` — Names of the build arguments
cell array of character vectors

`values` — Values of the build arguments
cell array of character vectors

See Also

`findBuildArg`

Topics

"Customize Post-Code-Generation Build Processing"

Introduced in R2014a

getCodeDescriptor

Create `coder.codedescriptor.CodeDescriptor` object for model

Syntax

```
getCodeDescriptor(model)
getCodeDescriptor(folder)
```

Description

`getCodeDescriptor(model)` creates a `coder.codedescriptor.CodeDescriptor` object for the specified model.

`getCodeDescriptor(folder)` creates a `coder.codedescriptor.CodeDescriptor` object for the specified build folder.

Examples

Create a Code Descriptor Object Using Model Name

Create a `coder.codedescriptor.CodeDescriptor` object by using model name:

```
codeDescObj = coder.getCodeDescriptor('rtwdemo_comments')
```

Create a Code Descriptor Object Using Build Folder

Create a `coder.codedescriptor.CodeDescriptor` object by using build folder:

```
codeDescObj = coder.getCodeDescriptor('C:\Users\Desktop\work\rtwdemo_comments_ert_rtw')
```

Input Arguments

`model` — Name of the model
string

Model object or name for which to obtain the `coder.codedescriptor.CodeDescriptor` object. You can get the `coder.codedescriptor.CodeDescriptor` object only for the top model if the model has referenced models.

Example: `rtwdemo_comments`

Data Types: `string`

`folder` — Build folder of the model
string

Build folder of the model for which to obtain the `coder.codedescriptor.CodeDescriptor` object. You can get the `coder.codedescriptor.CodeDescriptor` object only for the top model if the model has referenced models.

Example: `C:\Users\Desktop\Work\rtwdemo_comments_ert_rtw`

Data Types: `string`

See Also
`coder.codedescriptor.CodeDescriptor`

Topics
"Get Code Description of Generated Code"

Introduced in R2018a

getCompileFlags

Get compiler options from build information

Syntax

`options = getCompileFlags(buildinfo,includeGroups,excludeGroups)`

Description

`options = getCompileFlags(buildinfo,includeGroups,excludeGroups)` returns compiler options from the build information.

The function requires the *`buildinfo`* argument. You can use optional *`includeGroups`* and *`excludeGroups`* arguments. These optional arguments let you include or exclude groups selectively from the compiler options returned by the function.

If you choose to specify *`excludeGroups`* and omit *`includeGroups`*, specify a null character vector (`''`) for *`includeGroups`*.

Examples

Get Compiler Options from Build Information

Get the compiler options stored in the build information `myBuildInfo`.

```
myBuildInfo = RTW.BuildInfo;
addCompileFlags(myBuildInfo,{'-Zi -Wall' '-O3'}, ...
   'OPTS');
compflags = getCompileFlags(myBuildInfo);
```

```
>> compflags

compflags =

    '-Zi -Wall'    '-O3'
```

Get Compiler Options with Include Group Argument

Get the compiler options stored with the group name `Debug` in the build information `myBuildInfo`.

```
myBuildInfo = RTW.BuildInfo;
addCompileFlags(myBuildInfo,{'-Zi -Wall' '-O3'}, ...
   {'Debug' 'MemOpt'});
compflags = getCompileFlags(myBuildInfo,'Debug');
```

```
>> compflags
```

```
compflags =

    '-Zi -Wall'
```

Get Compiler Options with Exclude Group Argument

Get the compiler options stored in the build information `myBuildInfo`, except those options with the group name `Debug`.

```
myBuildInfo = RTW.BuildInfo;
addCompileFlags(myBuildInfo,{'-Zi -Wall' '-O3'}, ...
   {'Debug' 'MemOpt'});
compflags = getCompileFlags(myBuildInfo,'','Debug');
```

```
>> compflags

compflags =

    '-O3'
```

Input Arguments

`buildinfo` — Name of the build information object returned by `RTW.BuildInfo`
object

`includeGroups` — Group names of compiler options to include in the return from the function
cell array of character vectors | string

To use the *`includeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

`excludeGroups` — Group names of compiler options to exclude from the return from the function
cell array of character vectors | string

To use the *`excludeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

Output Arguments

`options` — Compiler options from the build information
cell array of character vectors

See Also
`addCompileFlags` | `getDefines` | `getLinkFlags`

Topics
"Customize Post-Code-Generation Build Processing"

Introduced in R2006a

getDefines

Get preprocessor macro definitions from build information

Syntax

`[macrodefs,identifiers,values] = getDefines(buildinfo,includeGroups, excludeGroups)`

Description

`[macrodefs,identifiers,values] = getDefines(buildinfo,includeGroups, excludeGroups)` returns preprocessor macro definitions from the build information.

The function requires the *`buildinfo`*, *`macrodefs`*, *`identifiers`*, and *`values`* arguments. You can use optional *`includeGroups`* and *`excludeGroups`* arguments. These optional arguments let you include or exclude groups selectively from the preprocessor macro definitions returned by the function.

If you choose to specify *`excludeGroups`* and omit *`includeGroups`*, specify a null character vector (`''`) for *`includeGroups`*.

Examples

Get Macro Definitions from Build Information

Get the preprocessor macro definitions stored in the build information `myBuildInfo`.

```
myBuildInfo = RTW.BuildInfo;
addDefines(myBuildInfo, ...
   {'PROTO=first' '-DDEBUG' 'test' '-dPRODUCTION'},'OPTS');
[defs,names,values] = getDefines(myBuildInfo);
```

```
>> defs

defs =

    '-DPROTO=first'    '-DDEBUG'    '-Dtest'    '-DPRODUCTION'

>> names

names =

    'PROTO'
    'DEBUG'
    'test'
    'PRODUCTION'

>> values

values =
```

```
    'first'
    ''
    ''
    ''
```

Get Macro Definitions with Include Group Argument

Get the preprocessor macro definitions stored with the group name `Debug` in the build information `myBuildInfo`.

```
myBuildInfo = RTW.BuildInfo;
addDefines(myBuildInfo, ...
   {'PROTO=first' '-DDEBUG' 'test' '-dPRODUCTION'}, ...
   {'Debug' 'Debug' 'Debug' 'Release'});
[defs,names,values] = getDefines(myBuildInfo,'Debug');
```

```
>> defs

defs =

    '-DPROTO=first'    '-DDEBUG'     '-Dtest'
```

Get Macro Definitions with Exclude Group Argument

Get the preprocessor macro definitions stored in the build information `myBuildInfo`, except those definitions with the group name `Debug`.

```
myBuildInfo = RTW.BuildInfo;
addDefines(myBuildInfo, ...
   {'PROTO=first' '-DDEBUG' 'test' '-dPRODUCTION'}, ...
   {'Debug' 'Debug' 'Debug' 'Release'});
[defs,names,values] = getDefines(myBuildInfo,'','Debug');
```

```
>> defs

defs =

    '-DPRODUCTION'
```

Input Arguments

`buildinfo` — Name of the build information object returned by `RTW.BuildInfo`
object

`includeGroups` — Group names of macro definitions to include in the return from the function
cell array of character vectors | string

To use the *`includeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

`excludeGroups` — Group names of macro definitions to exclude from the return from the function
cell array of character vectors | string

To use the *`excludeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

Output Arguments

`macrodefs` — Macro definitions from the build information
cell array of character vectors

The *`macrodefs`* provide the complete macro definitions with a `-D` prefix. When the function returns a definition:

- If the `-D` was not specified when the definition was added to the build information, prepends a `-D` to the definition.
- Changes a lowercase `-d` to `-D`.

`identifiers` — Names of the macros from the build information
cell array of character vectors

`values` — Values assigned to the macros from the build information
cell array of character vectors

The *`values`* provide anything specified to the right of the first equal sign in the macro definition. The default is an empty character vector (`''`).

See Also
`addDefines` | `getCompileFlags` | `getLinkFlags`

Topics
"Customize Post-Code-Generation Build Processing"

Introduced in R2006a

getFullFileList

Get list of files from build information

Syntax

`[fPathNames,names] = getFullFileList(buildinfo,fcase)`

Description

`[fPathNames,names] = getFullFileList(buildinfo,fcase)` returns the fully qualified paths and names of files, or files of a selected type (source, include, or nonbuild), from the build information.

The function requires the *`buildinfo`*, *`fPathNames`*, and *`names`* arguments. You can use the optional *`fcase`* argument. This optional argument lets you include or exclude file cases selectively from file list returned by the function.

To ensure that header files are added to the file list (for example, header files in the _sharedutils folder), run `findIncludeFiles` before `getFullFileList`.

The `packNGo` function calls `getFullFileList` to return a list of files in the build information before processing files for packaging.

The makefile for the build resolves file locations based on source paths and rules. The build process does not require you to resolve the path of every file in the build information. The `getFullFileList` function returns the path for each file:

- If a path was explicitly associated with the file when it was added.
- If you called `updateFilePathsAndExtensions` to resolve file paths and extensions before calling `getFullFileList`.

Examples

Get Full List of Files

After building a and loading the generated `buildInfo.mat` file, you can list the files stored in the build information object, `buildInfo`. This example returns information for the current and its subs.

From the code generation folder that contains the `buildInfo.mat` file, run:

```
bi = load('buildInfo.mat');
findIncludeFiles(bi.buildInfo);
[fPathNames, names] = getFullFileList(bi.buildInfo);
```

Get List of Source Files

If you use an fcase option, you limit the listing to the files stored in the build information object for the current . This example returns information for the current only (not for subs).

```
[fPathNames,names] = getFullFileList(bi.buildInfo,'source');
```

Input Arguments

buildinfo — Name of the build information object returned by RTW.BuildInfo
object

fcase — File case to return from the build information
'' (default) | 'source' | 'include' | 'nonbuild'

The *fcase* argument selects whether the function returns the full list for files in the build information or returns selected cases of files. If you omit the argument or specify a null character vector (''), the function returns the files from the build information.

Specify	Function Action
'source'	Returns source files from the build information.
'include'	Returns include files from the build information.
'nonbuild'	Returns nonbuild files from the build information.

Example: 'source'

Output Arguments

fPathNames — Fully qualified file paths from the build information
cell array of character vectors

names — File names from the build information
cell array of character vectors

See Also

findIncludeFiles

Topics

"Customize Post-Code-Generation Build Processing"

Introduced in R2008a

getIncludeFiles

Get include files from build information

Syntax

```
files = getIncludeFiles(buildinfo,concatenatePaths,replaceMatlabroot,
includeGroups,excludeGroups)
```

Description

`files = getIncludeFiles(buildinfo,concatenatePaths,replaceMatlabroot, includeGroups,excludeGroups)` returns the names of include files from the build information.

The function requires the *`buildinfo`*, *`concatenatePaths`*, and *`replaceMatlabroot`* arguments. You can use optional *`includeGroups`* and *`excludeGroups`* arguments. These optional arguments let you include or exclude groups selectively from the include files returned by the function.

If you choose to specify *`excludeGroups`* and omit *`includeGroups`*, specify a null character vector (`''`) for *`includeGroups`*.

The makefile for the build resolves file locations based on source paths and rules. The build process does not require you to resolve the path of every file in the build information. If you specify `true` for the *`concatenatePaths`* argument, the `getIncludeFiles` function returns the path for each file:

- If a path was explicitly associated with the file when it was added.
- If you called `updateFilePathsAndExtensions` to resolve file paths and extensions before calling `getIncludeFiles`.

Examples

Get Include Paths and Files from Build Information

Get the include paths and file names from the build information, `myBuildInfo`.

```
myBuildInfo = RTW.BuildInfo;
addIncludeFiles(myBuildInfo,{'etc.h' 'etc_private.h' ...
   'mytypes.h'},{'/etc/proj/etclib' '/etcproj/etc/etc_build' ...
   '/common/lib'},{'etc' 'etc' 'shared'});
incfiles=getIncludeFiles(myBuildInfo,true,false);
```

```
>> incfiles

incfiles = 

    [1x22 char]    [1x36 char]    [1x21 char]
```

Get Include Paths and Files with Include Group Argument

Get the names of include files in group `etc` from the build information, `myBuildInfo`.

```
myBuildInfo = RTW.BuildInfo;
addIncludeFiles(myBuildInfo,{'etc.h' 'etc_private.h' ...
   'mytypes.h'},{'/etc/proj/etclib' '/etcproj/etc/etc_build' ...
   '/common/lib'},{'etc' 'etc' 'shared'});
incfiles = getIncludeFiles(myBuildInfo,false,false, ...
   'etc');
```

```
>> incfiles

incfiles =

    'etc.h'     'etc_private.h'
```

Input Arguments

`buildinfo` — Name of the build information object returned by `RTW.BuildInfo`
object

`concatenatePaths` — Choice of whether to concatenate paths and file names in return
`false` | `true`

Specify	Function Action
`true`	Concatenates and returns each file name with its corresponding path.
`false`	Returns only file names.

Example: `true`

`replaceMatlabroot` — Choice of whether to replace the `$(MATLAB_ROOT)` token with absolute paths in return
`false` | `true`

Use the *`replaceMatlabroot`* argument to control whether the function includes the MATLAB root definition in the output it returns.

Specify	Function Action
`true`	Replaces the token `$(MATLAB_ROOT)` with the absolute path for your MATLAB installation folder.
`false`	Does not replace the token `$(MATLAB_ROOT)`.

Example: `true`

`includeGroups` — Group names of include paths and files to include in the return from the function
cell array of character vectors | string

To use the *`includeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

`excludeGroups` — Group names of include paths and files to exclude from the return from the function
cell array of character vectors | string

To use the *`excludeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

Output Arguments

`files` — Names of include files from the build information
cell array of character vectors

The names of include files that you add with the `addIncludeFiles` function. If you call the `packNGo` function, the names include files that `packNGo` found and added while packaging code.

See Also

`addIncludeFiles` | `findIncludeFiles` | `getIncludePaths` | `getSourceFiles` | `getSourcePaths` | `updateFilePathsAndExtensions`

Topics
"Customize Post-Code-Generation Build Processing"

Introduced in R2006a

getIncludePaths

Get include paths from build information

Syntax

`paths = getIncludePaths(buildinfo,replaceMatlabroot,includeGroups,`
`excludeGroups)`

Description

`paths = getIncludePaths(buildinfo,replaceMatlabroot,includeGroups,`
`excludeGroups)` returns the names of include file paths from the build information.

The function requires the *`buildinfo`* and *`replaceMatlabroot`* arguments. You can use optional *`includeGroups`* and *`excludeGroups`* arguments. These optional arguments let you include or exclude groups selectively from the include paths returned by the function.

If you choose to specify *`excludeGroups`* and omit *`includeGroups`*, specify a null character vector (`''`) for *`includeGroups`*.

Examples

Get Include Paths from Build Information

Get the include paths from the build information, `myBuildInfo`.

```
myBuildInfo = RTW.BuildInfo;
addIncludePaths(myBuildInfo,{'/etc/proj/etclib' ...
   '/etcproj/etc/etc_build' '/common/lib'}, ...
   {'etc' 'etc' 'shared'});
incpaths = getIncludePaths(myBuildInfo,false);
```

```
>> incpaths

incpaths =

    '\etc\proj\etclib'   [1x22 char]    '\common\lib'
```

Get Include Paths with Include Group Argument

Get the paths in group `shared` from the build information, `myBuildInfo`.

```
myBuildInfo = RTW.BuildInfo;
addIncludePaths(myBuildInfo,{'/etc/proj/etclib' ...
   '/etcproj/etc/etc_build' '/common/lib'}, ...
   {'etc' 'etc' 'shared'});
incpaths = getIncludePaths(myBuildInfo,false,'shared');
```

```
>> incpaths

incpaths =

    '\common\lib'
```

Input Arguments

`buildinfo` — Name of the build information object returned by `RTW.BuildInfo`
object

`replaceMatlabroot` — Choice of whether to replace the `$(MATLAB_ROOT)` token with absolute paths in return from the function
`false` | `true`

Use the *`replaceMatlabroot`* argument to control whether the function includes the MATLAB root definition in the output that it returns.

Specify	Function Action
`true`	Replaces the token `$(MATLAB_ROOT)` with the absolute path for your MATLAB installation folder.
`false`	Does not replace the token `$(MATLAB_ROOT)`.

Example: `true`

`includeGroups` — Group names of include paths to include in the return from the function
cell array of character vectors | string

To use the *`includeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

`excludeGroups` — Group names of include paths to exclude from the return from the function
cell array of character vectors | string

To use the *`excludeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

Output Arguments

`paths` — Paths of include files from the build information
cell array of character vectors

See Also

`addIncludePaths` | `getIncludeFiles` | `getSourceFiles` | `getSourcePaths`

Topics
"Customize Post-Code-Generation Build Processing"

Introduced in R2006a

getLinkFlags

Get link options from build information

Syntax

`options = getLinkFlags(buildinfo,includeGroups,excludeGroups)`

Description

`options = getLinkFlags(buildinfo,includeGroups,excludeGroups)` returns linker options from the build information.

The function requires the *`buildinfo`* argument. You can use optional *`includeGroups`* and *`excludeGroups`* arguments. These optional arguments let you include or exclude groups selectively from the compiler options returned by the function.

If you choose to specify *`excludeGroups`* and omit *`includeGroups`*, specify a null character vector (`''`) for *`includeGroups`*.

Examples

Get Linker Options from Build Information

Get the linker options from the build information, `myBuildInfo`.

```
myBuildInfo = RTW.BuildInfo;
addLinkFlags(myBuildInfo,{'-MD -Gy' '-T'},'OPTS');
linkflags = getLinkFlags(myBuildInfo);
```

```
>> linkflags

linkflags = 

    '-MD -Gy'    '-T'
```

Get Linker Options with Include Group Argument

Get the linker options with the group name `Debug` from the build information, `myBuildInfo`.

```
myBuildInfo = RTW.BuildInfo;
addLinkFlags(myBuildInfo,{'-MD -Gy' '-T'}, ...
   {'Debug' 'MemOpt'});
linkflags = getLinkFlags(myBuildInfo,{'Debug'});
```

```
>> linkflags

linkflags = 
```

```
    '-MD -Gy'
```

Get Linker Options with Exclude Group Argument

Get the linker options from the build information `myBuildInfo`, except those options with the group name `Debug`.

```
myBuildInfo = RTW.BuildInfo;
addLinkFlags(myBuildInfo,{'-MD -Gy' '-T'}, ...
   {'Debug' 'MemOpt'});
linkflags = getLinkFlags(myBuildInfo,'',{'Debug'});
```

```
>> linkflags

linkflags = 

    '-T'
```

Input Arguments

`buildinfo` — Name of the build information object returned by `RTW.BuildInfo`
object

`includeGroups` — Group names of linker options to include in the return from the function
cell array of character vectors | string

To use the *`includeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

`excludeGroups` — Group names of linker options to exclude from the return from the function
cell array of character vectors | string

To use the *`excludeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

Output Arguments

`options` — Linker options from the build information
cell array of character vectors

See Also

`addLinkFlags` | `getCompileFlags` | `getDefines`

Topics
"Customize Post-Code-Generation Build Processing"

Introduced in R2006a

getNonBuildFiles

Get nonbuild-related files from build information

Syntax

```
files = getNonBuildFiles(buildinfo, concatenatePaths,replaceMatlabroot,
includeGroups,excludeGroups)
```

Description

`files = getNonBuildFiles(buildinfo, concatenatePaths,replaceMatlabroot, includeGroups,excludeGroups)` returns the names of non-build files from the build information, such as DLL files required for a final executable or a README file.

The function requires the *`buildinfo`*, *`concatenatePaths`*, and *`replaceMatlabroot`* arguments. You can use optional *`includeGroups`* and *`excludeGroups`* arguments. These optional arguments let you include or exclude groups selectively from the non-build files returned by the function.

If you choose to specify *`excludeGroups`* and omit *`includeGroups`*, specify a null character vector (`''`) for *`includeGroups`*.

The makefile for the build resolves file locations based on source paths and rules. The build process does not require you to resolve the path of every file in the build information. If you specify `true` for the *`concatenatePaths`* argument, the `getNonBuildFiles` function returns the path for each file:

- If a path was explicitly associated with the file when it was added.
- If you called `updateFilePathsAndExtensions` to resolve file paths and extensions before calling `getIncludeFiles`.

Examples

Get Nonbuild Files from Build Information

Get the nonbuild file names stored in the build information, `myBuildInfo`.

```
myBuildInfo = RTW.BuildInfo;
addNonBuildFiles(myBuildInfo,{'readme.txt' 'myutility1.dll' ...
   'myutility2.dll'});
nonbuildfiles = getNonBuildFiles(myBuildInfo,false,false);
```

```
>> nonbuildfiles

nonbuildfiles =
```

```
    'readme.txt'    'myutility1.dll'    'myutility2.dll'
```

Input Arguments

`buildinfo` — Name of the build information object returned by `RTW.BuildInfo`
object

`concatenatePaths` — Choice of whether to concatenate paths and file names in return from function
`false` | `true`

Specify	Function Action
`true`	Concatenates and returns each file name with its corresponding path.
`false`	Returns only file names.

Example: `true`

`replaceMatlabroot` — Choice of whether to replace the `$(MATLAB_ROOT)` token with absolute paths in return from function
`false` | `true`

Use the *`replaceMatlabroot`* argument to control whether the function includes the MATLAB root definition in the output that it returns.

Specify	Function Action
`true`	Replaces the token `$(MATLAB_ROOT)` with the absolute path for your MATLAB installation folder.
`false`	Does not replace the token `$(MATLAB_ROOT)`.

Example: `true`

`includeGroups` — Group names of non-build files to include in the return from the function
cell array of character vectors | string

To use the *`includeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

`excludeGroups` — Group names of non-build files to exclude from the return from the function
cell array of character vectors | string

To use the *`excludeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

Output Arguments

`files` — Names of non-build files from the build information
cell array of character vectors

See Also
`addNonBuildFiles`

Topics
"Customize Post-Code-Generation Build Processing"

Introduced in R2008a

getSourceFiles

Get source files from build information

Syntax

`srcfiles = getSourceFiles(buildinfo,concatenatePaths,replaceMatlabroot, includeGroups,excludeGroups)`

Description

`srcfiles = getSourceFiles(buildinfo,concatenatePaths,replaceMatlabroot, includeGroups,excludeGroups)` returns the names of source files from the build information.

The function requires the *`buildinfo`*, *`concatenatePaths`*, and *`replaceMatlabroot`* arguments. You can use optional *`includeGroups`* and *`excludeGroups`* arguments. These optional arguments let you include or exclude groups selectively from the source files returned by the function.

If you choose to specify *`excludeGroups`* and omit *`includeGroups`*, specify a null character vector (`''`) for *`includeGroups`*.

The makefile for the build resolves file locations based on source paths and rules. The build process does not require you to resolve the path of every file in the build information. If you specify `true` for the *`concatenatePaths`* argument, the `getSourceFiles` function returns the path for each file:

- If a path was explicitly associated with the file when it was added.
- If you called `updateFilePathsAndExtensions` to resolve file paths and extensions before calling `getSourceFiles`.

Examples

Get Source Files from Build Information

Get the source paths and file names from the build information, `myBuildInfo`.

```
myBuildInfo = RTW.BuildInfo;
addSourceFiles(myBuildInfo, ...
   {'test1.c' 'test2.c' 'driver.c'},'', ...
   {'Tests' 'Tests' 'Drivers'});
srcfiles = getSourceFiles(myBuildInfo,false,false);
```

```
>> srcfiles

srcfiles = 

    'test1.c'    'test2.c'    'driver.c'
```

Get Source Files with Include Group Argument

Get the names of source files in group `tests` from the build information, `myBuildInfo`.

```
myBuildInfo = RTW.BuildInfo;
addSourceFiles(myBuildInfo,{'test1.c' 'test2.c'...
   'driver.c'}, {'/proj/test1' '/proj/test2'...
   '/drivers/src'}, {'tests', 'tests', 'drivers'});
incfiles = getSourceFiles(myBuildInfo,false,false,...
   'tests');
```

```
>> incfiles

incfiles = 

    'test1.c'    'test2.c'
```

Input Arguments

buildinfo — Name of the build information object returned by RTW.BuildInfo
object

concatenatePaths — Choice of whether to concatenate paths and file names in return
false | true

Specify	Function Action
true	Concatenates and returns each file name with its corresponding path.
false	Returns only file names.

Example: `true`

replaceMatlabroot — Choice of whether to replace the $(MATLAB_ROOT) token with absolute paths in return
false | true

Use the *replaceMatlabroot* argument to control whether the function includes the MATLAB root definition in the output it returns.

Specify	Function Action
true	Replaces the token `$(MATLAB_ROOT)` with the absolute path for your MATLAB installation folder.
false	Does not replace the token `$(MATLAB_ROOT)`.

Example: `true`

includeGroups — Group names of source files to include in the return from the function
cell array of character vectors | string

To use the *includeGroups* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

`excludeGroups` — Group names of source files to exclude from the return from the function
cell array of character vectors | string

To use the *`excludeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

Output Arguments

`srcfiles` — Names of source files from the build information
cell array of character vectors

The names of source files that you add with the `addSourceFiles` function. If you call the `packNGo` function, the names include files that `packNGo` found and added while packaging code.

See Also

`addSourceFiles` | `getIncludeFiles` | `getIncludePaths` | `getSourcePaths` | `updateFilePathsAndExtensions`

Topics
"Customize Post-Code-Generation Build Processing"

Introduced in R2006a

getSourcePaths

Get source paths from build information

Syntax

```
srcpaths = getSourcePaths(buildinfo,replaceMatlabroot,includeGroups,
excludeGroups)
```

Description

`srcpaths = getSourcePaths(buildinfo,replaceMatlabroot,includeGroups, excludeGroups)` returns the names of source file paths from the build information. The function returns only the file paths that were added to the build information by using `addSourcePaths`. The build process uses build information source paths to locate source files that were specified without an explicit path.

The function requires the *`buildinfo`* and *`replaceMatlabroot`* arguments. You can use optional *`includeGroups`* and *`excludeGroups`* arguments. These optional arguments let you include or exclude groups selectively from the source paths returned by the function.

If you choose to specify *`excludeGroups`* and omit *`includeGroups`*, specify a null character vector (`''`) for *`includeGroups`*.

Examples

Get Source Paths from Build Information

Get the source paths from the build information, `myBuildInfo`.

```
myBuildInfo = RTW.BuildInfo;
addSourcePaths(myBuildInfo,{'/proj/test1' ...
   '/proj/test2' '/drivers/src'}, {'tests' 'tests' ...
   'drivers'});
srcpaths = getSourcePaths(myBuildInfo,false);
```

```
>> srcpaths

srcpaths =

   '\proj\test1'    '\proj\test2'      '\drivers\src'
```

Get Source Paths with Include Group Argument

Get the paths in group `tests` from the build information, `myBuildInfo`.

```
myBuildInfo = RTW.BuildInfo;
addSourcePaths(myBuildInfo,{'/proj/test1' ...
```

```
   '/proj/test2' '/drivers/src'}, {'tests' 'tests' ...
   'drivers'});
srcpaths = getSourcePaths(myBuildInfo,true,'tests');
```

```
>> srcpaths

srcpaths = 

     '\proj\test1'    '\proj\test2'
```

Get Source Paths from Build Information

Get a source path from the build information, `myBuildInfo`. First, get the path without replacing `$(MATLAB_ROOT)` with an absolute path. Then, get it with replacement. Here, the MATLAB root folder is `\\myserver\myworkspace\matlab`.

```
myBuildInfo = RTW.BuildInfo;
addSourcePaths(myBuildInfo, fullfile(matlabroot, ...
   'rtw', 'c', 'src'));
srcpaths = getSourcePaths(myBuildInfo,false);
```

```
>> srcpaths{:}

ans =

$(MATLAB_ROOT)\rtw\c\src

>> srcpaths = getSourcePaths(myBuildInfo,true);
>> srcpaths{:}

ans =

\\myserver\myworkspace\matlab\rtw\c\src
```

Input Arguments

`buildinfo` — Name of the build information object returned by `RTW.BuildInfo`
object

`RTW.BuildInfo` object that contains information for compiling and linking generated code.

`replaceMatlabroot` — Choice of whether to replace the `$(MATLAB_ROOT)` token with absolute paths in return
`false` | `true`

Use the *`replaceMatlabroot`* argument to control whether the function includes the MATLAB root definition in the output it returns.

Specify	Function Action
`true`	Replaces the token `$(MATLAB_ROOT)` with the absolute path for your MATLAB installation folder.

Specify	Function Action
`false`	Does not replace the token `$(MATLAB_ROOT)`.

Example: `true`

`includeGroups` — Group names of source paths to include in the return from the function
cell array of character vectors | string

To use the *`includeGroups`* argument, view available groups with `myGroups = getGroups(buildInfo)`.

Example: `''`

`excludeGroups` — Group names of source paths to exclude from the return from the function
cell array of character vectors | string

To use the *`excludeGroups`* argument, view available groups with `myGroups = getGroups(buildInfo)`.

Example: `''`

Output Arguments

`srcpaths` — Source file paths
cell array of character vectors

Paths of source files from the build information.

See Also

`addSourcePaths` | `getIncludeFiles` | `getIncludePaths` | `getSourceFiles`

Topics
"Customize Post-Code-Generation Build Processing"

Introduced in R2006a

model_initialize

Generated C/C++ entry-point function that contains initialization code for a Simulink model

Syntax

```
void model_initialize(void)
```

Description

`void model_initialize(void)` is a generated C or C++ entry-point function called one time to execute the initialization code for a Simulink model. This function is not intended to reset the real-time model data structure (rtM) for a model.

The generated calling interface of the initialize entry-point function for a model differs depending on the **Language** and **Code interface packaging** parameters. For more information, see “Code interface packaging”.

To preview and customize the name of a generated C initialize entry-point function an Embedded Coder license is required. To preview the function, open the Code Mappings editor and click the **Functions** tab. To customize the function name, in the **Function Name** column click and edit the spreadsheet directly. To customize the function using a template, in the **Function Customization Template** column select a template to apply to the function. For more information, see “Configure Names for Individual C Entry-Point Functions”, and “Configure Default Code Generation for Functions”.

To view the generated initialize entry-point function, open the **Code** view or Code Generation Report and examine the source code for your model. For more information see, “Analyze the Generated Code Interface”.

Examples

Generate An Initialize Entry-Point Function

This example shows the basic workflow for how to configure, customize, generate, and examine an initialize entry-point function. This specific example generates a nonreusable C initialization function for the model `rtwdemo_irt_base`.

1. Open a model. For this example, use rtwdemo_irt_base.
2. Select a coder. In the Apps gallery, click **Simulink Coder** or **Embedded Coder**.
3. Configure the parameters. In the Configuration Parameters dialog box, set the **Language** and **Code interface packaging** parameters. In this example, the parameters are set for you.
4. (Embedded Coder only) Customize the function. Using Embedded Coder, you can customize the name of the initialize entry-point function.

   - Open the Code Mappings editor.
   - Click the **Functions** tab.

- Edit the spreadsheet directly. In the **Function Name** column, enter a name for the function.

5. Generate code.
6. Examine the generated code. In the **Code** view, verify that the generated initialize function appears with the expected name and parameters.

Input Arguments

`void` — Default initialize function parameter

void (default)

The initialize entry-point C or C++ function provides an interface to start application code. By default, the generated function provides a `void-void` interface that does not have arguments.

Example: void

Output Arguments

`void` — Initialize function return value

void (default)

The initialize entry-point C or C++ function provides an interface to start application code. By default, the generated function provides a `void-void` interface that does not have a return value.

Example: void

See Also

`model_reset` | `model_step` | `model_terminate`

Topics

"Configure C Code Generation for Model Entry-Point Functions"
"Configure Names for Individual C Entry-Point Functions"
"Configure Default Code Generation for Functions"
"Analyze the Generated Code Interface"
"Generate Code That Responds to Initialize, Reset, and Terminate Events"

Introduced before R2006a

model_reset

Generated C/C++ entry-point function that contains reset code for a Simulink model

Syntax

```
void model_reset(void)
```

Description

`void model_reset(void)` is a generated C or C++ entry-point function that executes reset code for a Simulink model. If a model includes a Reset Function block, reset code is generated. To reset conditions or state, call the function from the application code.

The generated calling interface of the reset entry-point function for a model differs depending on the **Language** and **Code interface packaging** parameters. For more information, see "Code interface packaging".

To preview and customize the name of the generated reset entry-point function an Embedded Coder license is required. To preview the reset entry-point function, open the Code Mappings editor and click the **Functions** tab. To customize the function name, in the **Function Name** column click and edit the spreadsheet directly. To customize the function name and arguments, in the **Function Preview** column click the function hyperlink and configure the reset function from the opened dialogue box. To customize a function using a template, in the **Function Customization Template** column select a template to apply to the function. For more information, see "Configure Name and Arguments for Individual Step Functions", "Interactively Configure C++ Interface", and "Configure Default Code Generation for Functions".

To view the generated reset entry-point function, open the **Code** view or Code Generation Report and view the source code for your model. For more information see, "Analyze the Generated Code Interface".

Examples

Generate A Reset Entry-Point Function

This example shows the basic workflow for how to configure, customize, generate, and examine a reset entry-point function. This specific example generates a nonreusable C reset function for the model `rtwdemo_irt_base`.

1. Open a model. For this example, use rtwdemo_irt_base.
2. Select a coder. In the Apps gallery, click **Simulink Coder** or **Embedded Coder**.
3. Configure the parameters. In the Configuration Parameters dialog box, set the **Language**, and **Code interface packaging** parameters. In this example, the parameters are set for you.
4. (Embedded Coder only) Customize the function. Using Embedded Coder, you can customize the name of the reset entry-point function.

   - Open the Code Mappings editor.

- Click the **Functions** tab.
- Edit the spreadsheet directly. In the **Function Name** column, enter a name for the function.

5. Generate code.
6. Examine the generated code. In the **Code** view, verify that the generated reset function appears with the expected name and parameters.

Input Arguments

`void` — Default reset function parameter

void (default)

The reset entry-point C or C++ function provides an interface to the model reset code. By default, the generated function provides a `void-void` interface that does not have arguments.

Output Arguments

`void` — Reset function return value

void

The reset entry-point C or C++ function provides an interface to model reset code. By default, the generated function provides a `void-void` interface that does not have a return value.

See Also

`model_initialize` | `model_step` | `model_terminate`

Topics

"Configure C Code Generation for Model Entry-Point Functions"
"Configure Name and Arguments for Individual Step Functions"
"Interactively Configure C++ Interface"
"Configure Default Code Generation for Functions"
"Analyze the Generated Code Interface"
"Generate Code That Responds to Initialize, Reset, and Terminate Events"

Introduced before R2006a

model_step

Generated C/C++ entry-point function that contains execution code for each step in a Simulink model

Syntax

```
void model_step(void)
void model_step_N(void)
```

Description

`void model_step(void)` is an execution function that contains the output and update code for the blocks in a Simulink model.

`void model_step_N(void)` is an execution function with a task identifier that contains the output and update code for the blocks in a Simulink model.

The step entry-point function computes the current values of the blocks. If logging is enabled, the step function updates logging variables. By design, the step function is called at the interrupt level from `rt_OneStep` (invoked as a timer ISR). The `rt_OneStep` function calls the `model_step` function to execute processing for one clock period of the model. For a more information, see "rt_OneStep and Scheduling Considerations".

If the model has a finite stop time, the step function signals the end of execution when the current time equals the stop time. Otherwise, if one or more of these conditions are true, the step function does not check the current time against the stop time and the program runs indefinitely:

- The model stop time is set to `inf`.
- Logging is disabled.
- Parameter **Terminate function required** is not selected.

The generated calling interface of the step entry-point function for a model differs depending on these parameters:

1. To generate a step entry-point function, select the **Single output/update function** parameter. If you clear this parameter, `model_output` and `model_update` entry-point functions are generated in place of the step function.
2. To generate a single step function with configurable arguments, clear the **Treat each discrete rate as a separate task** parameter. To generate separate step functions based on timing requirements, select this parameter. For more information, see Treat each discrete rate as a separate task.

Parameter Value	Function Prototype
`Off` (single rate or multirate single-tasking model)	`void model_step(void);`
`On` (multirate multi-tasking model)	`void model_step_N (void);` (`N` is a task identifier)

3. To change the generated calling interface, set the **Language** and **Code interface packaging** parameters. For more information, see “Code interface packaging”.

To preview and customize the name and arguments of the generated C or C++ step entry-point function an Embedded Coder license is required. To preview a step entry-point function, open the Code Mappings editor and click the **Functions** tab. To customize the function name, in the **Function Name** column click and edit the spreadsheet directly. To customize the function name and arguments, in the **Function Preview** column click the function hyperlink and configure the step function from the opened dialog box. To customize a function using a template, in the **Function Customization Template** column select a template to apply to the function. For more information, see “Configure Name and Arguments for Individual Step Functions”, “Interactively Configure C++ Interface”, and “Configure Default Code Generation for Functions”.

To view the generated step entry-point function, open the **Code** view or Code Generation Report and view the source code for your model. For more information see, “Analyze the Generated Code Interface”.

Examples

Generate A Step Entry-Point Function

This example shows the basic workflow for how to configure, customize, generate, and examine a step entry-point function. This specific example generates a nonreusable C termination function for the model `rtwdemo_irt_base`.

1. Open a model. For this example, use the `rtwdemo_irt_base` model.
2. Select a coder. In the Apps gallery, click **Simulink Coder** or **Embedded Coder**.
3. Configure the parameters. In the Configuration Parameters dialog box, set the **Single output/update function**, **Treat each discrete rate as a separate task**, **Language**, and **Code interface packaging** parameters. In this example, the parameters are set for you.
4. (Embedded Coder only) Customize the function. Using Embedded Coder, you can customize the name and arguments of the step entry-point function.

   - Open the Code Mappings editor.
   - Click on the **Functions** tab.
   - Customize the name and arguments. In the **Function Preview** column, click the function hyperlink to open the **Configure C Step Function Interface** dialog box. Configure the name and arguments.
5. Generate code.
6. Examine the generated code. In the **Code** view, verify the generated termination function appears with the expected name and parameters.

Input Arguments

`void` — Default step function parameter

void (default)

The step entry-point C or C++ function provides an interface to the model execution code. By default, the generated function provides a `void-void` interface that does not have arguments. To configure

the input arguments for a C step function, use the **Code Mappings Editor**. To configure the input arguments for a C++ step function, use the **Code Mappings- C++ Editor**.

Output Arguments

`void` — Step function return value
void

The step entry-point C or C++ function provides an interface to the model execution code. By default, the generated function provides a `void-void` interface that does not have arguments. To configure the output arguments for a C step function, use the **Code Mappings Editor**. To configure the output arguments for a C++ step function, use the **Code Mappings- C++ Editor**.

See Also

`model_initialize` | `model_reset` | `model_terminate`

Topics
"Configure C Code Generation for Model Entry-Point Functions"
"Configure Name and Arguments for Individual Step Functions"
"Interactively Configure C++ Interface"
"Configure Default Code Generation for Functions"
"Analyze the Generated Code Interface"
"Generate Code That Responds to Initialize, Reset, and Terminate Events"

Introduced before R2006a

model_terminate

Generated C/C++ entry-point function that contains termination code for a Simulink model

Syntax

```
void model_terminate(void)
```

Description

`void model_terminate(void)` is a generated C or C++ entry-point function called one time to execute termination code for a Simulink model.

The generated calling interface of the termination entry-point function for a model differs depending on the **Language** and **Code interface packaging** parameters. For more information, see “Code interface packaging”. With Embedded Coder, you can choose whether to generate a termination function for a model by using the **Terminate function required** parameter. If your application runs indefinitely, clear this parameter. For more information, see “Terminate function required” on page 14-32.

To preview and customize the name of a generated C terminate entry-point function an Embedded Coder license is required. To preview the terminate entry-point function, open the Code Mappings editor and click the **Functions** tab. To customize the function name, in the **Function Name** column click and edit the spreadsheet directly. To customize the function using a template, in the **Function Customization Template** column select a template to apply to the function. For more information, see “Configure Names for Individual C Entry-Point Functions”, and “Configure Default Code Generation for Functions”.

To view the generated terminate entry-point function, open the **Code** view or Code Generation Report and examine the source code for your model. For more information see, “Analyze the Generated Code Interface”.

Examples

Generate A Terminate Entry-Point Function

This example shows the basic workflow for how to configure, customize, generate, and examine the terminate entry-point function. This specific example generates a nonreusable C terminate function for the model `rtwdemo_irt_base`.

1. Open a model. For this example, use rtwdemo_irt_base.
2. Select a coder. In the Apps gallery, click **Simulink Coder** or **Embedded Coder**.
3. Configure the parameters. In the Configuration Parameters dialog box, select **Terminate function required** and set the **Language** and **Code interface packaging** parameters. In this example, the parameters are set for you.
4. (Embedded Coder only) Customize the function. Using Embedded Coder, you can customize the name of the terminate entry-point function.

- Open the Code Mappings editor.
- Click the **Functions** tab.
- Edit the spreadsheet directly. In the **Function Name** column, enter a name for the function.

5. Generate code.
6. Examine the generated code. In the **Code** view, verify that the generated terminate function appears with the expected name and parameters.

Input Arguments

`void` — Default terminate function parameter
void (default)

The terminate entry-point C or C++ function provides an interface to terminate application code. By default, the generated function provides a `void-void` interface that does not have arguments.

Example: void

Output Arguments

`void` — Terminate function return value
void (default)

The terminate entry-point C or C++ function provides an interface to terminate application code. By default, the generated function provides a `void-void` interface that does not have a return value.

Example: void

See Also

`model_initialize` | `model_reset` | `model_step`

Topics
"Configure C Code Generation for Model Entry-Point Functions"
"Configure Names for Individual C Entry-Point Functions"
"Configure Default Code Generation for Functions"
"Analyze the Generated Code Interface"
"Generate Code That Responds to Initialize, Reset, and Terminate Events"

Introduced before R2006a

packNGo

Package generated code in ZIP file for relocation

Syntax

```
packNGo(buildInfo,Name,Value)
```

Description

`packNGo(buildInfo,Name,Value)` packages the code files in a compressed ZIP file so that you can relocate, unpack, and rebuild them in another development environment. The list of name-value pairs is optional.

The ZIP file can include these types of files:

- Source files (for example, `.c` and `.cpp` files)
- Header files (for example, `.h` and `.hpp` files)
- MAT-file that contains the build information object (`.mat` file)
- Nonbuild-related files (for example, `.dll` files and `.txt` informational files) required for a final executable
- Build-generated binary files (for example, executable `.exe` file or dynamic link library `.dll`).

  The code generator includes the build-generated binary files (if present) in the ZIP file. The **ignoreFileMissing** property does not apply to build-generated binary files.
- CMake configuration files (`CMakeLists.txt`) that you use to generate makefiles or projects for a compiler environment.

Use this function to relocate files. You can then recompile the files for a specific target environment or rebuild them in a development environment in which MATLAB is not installed. By default, the function packages the files as a flat folder structure in a ZIP file within the code generation folder. You can customize the output by specifying name-value pairs. After relocating the ZIP file, use a standard ZIP utility to unpack the compressed file.

The `packNGo` function can potentially modify the build information passed in the first `packNGo` argument. As part of code packaging, `packNGo` can find additional files from source and include paths recorded in the build information. When these files are found, `packNGo` adds them to the build information.

To ensure that `packNGo` finds header files, add their paths to `buildInfo` by using the `addIncludePaths` function.

Note When generating standalone code by using the `codegen` command, you can use the `-package` option to both generate code and package the code in a ZIP file in a single step.

Examples

Run packNGo from Command Window

After the build process is complete, you can run packNGo from the Command Window. Use `packNGo` for ZIP file packaging of generated code in the file `portzingbit.zip`. Maintain the relative file hierarchy.

1. Change folders to the code generation folder. For example, using MATLAB Coder, `codegen/dll/zingbit` or for Simulink code generation, `zingbit_grt_rtw`.
2. Load the `buildInfo` object that describes the build.
3. Run `packNGo` with property settings for `packType` and `fileName`.

```
cd codegen/dll/zingbit;
load buildInfo.mat
packNGo(buildInfo,'packType', 'hierarchical', ...
   'fileName','portzingbit');
```

Configure packNGo in the Simulink Editor

If you configure ZIP file packaging from the code generation pane, the code generator uses `packNGo` to output a ZIP file during the build process.

1. Select **Code Generation** > **Package code and artifacts**. Optionally, provide a **Zip file name**. To apply the changes, click **OK**.
2. Build the model. At the end of the build process, the code generator outputs the ZIP file. The folder structure in the ZIP file is hierarchical.

Configure packNGo for Simulink from the Command Line

If you configure ZIP file packaging by using the `set_param` function, the code generator uses `packNGo` to output a ZIP file during the build process.

To configure ZIP file packaging for the model `zingbit` in the file `zingbit.zip`, use the `set_param` function.

```
set_param('zingbit','PostCodeGenCommand', ...
   'packNGo(buildInfo);');
```

Input Arguments

`buildInfo` — Object that provides build information

`buildInfo` object | path to `buildInfo.mat`

During the build process, the code generator places `buildInfo.mat` in the code generation folder. This MAT-file contains the `buildInfo` object. The object provides information that `packNGo` uses to produce the ZIP file.

You can specify the argument as a `buildInfo` object:

```
load buildInfo.mat
packNGo(buildInfo,'packType', 'hierarchical', ...
   'fileName','portzingbit');
```

Or, you can specify the argument as the path to the buildInfo.mat file:

```
buildInfoFile = fullfile(pathToBuildFolder, 'buildInfo.mat');
packNGo(buildInfoFile,'packType', 'hierarchical', ...
   'fileName','portzingbit');
```

Or, you can specify the argument as the path to the folder that contains buildInfo.mat:

```
packNGo(pathToBuildFolder,'packType', 'hierarchical', ...
   'fileName','portzingbit');
```

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'packType','flat','nestedZipFiles',true

packType — Specify structure type of ZIP file
'flat' (default) | 'hierarchical'

If 'flat', package the generated code files in a ZIP file as a single, flat folder. The function does *not* package:

- Child buildInfo.mat files.
- CMakeLists.txt files.

If 'hierarchical', package the generated code files hierarchically in a primary ZIP file. The hierarchy contains top model, referenced model, and shared utility folders. The function also packages:

- The corresponding buildInfo.mat files for the folders.
- CMakeLists.txt files in the build folder.

Example: 'packType','flat'

nestedZipFiles — Determines whether the paths for files in the secondary ZIP files are relative to the root folder of the primary ZIP file
true (default) | false

If true, create a primary ZIP file that contains three secondary ZIP files:

- mlrFiles.zip — Files in your *matlabroot* folder tree
- sDirFiles.zip — Files in and under your code generation folder
- otherFiles.zip — Required files not in the *matlabroot* or start folder trees

If false, create a primary ZIP file that contains folders, for example, your code generation folder and *matlabroot*.

Example: 'nestedZipFiles',true

fileName — Specifies a file name for the primary ZIP file
'*modelOrFunctionName*.zip' (default) | '*myName*'

If you do not specify the `'fileName'`-value pair, the function packages the files in a ZIP file named *`modelOrFunctionName`*`.zip` and places the ZIP file in the code generation folder.

If you specify `'fileName'` with the value, `'`*`myName`*`'`, the function creates *`myName`*`.zip` in the code generation folder.

To specify another location for the primary ZIP file, provide the absolute path to the location, *`fullPath/myName`*`.zip`

Example: `'fileName','/home/user/myModel.zip'`

`minimalHeaders` — Selects whether to include only the minimal header files
`true` (default) | `false`

If `true`, include only the minimal header files required to build the code in the ZIP file.

If `false`, include header files found on the include path in the ZIP file.

Example: `'minimalHeaders',true`

`includeReport` — Selects whether to include the `html` folder for your code generation report
`false` (default) | `true`

If `false`, do not include the `html` folder in the ZIP file.

If `true`, include the `html` folder in the ZIP file.

Example: `'includeReport',false`

`ignoreParseError` — Instruct packNGo not to terminate on parse errors
false (default) | true

If `false`, terminate on parse errors.

If `true`, do not terminate on parse errors.

Example: `'ignoreParseError',false`

`ignoreFileMissing` — Instruct packNGo not to terminate if files are missing
false (default) | true

If `false`, terminate on missing file errors.

If `true`, do not terminate on missing files errors.

Example: `'ignoreFileMissing',false`

Limitations

- The function operates on source files only, such as `*.c`, `*.cpp`, and `*.h` files. The function does not support compile flags, defines, or makefiles.
- The function does not package source files for reusable library subsystems.
- Unnecessary files might be included. The function might find additional files from source paths and include paths recorded in the build information, even if those files are not used.

See Also
`codebuild`

Topics
"Customize Post-Code-Generation Build Processing"
"Relocate Code to Another Development Environment"
"Compile Code in Another Development Environment"

Introduced in R2006b

rtiostreamtest

Test custom `rtiostream` interface implementation

Syntax

```
rtiostreamtest(connection, parameterOne, parameterTwo, verbosityFlag)
rtiostreamtest('tcp', host, port)
rtiostreamtest('serial', port, baud)
```

Description

`rtiostreamtest(connection, parameterOne, parameterTwo, verbosityFlag)` runs a test suite to verify your custom `rtiostream` interface implementation.

`rtiostreamtest('tcp', host, port)`, via TCP/IP communication, connects MATLAB to the target hardware using the specified `host` and `port`.

`rtiostreamtest('serial', port, baud)`, via serial communication, connects MATLAB to the target hardware using the specified `port` and `baud` value.

During initialization, the function uses basic `rtiostream` I/O. The function determines:

- Byte ordering of data on the target hardware.
- Granularity of memory address.
- Size of data types.
- Whether `rtIOStreamRecv` blocks, that is, when there is no data whether `rtIOStreamRecv` waits for data or returns immediately with size received == 0.
- The size (`BUFFER_SIZE`) of its internal buffer for receiving or transmitting data through `rtiostream`. The default is 128 bytes.

In `Test 1 (fixed size data exchange)`, the function:

- Checks data can be sent and received correctly in different chunk sizes. The chunk sizes for your development computer and target hardware are *symmetric*.
- Sends data as a known sequence that it can validate.
- Performs “host-to-target” tests. Your development computer sends data and your target hardware receives data in successive chunks of 1, 4, and 128 bytes.
- Performs “target-to-host” tests. Your target hardware sends data and your development computer receives data in successive chunks of 1, 4, and 128 bytes.

In `Test 2 (varying size data exchange)`, the function:

- Checks that data can be sent and received correctly in different chunk sizes. The chunk sizes for your development computer and target hardware are *asymmetric*.
- Sends data as a known sequence that it can validate.
- Performs “host-to-target” tests:

  - Your development computer sends data in chunks of 128 bytes and your target hardware receives data in chunks of 64 bytes.
  - Your development computer sends data in chunks of 64 bytes and your target hardware receives data in chunks of 128 bytes.
- Performs "target-to-host" tests:
  - Your target hardware sends data in chunks of 64 bytes and your development computer receives data in chunks of 128 bytes.
  - Your target hardware sends data in chunks of 128 bytes and your development computer receives data in chunks of 64 bytes.

In `Test 3 (receive buffer detection)`, the function determines the data that it can store in between calls to `rtIOStreamRecv` on the target hardware. The function uses an iterative process:

1. The development computer transmits a data sequence while the target hardware sleeps. `rtIOStreamRecv` is not called while the target hardware sleeps.
2. When the target hardware wakes up, it calls `rtIOStreamRecv` to receive data from the internal buffer of the driver.
3. The function determines whether the internal buffer overflowed by checking for errors and checking the received data values.
4. If there are no overflow errors and the transmitted data is received correctly, the function starts another iteration, performing step 1 with a larger data sequence.

The function reports the size of the last known good buffer.

Examples

Verify Behavior of Custom `rtiostream` Interface Implementation

The test suite consists of two parts. One part of the test suite is an application that runs on the target hardware. The other part runs in MATLAB.

1. To create the target application, compile and link these files:
   - `rtiostreamtest.c`
   - `rtiostreamtest.h`
   - `rtiostream.h`
   - The `rtiostream` implementation under investigation, for example, `rtiostream_tcpip.c`.
   - `main.c`

   `rtiostreamtest.c`, `rtiostreamtest.h`, and `main.c` are located in *`matlabroot`*`/toolbox/coder/rtiostream/src/rtiostreamtest`.
2. Download and run the application on your target hardware.
3. To run the MATLAB part of the test suite, invoke the `rtiostreamtest` function. For example:

   ```
   rtiostreamtest('tcp','myProcessor','2345')
   ```

   The function produces an output like this:

   ```
   ### Test suite for rtiostream ###
   Initializing connection with target...
   ```

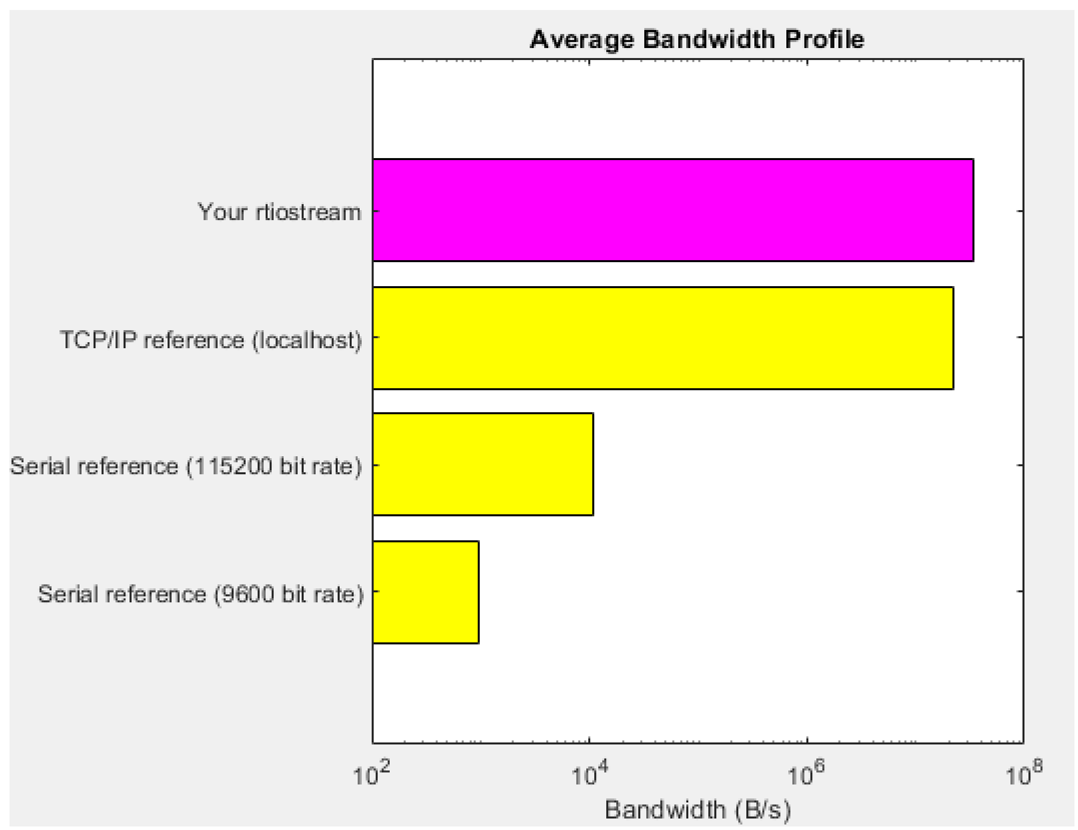
```
### Hardware characteristics discovered
Size of char    : 8 bit
Size of short   : 16 bit
Size of int     : 32 bit
Size of long    : 32 bit
Size of float   : 32 bit
Size of double  : 64 bit
Size of pointer : 64 bit
Byte ordering   : Little Endian

### rtiostream characteristics discovered
Round trip time : 0.25098 ms
rtIOStreamRecv behavior : non-blocking

### Test results
Test 1 (fixed size data exchange): ......... PASS
Test 2 (varying size data exchange): ......... PASS

### Test suite for rtiostream finished successfully ###
```

The function also generates the average bandwidth profile.



Input Arguments

`connection` — Transport protocol

'tcp' | 'serial'

Specify transport protocol for communication channel:

- `'tcp'` –– TCP/IP
- `'serial'` –– RS-232 serial

`parameterOne` — Host name or COM port
character vector | string scalar

If `connection` is `'tcp'`, specify name of target processor. For example, if your development computer is the target processor, you can specify `'localhost'`.

If `connection` is `'serial'`, specify serial port ID, for example, `'COM1'` for COM1, ‘`'COM2'` for COM2, and so on.

`parameterTwo` — Port number or baud value
integer

If `connection` is `'tcp'`, specify port number of TCP/IP server, an integer value between 256 and 65535.

If `connection` is `'serial'`, specify baud value, for example, 9600.

`verbosityFlag` — Verbosity
'' (default) | 'verbose'

If you specify `'verbose'`, the function displays messages that contain progress information. You can use the messages to debug runtime failures.

See Also

Topics
“Host-Target Communication for Simulink PIL simulation”
“Host-Target Communication for MATLAB PIL Execution”
“Customize XCP Slave Software”

Introduced in R2013a

rsimgetrtp

Global model parameter structure

Syntax

```
parameter_structure = rsimgetrtp('model')
```

Description

`parameter_structure = rsimgetrtp('model')` forces a block update diagram action for *`model`*, a model for which you are running rapid simulations, and returns the global parameter structure for that model. The function includes tunable parameter information in the parameter structure.

The model parameter structure contains the following fields:

Field	Description
`modelChecksum`	A four-element vector that encodes the structure. The code generator uses the *`checksum`* to check whether the structure has changed since the RSim executable was generated. If you delete or add a block, and then generate a new version of the structure, the new *`checksum`* will not match the original *`checksum`*. The RSim executable detects this incompatibility in model parameter structures and exits to avoid returning incorrect simulation results. If the structure changes, you must regenerate code for the model.
`parameters`	A structure that defines model global parameters.

The `parameters` substructure includes the following fields:

Field	Description
`dataTypeName`	Name of the parameter data type, for example, `double`
`dataTypeID`	An internal data type identifier
`complex`	Value 1 if parameter values are complex and 0 if real
`dtTransIdx`	Internal use only
`values`	Vector of parameter values
`structParamInfo`	Information about structure and bus parameters in the model

The `structParamInfo` substructure contains these fields:

Field	Description
`Identifier`	Name of the parameter
`ModelParam`	Value 1 if parameter is a model parameter and 0 if it is a block parameter

Field	Description
BlockPath	Block path for a block parameter. This field is empty for model parameters.
CAPIIdx	Internal use only

Do not modify fields in `structParamInfo`.

The function also includes an array of substructures `map` that represents tunable parameter information with these fields:

Field	Description
Identifier	Parameter name
ValueIndicies	Vector of indices to parameter values
Dimensions	Vector indicating parameter dimensions

Examples

Return global parameter structure for model `rtwdemo_rsimtf` to `param_struct`:

```
rtwdemo_rsimtf
param_struct = rsimgetrtp('rtwdemo_rsimtf')

param_struct =

    modelChecksum: [1.7165e+009 3.0726e+009 2.6061e+009
2.3064e+009]
       parameters: [1x1 struct]
```

See Also

`rsimsetrtpparam`

Topics

“Create a MAT-File That Includes a Model Parameter Structure”
“Update Diagram and Run Simulation”
“Default parameter behavior”
“Block Authoring and Simulation Integration”
“Tune Parameters”

Introduced in R2006a

rsimsetrtpparam

Set parameters of `rtP` model parameter structure

Syntax

```
rtP = rsimsetrtpparam(rtP,idx)
rtP = rsimsetrtpparam(rtP,'paramName',paramValue)
rtP = rsimsetrtpparam(rtP,idx,'paramName',paramValue)
```

Description

`rtP = rsimsetrtpparam(rtP,idx)` expands the `rtP` structure to have `idx` sets of parameters. The `rsimsetrtpparam` utility defines the values of an existing `rtP` parameter structure. The `rtP` structure matches the format of the structure returned by `rsimgetrtp('`*`modelName`*`')`.

`rtP = rsimsetrtpparam(rtP,'paramName',paramValue)` takes an `rtP` structure with tunable parameter information and sets the values associated with `'paramName'`to be `paramValue` if possible. There can be more than one name-value pair.

`rtP = rsimsetrtpparam(rtP,idx,'paramName',paramValue)` takes an `rtP` structure with tunable parameter information and sets the values associated with `'paramName'` to be `paramValue` in the nth `idx` parameter set. There can be more than one name-value pair. If the `rtP` structure does not have `idx` parameter sets, the first set is copied and appended until there are `idx` parameter sets. Subsequently, the nth `idx`set is changed.

Examples

Expand Parameter Sets

Expand the number of parameter sets in the `rtp` structure to 10.

```
rtp = rsimsetrtpparam(rtp,10);
```

Add Parameter Sets

Add three parameter sets to the parameter structure `rtp`.

```
rtp = rsimsetrtpparam(rtp,idx,'X1',iX1,'X2',iX2,'Num',iNum);
```

Input Arguments

`rtP` — A parameter structure that contains the sets of parameter names and their respective values
parameter structure

`idx` — An index used to indicate the number of parameter sets in the `rtP` structure
index of parameter sets

`paramValue` — The value of the `rtP` parameter `paramName`
value of `paramName`

`paramName` — The name of the parameter set to add to the `rtP` structure
name of the parameter set

Output Arguments

`rtP` — An expanded `rtP` parameter structure that contains `idx` additional parameter sets defined by the `rsimsetrtpparam` function call
expanded `rtP` parameter structure

See Also

`rsimgetrtp`

Topics

“Create a MAT-File That Includes a Model Parameter Structure”
“Update Diagram and Run Simulation”
“Default parameter behavior”
“Block Authoring and Simulation Integration”
“Tune Parameters”

Introduced in R2009b

rtw_precompile_libs

Rebuild precompiled libraries within model without building model

Syntax

`rtw_precompile_libs(model,build_spec)`

Description

`rtw_precompile_libs(model,build_spec)` builds libraries within *`model`*, according to the *`build_spec`* field values, and places the libraries in a precompiled folder. Model builds that use the template makefile approach support the `rtw_precompile_libs` function. Toolchain approach model builds do not support the `rtw_precompile_libs` function.

Examples

Precompile Libraries for Model

Build the libraries in *`my_model`* without building *`my_model`*.

```
% Specify the library suffix
if isunix
   suffix = '_std.a';
elseif ismac
   suffix = '_std.a';
else
   suffix = '_vcx64.lib';
end
open_system(my_model);
set_param(my_model, 'TargetLibSuffix',suffix);

% Set the precompiled library folder
set_param(my_model, 'TargetPreCompLibLocation',fullfile(pwd,'lib'));

% Define a build specification that specifies
% the location of the files to compile.
my_build_spec = [];
my_build_spec.rtwmakecfgDirs = {fullfile(pwd,'src')};

% Build the libraries in 'my_model'
rtw_precompile_libs(my_model,my_build_spec);
```

Input Arguments

`model` — Model object or name for which to build libraries
`object` | *`'modelName'`*

Name of the model containing the libraries that you want to build.

`build_spec` — Structure with field values that provides the build specification
struct

Structure with fields that define a build specification. Fields except `rtwmakecfgDirs` are optional.

Field Values in build_spec

Specify the structure field values of the `build_spec`.

Example: `build_spec.rtwmakecfgDirs = {fullfile(pwd, 'src')};`

`rtwmakecfgDirs` — Fully qualified paths to the folders containing `rtwmakecfg` files for libraries to precompile
array of paths

Uses the `Name` and `Location` elements of `makeInfo.library`, as returned by the `rtwmakecfg` function, to specify name and location of precompiled libraries. If you use the `TargetPreCompLibLocation` parameter to specify the library folder, it overrides the `makeInfo.library.Location` setting.

The specified model must contain S-function blocks that use precompiled libraries, which the `rtwmakecfg` files specify. The makefile that the build approach generates contains the library rules only if the conversion requires the libraries.

Example: `build_spec.rtwmakecfgDirs = {fullfile(pwd, 'src')};`

`libSuffix` — Suffix, including the file type extension, to append to the name of each library (for example, `_std.a` or `_vcx64.lib`)
character vector

The suffix must include a period (.). Set the suffix by using either this field or the `TargetLibSuffix` parameter. If you specify a suffix with both mechanisms, the `TargetLibSuffix` setting overrides the setting of this field.

Example: `build_spec.libSuffix = '_vcx64.lib';`

`intOnlyBuild` — Selects library optimization
`'false'` (default) | `'true'`

When set to `true`, indicates that the function optimizes the libraries so that they compile from integer code only. Applies to ERT-based targets only.

Example: `build_spec.intOnlyBuild = 'false';`

`makeOpts` — Specifies an option for `rtwMake`
character vector

Specifies an option to include in the `rtwMake` command line.

Example: `build_spec.makeOpts = '';`

`addLibs` — Specifies libraries to build
cell array of structures

This cell array of structures specifies the libraries to build that an `rtwmakecfg` function does not specify. Define each structure with two fields that are character arrays:

- `libName` — Name of the library without a suffix
- `libLoc` — Location for the precompiled library

The build approach (toolchain approach or template makefile approach) lets you specify other libraries and how to build them. Use this field if you must precompile libraries.

Example: `build_spec.addLibs = 'libs_list';`

See Also

Topics
"Precompile S-Function Libraries"
"Recompile Precompiled Libraries"
"Choose Build Approach and Configure Build Process"
"Use rtwmakecfg.m API to Customize Generated Makefiles"

Introduced in R2009b

rtwbuild

(Not recommended) Build generated code from a model

Note `rtwbuild` is not recommended. Use `slbuild` instead.

Syntax

```
rtwbuild(model)
rtwbuild(model,Name,Value)

rtwbuild(subsystem)

rtwbuild(subsystem,'Mode','ExportFunctionCalls')
blockHandle = rtwbuild(subsystem,'Mode','ExportFunctionCalls')
```

Description

`rtwbuild(model)` generates code from `model` based on current model configuration parameter settings. If `model` is not already loaded into the MATLAB environment, `rtwbuild` loads it before generating code.

If you clear the **Generate code only** model configuration parameter, the function generates code and builds an executable image.

To reduce code generation time, when rebuilding a model, `rtwbuild` provides incremental model build. The code generator rebuilds a model or submodels only when they have changed since the most recent model build. To force a top-model build, see the `'ForceTopModelBuild'` argument.

`rtwbuild(model,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

`rtwbuild(subsystem)` generates code from `subsystem` based on current model configuration parameter settings. Before initiating the build, open (or load) the parent model.

If you clear the **Generate code only** model configuration parameter, the function generates code and builds an executable image.

`rtwbuild(subsystem,'Mode','ExportFunctionCalls')` generates code from `subsystem` that includes function calls that you can export to external application code if you have Embedded Coder.

`blockHandle = rtwbuild(subsystem,'Mode','ExportFunctionCalls')` returns the handle to a SIL block created for code generated from the specified subsystem if the **Create block** configuration parameter is set to `SIL` and if you have Embedded Coder. You can then use the SIL block for numerical equivalence testing.

Examples

Generate Code and Build Executable Image for Model

Generate C code for model rtwdemo_rtwintro.

```
rtwbuild('rtwdemo_rtwintro')
```

For the GRT system target file, the code generator produces the following code files and places them in folders rtwdemo_rtwintro_grt_rtw and slprj/grt/_sharedutils.

Model Files	Shared Files	Interface Files
rtwdemo_rtwintro.c rtwdemo_rtwintro.h rtwdemo_rtwintro_private.h rtwdemo_rtwintrotypes.h	rtGetInf.c rtGetInf.h rtGetNaN.c rtGetNaN.h rt_nonfinite.c rt_nonfinite.h rtwtypes.h multiword_types.h builtin_typeid_types.h	rtmodel.h

If the following model configuration parameters settings apply, the code generator produces additional results.

Parameter Setting	Results
Code Generation > **Generate code only** is cleared	Executable image rtwdemo_rtwintro.exe
Code Generation > **Report** > **Create code generation report** is selected	Report that provides information and links to generated code files, subsystem and code interface reports, entry-point functions, inports, outports, interface parameters, and data stores

Force Top Model Build

Generate code and build an executable image for rtwdemo_mdlreftop, which refers to model rtwdemo_mdlrefbot, regardless of model checksums and parameter settings.

```
rtwbuild('rtwdemo_mdlreftop', ...
    'ForceTopModelBuild',true)
```

Generate Code and Build Executable Image for Subsystem

Generate C code for subsystem Amplifier in model rtwdemo_rtwintro.

```
rtwbuild('rtwdemo_rtwintro/Amplifier')
```

For the GRT target, the code generator produces the following code files and places them in folders `Amplifier_grt_rtw` and `slprj/grt/_sharedutils`.

Model Files	Shared Files	Interface Files
Amplifier.c Amplifier.h Amplifier_private.h Amplifier_types.h	rtGetInf.c rtGetInf.h rtGetNaN.c rtGetNaN.h rt_nonfinite.c rt_nonfinite.h rtwtypes.h multiword_types.h builtin_typeid_types.h	rtmodel.h

If you apply the parameter settings listed in the table, the code generator produces the results listed.

Parameter Setting	Results
Code Generation > **Generate code only** is cleared	Executable image `Amplifier.exe`
Code Generation > **Report** > **Create code generation report** is selected	Report that provides information and links to generated code files, subsystem and code interface reports, entry-point functions, inports, outports, interface parameters, and data stores

Build Subsystem for Exporting Code to External Application

To export the image to external application code, build an executable image from a function-call subsystem.

```
rtwdemo_exporting_functions
rtwbuild('rtwdemo_exporting_functions/rtwdemo_subsystem','Mode','ExportFunctionCalls')
```

The executable image `rtwdemo_subsystem.exe` appears in your working folder.

Create SIL Block for Verification

From a function-call subsystem, create a SIL block that you can use to test the code generated from a model.

Open subsystem `rtwdemo_subsystem` in model `rtwdemo_exporting_functions` and set the **Create block** model configuration parameter to `SIL`.

Create the SIL block.

```
mysilblockhandle = rtwbuild('rtwdemo_exporting_functions/rtwdemo_subsystem',...
'Mode','ExportFunctionCalls')
```

The code generator produces a SIL block for the generated subsystem code. You can add the block to an environment or test harness model that supplies test vectors or stimulus input. You can then run simulations that perform SIL tests and verify that the generated code in the SIL block produces the same result as the original subsystem.

Name Exported Initialization Function

Name the initialization function generated when building an executable image from a function-call subsystem.

```
rtwdemo_exporting_functions
rtwbuild('rtwdemo_exporting_functions/rtwdemo_subsystem',...
'Mode','ExportFunctionCalls','ExportFunctionInitializeFunctionName','subsysinit')
```

The initialization function name `subsysinit` appears in `rtwdemo_subsystem_ert_rtw/ert_main.c`.

Display Status Information in Build Status Window

Display build information in the Build Status window while generating code and running a parallel build of model `rtwdemo_mdlreftop_witherr`.

```
rtwbuild('rtwdemo_mdlreftop_witherr', ...
    'OpenBuildStatusAutomatically',true)
```

Input Arguments

`model` — Model object or name for which to generate code or build an executable image

`object` | *`'modelName'`*

Model for which to generate code or build an executable image, specified as an object or a character vector representing the model name.

Example: `'rtwdemo_exporting_functions'`

`subsystem` — Subsystem name for which to generate code or build executable image

`'subsystemName'`

Subsystem for which to generate code or build an executable image, specified as a character vector representing the subsystem name or the full block path.

Example: `'rtwdemo_exporting_functions/rtwdemo_subsystem'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `rtwbuild('rtwdemo_mdlreftop','ForceTopModelBuild',true)`

`ForceTopModelBuild` — Force regeneration of top model code
`false` (default) | `true`

Force regeneration of top model code, specified as `true` or `false`.

Action	Specify
Force the code generator to regenerate code for the top model of a system that includes referenced models	`true`
Specify that the code generator determine whether to regenerate top model code based on model and model parameter changes	`false`

Consider forcing regeneration of code for a top model if you change items associated with external or custom code, such as code for a custom target. For example, set `ForceTopModelBuild` to `true` if you change:

- TLC code
- S-function source code, including `rtwmakecfg.m` files
- Integrated custom code

Alternatively, you can force regeneration of top model code by deleting folders in the code generation folder, such as `slprj` or the generated model code folder.

`generateCodeOnly` — Generate code only
`false` | `true`

If you do not specify a value, the **Generate code only** (`GenCodeOnly`) option on the **Code Generation** pane controls build process behavior.

If you specify a value, the argument overrides the **Generate code only** (`GenCodeOnly`) option on the **Code Generation** pane.

Action	Specify
Generate code only.	`true`
Generate code and build executable file.	`false`

`Mode` — Export function calls (for subsystem builds only)
`'ExportFunctionCalls'` | `'Normal'`

- `'ExportFunctionCalls'` –– If you have Embedded Coder, generates code from `subsystem` that includes function calls that you can export to external application code.
- `'Normal'` –– Does not export function calls.

`ExportFunctionInitializeFunctionName` — Function name
character vector

Name the exported initialization function for specified subsystem.

Example:
`rtwbuild(`*`subsystem`*`,'Mode','ExportFunctionCalls','ExportFunctionInitializeFunctionName',` *`fcnname`*`)`

`OpenBuildStatusAutomatically` — Display build information in the Build Status window
`false` (default) | `true`

Display build information in the Build Status window, specified as `true` or `false`. For more information about using the Build Status window, see “Monitor Parallel Building of Referenced Models”.

The Build Status window supports parallel builds of referenced model hierarchies. Do not use the Build Status window for serial builds.

Action	Specify
Display build information in the Build Status window	`true`
No action	`false`

`ObfuscateCode` — Generate obfuscated C code
false (default) | true

Specify whether to generate obfuscated C code, specified as `true` or `false`.

Action	Specify
Generate obfuscated C code that you can share with third parties with reduced likelihood of compromising intellectual property.	`true`
No action.	`false`

`IncludeModelReferenceSimulationTargets` — Option to build model reference simulation targets
`false` (default) | `true`

Option to build model reference simulation targets, specified as the comma-separated pair consisting of `'IncludeModelReferenceSimulationTargets'` and `true` or `false`.

Data Types: `logical`

Output Arguments

`blockHandle` — Handle to SIL block created for generated subsystem code
handle

Handle to SIL block created for generated subsystem code. Returned only if both of the following conditions apply:

- You are licensed to use Embedded Coder software.
- **Create block** model configuration parameter is set to `SIL`.

Tips

You can initiate code generation and the build process by:

- Pressing **Ctrl+B**.
- Selecting **Code** > **C/C++ Code** > **Build Model**.
- Invoking the `slbuild` command from the MATLAB command line.

Compatibility Considerations

`rtwbuild` does not generate model reference simulation targets by default

Behavior changed in R2020b

Starting in R2020b, the `rtwbuild` function does not generate model reference simulation targets by default. Excluding the model reference simulation targets allows for faster code generation for model hierarchies.

You can continue to generate both the simulation and code generation targets with the `rtwbuild` function by using the `IncludeModelReferenceSimulationTargets` argument.

Extended Capabilities

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To build referenced models in parallel, in the top model, select the configuration parameter check box **Enable parallel model reference builds**. For more information, see "Reduce Build Time for Referenced Models by Using Parallel Builds".

In Parallel Computing Toolbox™ commands, for example, a `parfor` or `spmd` loop, do not invoke `rtwbuild`, `rtwrebuild`, or `slbuild` commands that build models that are configured for parallel builds.

See Also

`codebuild` | `coder.buildstatus.close` | `coder.buildstatus.open` | `rtwrebuild` | `slbuild`

Topics

"Build and Run a Program"
"Choose Build Approach and Configure Build Process"
"Control Regeneration of Top Model Code"
"Generate Component Source Code for Export to External Code Base"
"Software-in-the-Loop Simulation"

Introduced in R2009a

RTW.BuildInfo

Provide information for compiling and linking generated code

Description

An `RTW.BuildInfo` object contains information for compiling and linking generated code.

Creation

Syntax

`buildInformation = RTW.BuildInfo`

Description

`buildInformation = RTW.BuildInfo` returns a build information object. You can use the object to specify information for compiling and linking generated code. For example:

- Compiler options
- Preprocessor identifier definitions
- Linker options
- Source files and paths
- Include files and paths
- Precompiled external libraries

Properties

`ComponentName` — Component name
character vector | string

Name of generated code component.

Object Functions

addCompileFlags	Add compiler options to build information
addDefines	Add preprocessor macro definitions to build information
addIncludeFiles	Add include files to build information
addIncludePaths	Add include paths to build information
addLinkFlags	Add link options to build information
addLinkObjects	Add link objects to build information
addNonBuildFiles	Add nonbuild-related files to build information
addSourceFiles	Add source files to build information
addSourcePaths	Add source paths to build information
addTMFTokens	Add template makefile (TMF) tokens to build information
findBuildArg	Find a specific build argument in build information

findIncludeFiles	Find and add include (header) files to build information
getBuildArgs	Get build arguments from build information
getCompileFlags	Get compiler options from build information
getDefines	Get preprocessor macro definitions from build information
getFullFileList	Get list of files from build information
getIncludeFiles	Get include files from build information
getIncludePaths	Get include paths from build information
getLinkFlags	Get link options from build information
getNonBuildFiles	Get nonbuild-related files from build information
getSourceFiles	Get source files from build information
getSourcePaths	Get source paths from build information
setTargetProvidesMain	Disable inclusion of code generator provided (generated or static) main.c source file during build
updateFilePathsAndExtensions	Update files in build information with missing paths and file extensions
updateFileSeparator	Update file separator character for file lists in build information

Examples

Retrieve Build Information Object

When you build generated code, the build process stores an `RTW.BuildInfo` object in the `buildInfo.mat` file. To retrieve the object, from the code generation folder that contains the `buildInfo.mat` file, run:

```
bi=load('buildInfo.mat');
bi.buildInfo
```

```
ans = 

  BuildInfo with properties:

          ComponentName: 'slexAircraftExample'
                 Viewer: []
                 Tokens: [27×1 RTW.BuildInfoKeyValuePair]
              BuildArgs: [13×1 RTW.BuildInfoKeyValuePair]
               MakeVars: []
               MakeArgs: ''
    TargetPreCompLibLoc: ''
        TargetLibSuffix: ''
              ModelRefs: []
                 SysLib: [1×1 RTW.BuildInfoModules]
                   Maps: [1×1 struct]
                LinkObj: []
                Options: [1×1 RTW.BuildInfoOptions]
                    Inc: [1×1 RTW.BuildInfoModules]
                    Src: [1×1 RTW.BuildInfoModules]
                  Other: [1×1 RTW.BuildInfoModules]
                   Path: []
               Settings: [1×1 RTW.BuildInfoSettings]
           DisplayLabel: 'Build Info'
                  Group: ''
```

The object contains build information.

Configure RTW.BuildInfo to Specify Code for Compilation

This example shows how to create an `RTW.BuildInfo` object and register source files.

Create an `RTW.BuildInfo` object.

```
buildInfo = RTW.BuildInfo;
```

Register source files.

```
buildInfo.ComponentName = 'foo1';
addSourceFiles(buildInfo, 'foo1.c');
```

Specify the build method and create a static library.

```
tmf = fullfile(tmffolder, 'ert_vcx64.tmf');
buildResult1 = codebuild(pwd, buildInfo, tmf)
```

See Also

Introduced in R2006a

RTW.getBuildDir

Get build folder information from model build information

Syntax

```
RTW.getBuildDir(model)
folderStruct = RTW.getBuildDir(model)
```

Description

`RTW.getBuildDir(model)` displays build folder information for model.

If the model is closed, the function opens and then closes the model, leaving it in its original state. If the model is large and closed, the `RTW.getBuildDir` function can take longer to execute.

`folderStruct = RTW.getBuildDir(model)` returns a structure containing build folder information.

You can use this function in automated scripts to determine the build folder in which the generated code for a model is placed.

This function can return build folder information for protected models.

Examples

Display Build Folder Information

Display build folder information for the model `'sldemo_fuelsys'`.

```
>> RTW.getBuildDir('sldemo_fuelsys')


ans =

              BuildDirectory: 'C:\work\modelref\sldemo_fuelsys_ert_rtw'
                 CacheFolder: 'C:\work\modelref'
               CodeGenFolder: 'C:\work\modelref'
            RelativeBuildDir: 'sldemo_fuelsys_ert_rtw'
              BuildDirSuffix: '_ert_rtw'
  ModelRefRelativeRootSimDir: 'slprj\sim'
  ModelRefRelativeRootTgtDir: 'slprj\ert'
    ModelRefRelativeBuildDir: 'slprj\ert\sldemo_fuelsys'
      ModelRefRelativeSimDir: 'slprj\sim\sldemo_fuelsys'
      ModelRefRelativeHdlDir: 'slprj\hdl\sldemo_fuelsys'
           ModelRefDirSuffix: ''
           SharedUtilsSimDir: 'slprj\sim\_sharedutils'
           SharedUtilsTgtDir: 'slprj\ert\_sharedutils'
```

Get Build Folder Information

Return a structure `my_folderStruct` that contains build folder information for the model `'MyModel'`.

```
>> my_folderStruct = RTW.getBuildDir('MyModel')


my_folderStruct =

              BuildDirectory: 'H:\MyModel_ert_rtw'
                 CacheFolder: 'H:\'
               CodeGenFolder: 'H:\'
            RelativeBuildDir: 'MyModel_ert_rtw'
              BuildDirSuffix: '_ert_rtw'
  ModelRefRelativeRootSimDir: 'slprj\sim'
  ModelRefRelativeRootTgtDir: 'slprj\ert'
    ModelRefRelativeBuildDir: 'slprj\ert\MyModel'
      ModelRefRelativeSimDir: 'slprj\sim\MyModel'
      ModelRefRelativeHdlDir: 'slprj\hdl\MyModel'
           ModelRefDirSuffix: ''
          SharedUtilsSimDir: 'slprj\sim\_sharedutils'
          SharedUtilsTgtDir: 'slprj\ert\_sharedutils'
```

Input Arguments

`model` — Model object or name for which to get the build folders
`object` | *`'modelName'`*

Model for which to get the build folder, specified as an object or a character vector representing the model name.

Example: `'sldemo_fuelsys'`

Output Arguments

`folderStruct` — Structure with field values that provide build folder information
struct

Structure with fields that provides build folder information.

Example: folderstruct = RTW.getBuildDir('MyModel')

`BuildDirectory` — Character vector specifying fully qualified path to build folder for model
character vector

`CacheFolder` — Character vector specifying root folder in which to place model build artifacts used for simulation
character vector

`CodeGenFolder` — Character vector specifying root folder in which to place code generation files
character vector

`RelativeBuildDir` — Character vector specifying build folder relative to the current working folder (`pwd`)
character vector

`BuildDirSuffix` — Character vector specifying suffix appended to model name to create build folder
character vector

`ModelRefRelativeRootSimDir` — Character vector specifying the relative root folder for the model reference target simulation folder
character vector

`ModelRefRelativeRootTgtDir` — Character vector specifying the relative root folder for the model reference target build folder
character vector

`ModelRefRelativeBuildDir` — Character vector specifying model reference target build folder relative to current working folder (`pwd`)
character vector

`ModelRefRelativeSimDir` — Character vector specifying model reference target simulation folder relative to current working folder (`pwd`)
character vector

`ModelRefRelativeHdlDir` — Character vector specifying model reference target HDL folder relative to current working folder (`pwd`)
character vector

`ModelRefDirSuffix` — Character vector specifying suffix appended to system target file name to create model reference build folder
character vector

`SharedUtilsSimDir` — Character vector specifying the shared utility folder for simulation
character vector

`SharedUtilsTgtDir` — Character vector specifying the shared utility folder for code generation
character vector

See Also
`slbuild`

Topics
"Working Folder"
"Manage Build Process Folders"

Introduced in R2008b

rtwrebuild

Rebuild generated code from model

Syntax

```
rtwrebuild()
```

```
rtwrebuild(model)
```

```
rtwrebuild(path)
```

Description

`rtwrebuild()` assumes that the current working folder is the build folder of the model (not the model location) and calls `codebuild`. If the current working folder is not the build folder, the function exits with an error.

`rtwrebuild` calls `codebuild` to recompile files you modified since that build. Operation of this function depends on the current working folder, not the current loaded model. If your model includes referenced models, `codebuild` recompiles code for all models in the hierarchy.

In Parallel Computing Toolbox commands, for example, a `parfor` or `spmd` loop, do not invoke `rtwbuild`, `rtwrebuild`, or `slbuild` commands that build models that are configured for parallel builds. For information about parallel builds of referenced models, see “Reduce Build Time for Referenced Models by Using Parallel Builds”.

`rtwrebuild(model)` assumes that the current working folder is one level above the build folder and calls `codebuild`. If the current working folder (`pwd`) is not one level above the build folder, the function exits with an error.

`rtwrebuild(path)` finds the build folder indicated with the *`path`* argument. The *`path`* argument syntax lets the function operate without regard to the relationship between the current working folder and the build folder of the model.

Examples

Rebuild Code from Build Folder

Call `codebuild` and recompile code when the current working folder is the build folder. For example,

- If the model name is `mymodel`
- And, if the model build was initiated in the `C:\work` folder
- And, if the system target is GRT

```
rtwrebuild()
```

Rebuild Code from Parent Folder of Build Folder

When the current working folder is one level above the build folder, call `codebuild` to recompile code.

```
rtwrebuild('mymodel')
```

Rebuild Code from a Folder

Recompile code from a current folder by specifying a path to the model build folder, `C:\work\mymodel_grt_rtw`.

```
rtwrebuild(fullfile('C:','work','mymodel_grt_rtw'))
```

Input Arguments

`model` — Model object or name for which to regenerate code or rebuild an executable image
`object` | *`'modelName'`*

Model for which to regenerate code or rebuild an executable image, specified as an object or a character vector representing the model name.

Example: `'rtwdemo_exporting_functions'`

`path` — Model path object or fully qualified path to the build folder for the model for which to regenerate code or rebuild an executable image
`object` | *`modelPath`*

Example: `fullfile('C:','work','mymodel_grt_rtw')`

See Also

`codebuild` | `rtwbuild` | `slbuild`

Topics

"Rebuild a Model"

Introduced in R2009a

rtwreport

(To be removed) Create generated code report for model with Simulink Report Generator

Note `rtwreport` will be removed in a future release. Use `coder.report.generate` instead. For more information, see "Compatibility Considerations".

Syntax

```
rtwreport(model)
rtwreport(model,folder)
```

Description

`rtwreport(model)` creates a report of code generation information for a model. Before creating the report, the function loads the model and generates code. The code generator names the report `codegen.html`. It places the file in your current folder. The report includes:

- Snapshots of the model, including subsystems.
- Block execution order list.
- Code generation summary with a list of generated code files, configuration settings, a subsystem map, and a traceability report.
- Full listings of generated code that reside in the build folder.

`rtwreport(model,folder)` specifies the build folder, *`model_target`*`_rtw`. The build folder (`folder`) and `slprj` folder must reside in the code generation folder. If the software cannot find the `folder`, an error occurs and code is not generated.

Examples

Create Report Specifying Build Folder

Create a report for model `rtwdemo_secondOrderSystem` using build folder, `rtwdemo_secondOrderSystem_grt_rtw`:

```
rtwreport('rtwdemo_secondOrderSystem', ...
    'rtwdemo_secondOrderSystem_grt_rtw');
```

Input Arguments

`model` — Model name

character vector

Model name for which the report is generated, specified as a character vector.

Example: `'rtwdemo_secondOrderSystem'`

Data Types: `char`

`folder` — Build folder name
character vector

Build folder name, specified as a character vector. When you have multiple build folders, include a folder name. For example, if you have multiple builds using different targets, such as GRT and ERT.

Example: `'rtwdemo_secondOrderSystem_grt_rtw'`

Data Types: `char`

Compatibility Considerations

rtwreport function will be removed
Warns starting in R2021a

The function `rtwreport` will be removed in a future release. Use `coder.report.generate` instead.

To update your code, change instances of the function name `rtwreport` to `coder.report.generate`. You do not need to change the input arguments.

Unlike the `rtwreport` function, the `coder.report.generate` function provides additional input options that you can use to configure the generated report. However, the `coder.report.generate` function does not include snapshots of the model or the block execution order list.

See Also
`coder.report.generate`

Topics
"Document Generated Code with Simulink Report Generator"
`Import Generated Code` (Simulink Report Generator)
"Working with the Report Explorer" (Simulink Report Generator)
`Code Generation Summary` (Simulink Report Generator)

Introduced in R2007a

rtwtrace

Trace a block to generated code in code generation report

Syntax

```
rtwtrace('blockpath')
rtwtrace('Simulink_identifier')
rtwtrace('blockpath', 'hdl')
rtwtrace('blockpath', 'plc')
```

Description

`rtwtrace('blockpath')` opens a code generation report that displays contents of the source code file and highlights the line of code corresponding to the specified block.

Before calling `rtwtrace`, make sure that:

- You select an ERT-based model and enable model to code navigation.

  In the Configuration Parameters dialog box, select the **Model-to-code** on page 12-7 parameter.
- You generate code for the model by using the code generator.
- Your build folder is under the current working folder. Otherwise, `rtwtrace` might produce an error.

`rtwtrace('Simulink_identifier')` opens a code generation report that displays contents of the source code file and highlights the line of code corresponding to the block identified by the Simulink identifier (SID). SID is a unique designation for each block or element in the model. For more information, see “Simulink Identifiers”.

`rtwtrace('blockpath', 'hdl')` opens a code generation report in HDL Coder™ that displays contents of the source code file and highlights the line of code corresponding to the specified block.

`rtwtrace('blockpath', 'plc')` opens a code generation report in Simulink PLC Coder™ that displays contents of the source code file and highlights the line of code corresponding to the specified block.

Examples

Display Generated Code for a Block

Display the generated code for block `Out1` in the model `rtwdemo_comments` in code generation report:

```
% Using block path
rtwtrace('rtwdemo_comments/Out1')

% Using Simulink identifier
rtwtrace('rtwdemo_comments:33')
```

Code Generation Report

Find: Match Case

Contents

Summary
Subsystem Report
Code Interface Report
Traceability Report
Static Code Metrics Report
Code Replacements Report
Coder Assumptions

Generated Code

[-] **Main file**
ert_main.c

[-] **Model files**
rtwdemo_comments.c (6)
rtwdemo_comments.h (1)

[+] **Shared files (1)**

Highlight code for block:
'<Root>/Out1'

<< < 1 of 7 > >> Remove Highlights

```
112   /* Gateway: Stateflow */
113   /* During: Stateflow */
114   if (rtDWork.bitsForTID0.is_active_c1_rtwdemo_comments == 0U) {
115     /* Entry: Stateflow */
116     rtDWork.bitsForTID0.is_active_c1_rtwdemo_comments = 1;
117
118     /* Entry Internal: Stateflow */
119     /* Transition: '<S2>:9' */
120     rtDWork.bitsForTID0.is_c1_rtwdemo_comments = IN_First;
121
122     /* Outport: '<Root>/Out1' */
123     /* Entry 'First': '<S2>:2' */
124     rtY.Out1 = 1;
125   } else if (rtDWork.bitsForTID0.is_c1_rtwdemo_comments == IN_First) {
126     /* During 'First': '<S2>:2' */
127     /* The state is activated at initialization. */
128     /* Transition: '<S2>:10' */
129     /*  After initialize */
130     /* After initialization, transfer to the normal state. */
131     rtDWork.bitsForTID0.is_c1_rtwdemo_comments = IN_Normal;
132
133     /* Entry Internal 'Normal': '<S2>:1' */
134     /* Transition: '<S2>:4' */
135     if (is_data_xor) {
136       /* Outport: '<Root>/Out1' */
137       /* Transition: '<S2>:5' */
138       /* This transition is taken when exactly one input is enabled. */
139       /* Transition: '<S2>:6' */
140       rtY.Out1 = (int32_T)indexed_data;
```

OK Help

Input Arguments

blockpath — block path

character vector (default)

`blockpath` is a character vector enclosed in quotes specifying the full Simulink block path, for example, '*model_name/block_name*'.

Example: 'rtwdemo_comments/Out1'

Data Types: char

Simulink_identifier — Simulink identifier

character vector (default)

`Simulink_identifier` is a character vector enclosed in quotes specifying the Simulink identifier, for example, '*model_name:number*'.

Example: 'rtwdemo_comments:33'

Data Types: char

hdl — HDL Coder
character vector

`hdl` is a character vector enclosed in quotes specifying that the code report is from HDL Coder.

Example: `'Out1'`

Data Types: `char`

plc — PLC Coder
character vector

`plc` is a character vector enclosed in quotes specifying that the code report is from Simulink PLC Coder.

Example: `'Out1'`

Data Types: `char`

Alternatives

To trace from a block in the model diagram, right-click a block and select **C/C++ Code > Navigate to C/C++ Code**.

See Also

Topics
"Verify Generated Code by Using Code Tracing"
"Model-to-Code Traceability"
"Model-to-code" on page 12-7

Introduced in R2009b

setTargetProvidesMain

Disable inclusion of code generator provided (generated or static) `main.c` source file during build

Syntax

```
setTargetProvidesMain(buildinfo,providesmain)
```

Description

`setTargetProvidesMain(buildinfo,providesmain)` disables the code generator from including a sample `main.c` source file.

To replace the sample `main.c` file from the code generator with a custom `main.c` file, call the `setTargetProvidesMain` function during the `'after_tlc'` case in the `ert_make_rtw_hook.m` or `grt_make_rtw_hook.m` file.

Examples

Workflow for `setTargetProvidesMain`

To apply the `setTargetProvidesMain` function:

Add `buildInfo` to the arguments in the function call.

```
function ert_make_rtw_hook(hookMethod,Name,rtwroot, ...
   templateMakefile,buildOpts,buildArgs,buildInfo)
```

Add the `setTargetProvidesMain` function to the `'after_tlc'` stage.

```
case 'after_tlc'
 % Called just after to invoking TLC Compiler (actual code generation.)
 % Valid arguments at this stage are hookMethod, Name, and
 % buildArgs, buildInfo
 %
 setTargetProvidesMain(buildInfo,true);
```

Use the **Configuration Parameters** > **Code Generation** > **Custom Code** > **Source Files** field to add your custom `main.c` to the . When you indicate that the target provides `main.c`, the requires this file to build without errors.

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo`
object

`providesmain` — Logical value that specifies whether the code generator includes the target provided `main.c` file
false (default) | true

The *`providesmain`* argument specifies whether the code generator includes a (generated or static) `main.c` source file.

- `false` — The code generator includes a sample `main.obj` object file.
- `true` — The target provides the `main.c` source file.

See Also

`addSourceFiles` | `addSourcePaths`

Topics

“Customize Build Process with STF_make_rtw_hook File”

Introduced in R2009a

Simulink.fileGenControl

Specify root folders for files generated by diagram updates and model builds

Syntax

```
cfg = Simulink.fileGenControl('getConfig')
Simulink.fileGenControl(Action,Name,Value)
```

Description

`cfg = Simulink.fileGenControl('getConfig')` returns a handle to an instance of the `Simulink.FileGenConfig` object, which contains the current values of these file generation control parameters:

- `CacheFolder` – Specifies the root folder for model build artifacts that are used for simulation, including Simulink® cache files.
- `CodeGenFolder` – Specifies the root folder for code generation files.
- `CodeGenFolderStructure` – Controls the folder structure within the code generation folder.

To get or set the parameter values, use the `Simulink.FileGenConfig` object.

These Simulink preferences determine the initial parameter values for the MATLAB session:

- Simulation cache folder – `CacheFolder`
- Code generation folder – `CodeGenFolder`
- Code generation folder structure – `CodeGenFolderStructure`

`Simulink.fileGenControl(Action,Name,Value)` performs an action that uses the file generation control parameters of the current MATLAB session. Specify additional options with one or more `name,value` pair arguments.

Examples

Get File Generation Control Parameter Values

To obtain the file generation control parameter values for the current MATLAB session, use `getConfig`.

```
cfg = Simulink.fileGenControl('getConfig');

myCacheFolder = cfg.CacheFolder;
myCodeGenFolder = cfg.CodeGenFolder;
myCodeGenFolderStructure = cfg.CodeGenFolderStructure;
```

Set File Generation Control Parameters by Using `Simulink.FileGenConfig` Object

To set the file generation control parameter values for the current MATLAB session, use the `setConfig` action. First, set values in an instance of the `Simulink.FileGenConfig` object. Then,

pass the object instance. This example assumes that your system has aNonDefaultCacheFolder and aNonDefaultCodeGenFolder folders.

```
% Get the current configuration
cfg = Simulink.fileGenControl('getConfig');

% Change the parameters to non-default locations
% for the cache and code generation folders
cfg.CacheFolder = fullfile('C:','aNonDefaultCacheFolder');
cfg.CodeGenFolder = fullfile('C:','aNonDefaultCodeGenFolder');
cfg.CodeGenFolderStructure = 'TargetEnvironmentSubfolder';

Simulink.fileGenControl('setConfig', 'config', cfg);
```

Set File Generation Control Parameters Directly

You can set file generation control parameter values for the current MATLAB session without creating an instance of the Simulink.FileGenConfig object. This example assumes that your system has aNonDefaultCacheFolder and aNonDefaultCodeGenFolder folders.

```
myCacheFolder = fullfile('C:','aNonDefaultCacheFolder');
myCodeGenFolder = fullfile('C:','aNonDefaultCodeGenFolder');

Simulink.fileGenControl('set', 'CacheFolder', myCacheFolder, ...
   'CodeGenFolder', myCodeGenFolder, ...
   'CodeGenFolderStructure', ...
    Simulink.filegen.CodeGenFolderStructure.TargetEnvironmentSubfolder);
```

If you do not want to generate code for different target environments in separate folders, for 'CodeGenFolderStructure', specify the value Simulink.filegen.CodeGenFolderStructure.ModelSpecific.

Reset File Generation Control Parameters

You can reset the file generation control parameters to values from Simulink preferences.

```
Simulink.fileGenControl('reset');
```

Create Simulation Cache and Code Generation Folders

To create file generation folders, use the set action with the 'createDir' option. You can keep previous file generation folders on the MATLAB path through the 'keepPreviousPath' option.

```
%
myCacheFolder = fullfile('C:','aNonDefaultCacheFolder');
myCodeGenFolder = fullfile('C:','aNonDefaultCodeGenFolder');

Simulink.fileGenControl('set', ...
   'CacheFolder',myCacheFolder, ...
   'CodeGenFolder',myCodeGenFolder, ...
```

```
'keepPreviousPath',true, ...
'createDir',true);
```

Input Arguments

`Action` — Specify action

'reset' | 'set' | 'setConfig'

Specify an action that uses the file generation control parameters of the current MATLAB session:

- `'reset'` – Reset file generation control parameters to values from Simulink preferences.
- `'set'` – Set file generation control parameters for the current MATLAB session by directly passing values.
- `'setConfig'` – Set file generation control parameters for the current MATLAB session by using an instance of a `Simulink.FileGenConfig` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `Simulink.fileGenControl(`*`Action`*`, `*`Name`*`, `*`Value`*`);`

`config` — Specify instance of `Simulink.FileGenConfig`

object handle

Specify the `Simulink.FileGenConfig` object instance containing file generation control parameters that you want to set.

Option for `setConfig`.

Example: `Simulink.fileGenControl('setConfig', 'config', `*`cfg`*`);`

`CacheFolder` — Specify simulation cache folder

character vector

Specify a simulation cache folder path value for the `CacheFolder` parameter.

Option for `set`.

Example: `Simulink.fileGenControl('set', 'CacheFolder', `*`myCacheFolder`*`);`

`CodeGenFolder` — Specify code generation folder

character vector

Specify a code generation folder path value for the `CodeGenFolder` parameter. You can specify an absolute path or a path relative to build folders. For example:

- `'C:\Work\mymodelsimcache'` and `'/mywork/mymodelgencode'` specify absolute paths.
- `'mymodelsimcache'` is a path relative to the current working folder (`pwd`). The software converts a relative path to a fully qualified path at the time the `CacheFolder` or `CodeGenFolder` parameter is set. For example, if `pwd` is `'/mywork'`, the result is `'/mywork/mymodelsimcache'`.

- `'../test/mymodelgencode'` is a path relative to pwd. If pwd is `'/mywork'`, the result is `'/test/mymodelgencode'`.

Option for `set`.

Example: `Simulink.fileGenControl('set', 'CodeGenFolder', myCodeGenFolder);`

`CodeGenFolderStructure` — Specify generated code folder structure
`Simulink.filegen.CodeGenFolderStructure.ModelSpecific` (default) | `Simulink.filegen.CodeGenFolderStructure.TargetEnvironmentSubfolder`

Specify the layout of subfolders within the generated code folder:

- `Simulink.filegen.CodeGenFolderStructure.ModelSpecific` (default) – Place generated code in subfolders within a model-specific folder.
- `Simulink.filegen.CodeGenFolderStructure.TargetEnvironmentSubfolder` – If models are configured for different target environments, place generated code for each model in a separate subfolder. The name of the subfolder corresponds to the target environment.

Option for `set`.

Example: `Simulink.fileGenControl('set', 'CacheFolder', myCacheFolder, ... 'CodeGenFolder', myCodeGenFolder, ... 'CodeGenFolderStructure', ... Simulink.filegen.CodeGenFolderStructure.TargetEnvironmentSubfolder);`

`keepPreviousPath` — Keep previous folder paths on MATLAB path
false (default) | true

Specify whether to keep the previous values of `CacheFolder` and `CodeGenFolder` on the MATLAB path:

- `true` – Keep previous folder path values on MATLAB path.
- `false` (default) – Remove previous older path values from MATLAB path.

Option for `reset`, `set`, or `setConfig`.

Example: `Simulink.fileGenControl('reset', 'keepPreviousPath', true);`

`createDir` — Create folders for file generation
false (default) | true

Specify whether to create folders for file generation if the folders do not exist:

- `true` – Create folders for file generation.
- `false` (default) – Do not create folders for file generation.

Option for `set` or `setConfig`.

Example: `Simulink.fileGenControl('set', 'CacheFolder', myCacheFolder, 'CodeGenFolder', myCodeGenFolder, 'keepPreviousPath', true, 'createDir',true);`

Avoid Naming Conflicts

Using `Simulink.fileGenControl` to set `CacheFolder` and `CodeGenFolder` adds the specified folders to your MATLAB search path. This function has the same potential for introducing a naming

conflict as using addpath to add folders to the search path. For example, a naming conflict occurs if the folder that you specify for CacheFolder or CodeGenFolder contains a model file with the same name as an open model. For more information, see “What Is the MATLAB Search Path?” and “Files and Folders that MATLAB Accesses”.

To use a nondefault location for the simulation cache folder or code generation folder:

1. Delete any potentially conflicting artifacts that exist in:
   - The current working folder, pwd.
   - The nondefault simulation cache and code generation folders that you intend to use.
2. Specify the nondefault locations for the simulation cache and code generation folders by using Simulink.fileGenControl or Simulink preferences.

Output Arguments

cfg — Current values of file generation control parameters
object handle

Instance of a Simulink.FileGenConfig object, which contains the current values of file generation control parameters.

See Also
“Simulation cache folder” | “Code generation folder” | Code generation folder structure

Topics
“Manage Build Process Folders”
“Share Simulink Cache Files for Faster Simulation”

Introduced in R2010b

Simulink.ModelReference.modifyProtectedModel

Modify existing protected model

Syntax

```
Simulink.ModelReference.modifyProtectedModel(model)
Simulink.ModelReference.modifyProtectedModel(model,Name,Value)

[harnessHandle] = Simulink.ModelReference.modifyProtectedModel(model,'
Harness',true)
[~ ,neededVars] = Simulink.ModelReference.modifyProtectedModel(model)
```

Description

`Simulink.ModelReference.modifyProtectedModel(model)` modifies options for an existing protected model created from the specified `model`. If `Name,Value` pair arguments are not specified, the modified protected model is updated with default values and supports only simulation.

`Simulink.ModelReference.modifyProtectedModel(model,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. These options are the same options that are provided by the `Simulink.ModelReference.protect` function. However, these options have additional options to change encryption passwords for read-only view, simulation, and code generation. When you add functionality to the protected model or change encryption passwords, the unprotected model must be available. The software searches for the model on the MATLAB path. If the model is not found, the software reports an error.

`[harnessHandle] = Simulink.ModelReference.modifyProtectedModel(model,' Harness',true)` creates a harness model for the protected model. It returns the handle of the harnessed model in `harnessHandle`.

`[~ ,neededVars] = Simulink.ModelReference.modifyProtectedModel(model)` returns a cell array that includes the names of base workspace variables used by the protected model.

Examples

Update Protected Model with Default Values

Create a modifiable protected model with support for code generation, then reset it to default values.

Add the password for when a protected model is modified. If you skip this step, you are prompted to set a password when a modifiable protected model is created.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Create a modifiable protected model with support for code generation and Web view.

```
Simulink.ModelReference.protect('sldemo_mdlref_counter','Mode',...
'CodeGeneration','Modifiable',true,'Report',true);
```

Provide the password to modify the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Modify the model to use default values.

```
Simulink.ModelReference.modifyProtectedModel(...
'sldemo_mdlref_counter');
```

The resulting protected model is updated with default values and supports only simulation.

Remove Functionality from Protected Model

Create a modifiable protected model with support for code generation and Web view, then modify it to remove the Web view support.

Add the password for when a protected model is modified. If you skip this step, you are prompted to set a password when a modifiable protected model is created.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Create a modifiable protected model with support for code generation and Web view.

```
Simulink.ModelReference.protect('sldemo_mdlref_counter','Mode',...
'CodeGeneration','Webview',true,'Modifiable',true,'Report',true);
```

Provide the password to modify the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Remove support for Web view from the protected model that you created.

```
Simulink.ModelReference.modifyProtectedModel(...
'sldemo_mdlref_counter', 'Mode', 'CodeGeneration','Report',true);
```

Change Encryption Password for Code Generation

Change an encryption password for a modifiable protected model.

Add the password for when a protected model is modified. If you skip this step, you are prompted to set a password when a modifiable protected model is created.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Add the password that the protected model user must provide to generate code.

```
Simulink.ModelReference.ProtectedModel.setPasswordForSimulation(...
'sldemo_mdlref_counter','cgpassword');
```

Create a modifiable protected model with a report and support for code generation with encryption.

```
Simulink.ModelReference.protect('sldemo_mdlref_counter','Mode',...
'CodeGeneration','Encrypt',true,'Modifiable',true,'Report',true);
```

Provide the password to modify the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Change the encryption password for simulation.

```
Simulink.ModelReference.modifyProtectedModel(
'sldemo_mdlref_counter','Mode','CodeGeneration','Encrypt',true,...
'Report',true,'ChangeSimulationPassword',...
{'cgpassword','new_password'});
```

Add Harness Model for Protected Model

Add a harness model for an existing protected model.

Add the password for when a protected model is modified. If you skip this step, you are prompted to set a password when a modifiable protected model is created.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Create a modifiable protected model with a report and support for code generation with encryption.

```
Simulink.ModelReference.protect('sldemo_mdlref_counter','Mode',...
'CodeGeneration','Modifiable',true,'Report',true);
```

Provide the password to modify the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Add a harness model for the protected model.

```
[harnessHandle] = Simulink.ModelReference.modifyProtectedModel(...
'sldemo_mdlref_counter','Mode','CodeGeneration','Report',true,...
'Harness',true);
```

Input Arguments

`model` — Model name

string or character vector (default)

Model name, specified as a string or character vector. It contains the name of a model or the path name of a Model block that references the protected model.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Mode','CodeGeneration','OutputFormat','Binaries','ObfuscateCode',true` specifies that obfuscated code be generated for the protected model. It also specifies that only binary files and headers in the generated code be visible to users of the protected model.

General

`Path` — Folder for protected model
current working folder (default) | string or character vector

Folder for protected model, specified as a string or character vector.

Example: `'Path','C:\Work'`

`Report` — Option to generate a report
false (default) | true

Option to generate a report, specified as a Boolean value.

To view the report, right-click the protected-model badge icon and select **Display Report**. Or, call the `Simulink.ProtectedModel.open` function with the `report` option.

The report is generated in HTML format. It includes information on the environment, functionality, license requirements, and interface for the protected model.

Example: `'Report',true`

`hdl` — Option to generate HDL code
false (default) | true

Option to generate HDL code, specified as a Boolean value.

This option requires HDL Coder license. When you enable this option, make sure that you specify the **Mode**. You can set this option to `true` in conjunction with the **Mode** set to `CodeGeneration` to enable both C code and HDL code generation support for the protected model.

If you want to enable only simulation and HDL code generation support, but not C code generation, set **Mode** to `HDLCodeGeneration`. You do not have to set the **hdl** option to `true`.

Example: `'hdl',true`

`Harness` — Option to create a harness model
false (default) | true

Option to create a harness model, specified as a Boolean value.

Example: `'Harness',true`

`CustomPostProcessingHook` — Option to add postprocessing function for protected model files
function handle

Option to add a postprocessing function for protected model files, specified as a function handle. The function accepts a `Simulink.ModelReference.ProtectedModel.HookInfo` object as an input variable. This object provides information on the source code files and other files generated during protected model creation. The object also provides information on exported symbols that you must not modify. Prior to packaging the protected model, the postprocessing function is called.

For a protected model with a top model interface, the `Simulink.ModelReference.ProtectedModel.HookInfo` object cannot provide information on exported symbols.

Example: `'CustomPostProcessingHook',@(protectedMdlInf)myHook(protectedMdlInf)`

Functionality

`Mode` — Model protection mode

`'Normal'` (default) | `'Accelerator'` | `'CodeGeneration'` | `'HDLCodeGeneration'` | `'ViewOnly'`

Model protection mode. Specify one of the following values:

- `'Normal'`: If the top model is running in `'Normal'` mode, the protected model runs as a child of the top model.
- `'Accelerator'`: The top model can run in `'Normal'`, `'Accelerator'`, or `'Rapid Accelerator'` mode.
- `'CodeGeneration'`: The top model can run in `'Normal'`, `'Accelerator'`, or `'Rapid Accelerator'` mode and support code generation.
- `'HDLCodeGeneration'`: The top model can run in `'Normal'`, `'Accelerator'`, or `'Rapid Accelerator'` mode and support HDL code generation.
- `'ViewOnly'`: Turns off Simulate and Generate code functionality modes. Turns on the read-only view mode.

Example: `'Mode','Accelerator'`

`OutputFormat` — Protected code visibility

`'CompiledBinaries'` (default) | `'MinimalCode'` | `'AllReferencedHeaders'`

Note This argument affects the output only when you specify `Mode` as `'Accelerator'` or `'CodeGeneration`. When you specify `Mode` as `'Normal'`, only a MEX-file is part of the output package.

Protected code visibility. This argument determines what part of the code generated for a protected model is visible to users. Specify one of the following values:

- `'CompiledBinaries'`: Only binary files and headers are visible.
- `'MinimalCode'`: Includes only the minimal header files required to build the code with the chosen build settings. Code in the build folder is visible. Users can inspect the code in the protected model report and recompile it for their purposes.
- `'AllReferencedHeaders'`: Includes header files found on the include path. Code in the build folder is visible. Header files referenced by the code are also visible.

Example: `'OutputFormat','AllReferencedHeaders'`

`ObfuscateCode` — Option to obfuscate generated code

true (default) | false

Option to obfuscate generated code, specified as a Boolean value. Applicable only when code generation is enabled for the protected model. Obfuscation is not supported for HDL code generation.

Example: `'ObfuscateCode',true`

`Webview` — Option to include a Web view

false (default) | true

Option to include a read-only view of protected model, specified as a Boolean value.

To open the Web view of a protected model, use one of the following methods:

- Right-click the protected-model badge icon and select **Show Web view**.
- Use the `Simulink.ProtectedModel.open` function. For example, to display the Web view for protected model `sldemo_mdlref_counter`, you can call:

  ```
  Simulink.ProtectedModel.open('sldemo_mdlref_counter', 'webview');
  ```
- Double-click the `.slxp` protected model file in the Current Folder browser.
- In the Block Parameter dialog box for the protected model, click **Open Model**.

Example: `'Webview',true`

Encryption

`ChangeSimulationPassword` — Option to change the encryption password for simulation

cell array of two character vectors

Option to change the encryption password for simulation, specified as a cell array of two character vectors. The first vector is the old password, the second vector is the new password.

Example: `'ChangeSimulationPassword',{'old_password','new_password'}`

`ChangeViewPassword` — Option to change the encryption password for read-only view

cell array of two character vectors

Option to change the encryption password for read-only view, specified as a cell array of two character vectors. The first vector is the old password, the second vector is the new password.

Example: `'ChangeViewPassword',{'old_password','new_password'}`

`ChangeCodeGenerationPassword` — Option to change the encryption password for code generation

cell array of two character vectors

Option to change the encryption password for code generation, specified as a cell array of two character vectors. The first vector is the old password, the second vector is the new password.

Example: `'ChangeCodeGenerationPassword',{'old_password','new_password'}`

`Encrypt` — Option to encrypt protected model

false (default) | true

Option to encrypt a protected model, specified as a Boolean value. Applicable when you have specified a password during protection, or by using the following methods:

- Password for read-only view of model: `Simulink.ModelReference.ProtectedModel.setPasswordForView`
- Password for simulation: `Simulink.ModelReference.ProtectedModel.setPasswordForSimulation`

- Password for code generation: `Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration`
- Password for HDL code generation: `Simulink.ModelReference.ProtectedModel.setPasswordForHDLCodeGeneration`

Example: `'Encrypt',true`

Output Arguments

`harnessHandle` — Handle of the harness model

double

Handle of the harness model, returned as a double or `0`, depending on the value of `Harness`.

If `Harness` is `true`, the value is the handle of the harness model; otherwise, the value is `0`.

`neededVars` — Names of base workspace variables

cell array

Names of base workspace variables used by the protected model, returned as a cell array.

The cell array can also include variables that the protected model does not use.

See Also

`Simulink.ModelReference.ProtectedModel.setPasswordForModify` | `Simulink.ModelReference.protect`

Introduced in R2014b

Simulink.ModelReference.protect

Obscure referenced model contents to hide intellectual property

Syntax

```
Simulink.ModelReference.protect(model)
Simulink.ModelReference.protect(model,Name,Value)
```

```
[harnessHandle] = Simulink.ModelReference.protect(model,'Harness',true)
[~ ,neededVars] = Simulink.ModelReference.protect(model)
```

Description

`Simulink.ModelReference.protect(model)` creates a protected model from the specified `model`. It places the protected model in the current working folder. The protected model has the same name as the source model. It has the extension `.slxp`.

`Simulink.ModelReference.protect(model,Name,Value)` uses additional options specified by one or more name-value pair arguments.

`[harnessHandle] = Simulink.ModelReference.protect(model,'Harness',true)` creates a harness model for the protected model. It returns the handle of the harnessed model in `harnessHandle`.

`[~ ,neededVars] = Simulink.ModelReference.protect(model)` returns a cell array that includes the names of base workspace variables used by the protected model.

Examples

Protect Referenced Model

Protect a referenced model and place the protected model in the current working folder.

```
sldemo_mdlref_bus;
model= 'sldemo_mdlref_counter_bus'

Simulink.ModelReference.protect(model);
```

A protected model named `sldemo_mdlref_counter_bus.slxp` is created. The protected model file is placed in the current working folder.

Place Protected Model in Specified Folder

Protect a referenced model and place the protected model in a specified folder.

```
sldemo_mdlref_bus;
model= 'sldemo_mdlref_counter_bus'

Simulink.ModelReference.protect(model,'Path','C:\Work');
```

A protected model named sldemo_mdlref_counter_bus.slxp is created. The protected model file is placed in C:\Work.

Generate Code for Protected Model

Protect a referenced model, generate code for it in normal mode, and obfuscate the code.

```
sldemo_mdlref_bus;
model= 'sldemo_mdlref_counter_bus'

Simulink.ModelReference.protect(model,'Path','C:\Work','Mode','CodeGeneration',...
'ObfuscateCode',true);
```

A protected model named sldemo_mdlref_counter_bus.slxp is created. The protected model file is placed in the C:\Work folder. The protected model runs as a child of the parent model. The code generated for the protected model is obfuscated by the software.

Generate HDL Code for Protected Model

Protect a referenced model, and generate HDL code for it in normal mode.

```
parent_model= 'hdlcoder_protected_model_parent_harness';
reference_model_to_protect = 'hdlcoder_referenced_model_gain';

Simulink.ModelReference.protect(reference_model_to_protect, ...
                        'Mode','HDLCodeGeneration')
```

A protected model named hdlcoder_referenced_model_gain.slxp is created. The protected model file is placed in the same folder as the parent model and the referenced model. The protected model runs as a child of the parent model.

Set the **hdl** option to true with **Mode** set to CodeGeneration to enable both C code generation and HDL code generation support for a protected model that you create.

```
parent_model= 'hdlcoder_protected_model_parent_harness';
reference_model_to_protect = 'hdlcoder_referenced_model_gain';

Simulink.ModelReference.protect(reference_model_to_protect, ...
                        'Mode','CodeGeneration','hdl',true)
```

Control Code Visibility for Protected Model

Control code visibility by allowing users to view only binary files and headers in the code generated for a protected model.

```
sldemo_mdlref_bus;
model= 'sldemo_mdlref_counter_bus'

Simulink.ModelReference.protect(model,'Mode','CodeGeneration','OutputFormat',...
'CompiledBinaries');
```

A protected model named `sldemo_mdlref_counter_bus.slxp` is created. The protected model file is placed in the current working folder. Users can view only binary files and headers in the code generated for the protected model.

Create Harness Model for Protected Model

Create a harness model for a protected model and generate an HTML report.

```
sldemo_mdlref_bus;
modelPath= 'sldemo_mdlref_bus/CounterA'

[harnessHandle] = Simulink.ModelReference.protect(modelPath,'Path','C:\Work',...
'Harness',true,'Report',true);
```

A protected model named `sldemo_mdlref_counter_bus.slxp` is created, along with an untitled harness model. The protected model file is placed in the `C:\Work` folder. The folder also contains an HTML report. The handle of the harness model is returned in `harnessHandle`.

Determine Variables Required by Protected Model

To simulate a model that references a protected model, you might need to define variables in the base workspace or data dictionaries. For example, the `sldemo_mdlref_counter_bus` model needs the variables that specify the buses at the root input and output ports of the model. When you ship a protected model, you must include definitions of the required variables or the model is unusable.

Tip To automatically package required variable definitions with the protected model in a project, set `Project` to `true`.

Generate the protected model and determine the required variables.

```
sldemo_mdlref_bus;
model= 'sldemo_mdlref_counter_bus'

[~, neededVars] = Simulink.ModelReference.protect(model)
```

The second output, `neededVars`, determines the variables you must send to the recipient. The value of `neededVars` is a cell array that contains the names of the variables required by the protected model. However, the cell array might also contain the names of variables that the model does not need.

Before you share the protected model, edit `neededVars` to delete the names of any variables that the model does not need. Save the required variables in a data dictionary.

Input Arguments

`model` — Model name
character vector | string scalar

Model name, specified as a character vector or string scalar. It contains the name of a model or the path name of a Model block that references the model to be protected.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Mode','CodeGeneration','OutputFormat','Binaries','ObfuscateCode',true` specifies that obfuscated code be generated for the protected model. It also specifies that only binary files and headers in the generated code be visible to users of the protected model.

`Project` — Option to collect dependencies in project
`false` (default) | `true`

Option to collect dependencies in project, specified as the comma-separated pair consisting of `'Project'` and `true` or `false`.

The protected model, its dependencies, and its harness model are saved in a project archive (`.mlproj`). The project archive provides a way to share a project in a single file. You must open the project archive to create the interactive project.

Note Before sharing the project, check whether the project contains the necessary supporting files. If supporting files are missing, simulating or generating code for the related harness model can help identify them. Add the missing dependencies to the project and update the harness model as needed.

Example: `'Project',true`

Data Types: `logical`

`ProjectName` — Custom project name
character vector | string scalar

Custom project name, specified as the comma-separated pair consisting of `'ProjectName'` and a character vector or string scalar.

If you do not specify a custom project name, the default name for the project is the protected model name followed by `_protected`.

Example: `'ProjectName','myname'`

Dependencies

To enable `ProjectName`, set `Project` to `true`.

Data Types: `char` | `string`

`Harness` — Option to create harness model
false (default) | true

Option to create harness model, specified as the comma-separated pair consisting of `'Harness'` and a Boolean value.

When you create a harness model for a protected model that relies on base workspace definitions, Simulink creates a MAT-file that contains the base workspace definitions.

The harness model must have access to supporting files, such as a MAT-file with base workspace definitions or a data dictionary.

Example: `'Harness',true`

Data Types: `logical`

`Mode` — Model protection mode

`'Normal'` (default) | `'Accelerator'` | `'CodeGeneration'` | `'HDLCodeGeneration'` | `'ViewOnly'`

Model protection mode, specified as the comma-separated pair consisting of `'Mode'` and one of the following values:

- `'Normal'`: If the top model is running in `'Normal'` mode, the protected model runs as a child of the top model.
- `'Accelerator'`: The top model can run in `'Normal'`, `'Accelerator'`, or `'Rapid Accelerator'` mode.
- `'CodeGeneration'`: The top model can run in `'Normal'`, `'Accelerator'`, or `'Rapid Accelerator'` mode and support code generation.
- `'HDLCodeGeneration'`: The top model can run in `'Normal'`, `'Accelerator'`, or `'Rapid Accelerator'` mode and support HDL code generation. Requires HDL Coder license.
- `'ViewOnly'`: This value turns off Simulate and Generate code functionality modes and turns on the read-only view mode.

Example: `'Mode','Accelerator'`

`CodeInterface` — Interface through which generated code is accessed by Model block

`'Model reference'` (default) | `'Top model'`

Interface through which generated code is accessed by a Model block, specified as the comma-separated pair consisting of `'CodeInterface'` and one of the following values:

- `'Model reference'`: Code access through the model reference code interface, which allows use of the protected model within a model reference hierarchy. Users of the protected model can generate code from a parent model that contains the protected model. In addition, users can run Model block SIL/PIL simulations with the protected model.
- `'Top model'`: Code access through the standalone interface. Users of the protected model can run Model block SIL/PIL simulations with the protected model.

Example: `'CodeInterface','Top model'`

Dependencies

The system target file (`SystemTargetFile`) must be set to an ERT-based system target file, for example, `ert.tlc`). Requires Embedded Coder license.

`ObfuscateCode` — Option to obfuscate generated code

`true` (default) | `false`

Option to obfuscate generated code, specified as the comma-separated pair consisting of `'ObfuscateCode'` and a Boolean value. Applicable only when code generation during protection is enabled. Obfuscation is not supported for HDL code generation.

Example: `'ObfuscateCode',true`

Data Types: `logical`

`Path` — Folder for protected model
current working folder (default) | character vector | string scalar

Folder for protected model, specified as the comma-separated pair consisting of `'Path'` and a character vector or string scalar.

Example: `'Path','C:\Work'`

Data Types: `char` | `string`

`Report` — Option to generate report
`false` (default) | `true`

Option to generate report, specified as the comma-separated pair consisting of `'Report'` and a Boolean value.

To view the report, right-click the protected-model badge icon and select **Display Report**. Or, call the `Simulink.ProtectedModel.open` function with the `report` option.

The report is generated in HTML format. It includes information on the environment, functionality, and interface for the protected model.

Example: `'Report',true`

Data Types: `logical`

`hdl` — Option to generate HDL code
`false` (default) | `true`

Option to generate HDL code, specified as the comma-separated pair consisting of `'hdl'` and a Boolean value.

This option requires an HDL Coder license. When you enable this option, make sure that you specify the **Mode**. You can set this option to `true` in conjunction with the **Mode** set to `CodeGeneration` to enable both C code and HDL code generation support for the protected model.

If you want to enable only simulation and HDL code generation support, but not C code generation, set **Mode** to `HDLCodeGeneration`. You do not have to set the **hdl** option to `true`.

Example: `'hdl',true`

Data Types: `logical`

`OutputFormat` — Protected code visibility
`'CompiledBinaries'` (default) | `'MinimalCode'` | `'AllReferencedHeaders'`

Protected code visibility, specified as the comma-separated pair consisting of `'OutputFormat'` and one of the following values:

- `'CompiledBinaries'`: Only binary files and headers are visible.
- `'MinimalCode'`: Includes only the minimal header files required to build the code with the chosen build settings. All code in the build folder is visible. Users can inspect the code in the protected model report and recompile it for their purposes.
- `'AllReferencedHeaders'`: Includes header files found on the include path. All code in the build folder is visible. All headers referenced by the code are also visible.

This argument determines what part of the code generated for a protected model is visible to users.

Example: 'OutputFormat','AllReferencedHeaders'

Dependencies

This argument affects the output only when you specify Mode as 'Accelerator' or 'CodeGeneration'. When you specify Mode as 'Normal', only a MEX-file is part of the output package.

Webview — Option to include read-only Web view of protected model

false (default) | true

Option to include read-only Web view of protected model, specified as the comma-separated pair consisting of 'Webview' and a Boolean value.

To open the Web view of a protected model, use one of the following methods:

- Right-click the protected-model badge icon and select **Show Web view**.
- Use the Simulink.ProtectedModel.open function. For example, to display the Web view for protected model sldemo_mdlref_counter, call:

```
Simulink.ProtectedModel.open('sldemo_mdlref_counter', 'webview');
```

- Double-click the .slxp protected model file in the Current Folder browser.
- In the Block Parameter dialog box for the protected model, click **Open Model**.

Example: 'Webview',true

Data Types: logical

Encrypt — Option to encrypt protected model

false (default) | true

Option to encrypt protected model, specified as the comma-separated pair consisting of 'Encrypt' and a Boolean value. Applicable when you have specified a password during protection, or by using the following methods:

- Password for read-only view of model:
  Simulink.ModelReference.ProtectedModel.setPasswordForView
- Password for simulation:
  Simulink.ModelReference.ProtectedModel.setPasswordForSimulation
- Password for code generation:
  Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration
- Password for HDL code generation:
  Simulink.ModelReference.ProtectedModel.setPasswordForHDLCodeGeneration

Example: 'Encrypt',true

Data Types: logical

CustomPostProcessingHook — Option to add postprocessing function for protected model files

function handle

Option to add a postprocessing function for protected model files, specified as the comma-separated pair consisting of `'CustomPostProcessingHook'` and a function handle.

The function accepts a `Simulink.ModelReference.ProtectedModel.HookInfo` object as an input variable. This object provides information on the source code files and other files generated during protected model creation. It also provides information on exported symbols that you must not modify. Prior to packaging the protected model, the postprocessing function is called.

For a protected model with a top model interface, the `Simulink.ModelReference.ProtectedModel.HookInfo` object cannot provide information on exported symbols.

Example: `'CustomPostProcessingHook',@(protectedMdlInf)myHook(protectedMdlInf)`

`Modifiable` — Option to create modifiable protected model
`false` (default) | `true`

Option to create modifiable protected model, specified as the comma-separated pair consisting of `'Modifiable'` and a Boolean value. To use this option:

- Add a password for modification by using the `Simulink.ModelReference.ProtectedModel.setPasswordForModify` function. If a password has not been added at the time that you create the modifiable protected model, you are prompted to create one.
- Modify the options of your protected model by first providing the modification password using the `Simulink.ModelReference.ProtectedModel.setPasswordForModify` function. Then, use the `Simulink.ModelReference.modifyProtectedModel` function to make your option changes.

Example: `'Modifiable',true`

Data Types: `logical`

`Callbacks` — Option to specify callbacks for protected model
cell array

Option to specify callbacks for protected model, specified as the comma-separated pair consisting of `'Callbacks'` and a cell array of `Simulink.ProtectedModel.Callback` objects.

Example: `'Callbacks',{pmcallback_sim, pmcallback_cg}`

Data Types: `cell`

`Sign` — Option to sign protected model with digital certificate
character vector | string scalar

Option to sign protected model with digital certificate, specified as the comma-separated pair consisting of `'Sign'` and a character vector or string scalar that specifies the digital certificate. If the certificate file is password-protected, use the `Simulink.ModelReference.ProtectedModel.setPasswordForCertificate` function to provide the password before you use the certificate.

Example: `'Sign','my_certificate.pfx'`

Data Types: `char` | `string`

Output Arguments

`harnessHandle` — Handle of harness model
double

Handle of harness model, returned as a double or `0`, depending on the value of `Harness`.

If `Harness` is `true`, the value is the handle of the harness model. Otherwise, the value is `0`.

`neededVars` — Names of base workspace variables
cell array

Names of base workspace variables that the protected model uses, returned as a cell array.

The cell array can also include variables that the protected model does not use.

Alternatives

"Protect Models to Conceal Contents"

See Also

`Simulink.ModelReference.ProtectedModel.clearPasswords` |
`Simulink.ModelReference.ProtectedModel.clearPasswordsForModel` |
`Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration` |
`Simulink.ModelReference.ProtectedModel.setPasswordForHDLCodeGeneration` |
`Simulink.ModelReference.ProtectedModel.setPasswordForModify` |
`Simulink.ModelReference.ProtectedModel.setPasswordForSimulation` |
`Simulink.ModelReference.ProtectedModel.setPasswordForView` |
`Simulink.ModelReference.modifyProtectedModel`

Topics
"Protect Models to Conceal Contents"
"Explore Protected Model Capabilities"
"Test Protected Models"
"Package and Share Protected Models"
"Specify Custom Obfuscators for Protected Models"
"Configure and Run SIL Simulation"
"Define Callbacks for Protected Models"
"Reference Protected Models from Third Parties"
"Code Interfaces for SIL and PIL"

Introduced in R2012b

Simulink.ModelReference.ProtectedModel.clearPasswords

Clear cached passwords for protected models

Syntax

`Simulink.ModelReference.ProtectedModel.clearPasswords()`

Description

`Simulink.ModelReference.ProtectedModel.clearPasswords()` clears protected model passwords that have been cached during the current MATLAB session. If this function is not called, cached passwords are cleared at the end of a MATLAB session.

Examples

Clear cached passwords for protected models

After using protected models, clear passwords cached for the models during the MATLAB session.

```
Simulink.ModelReference.ProtectedModel.clearPasswords()
```

See Also

`Simulink.ModelReference.ProtectedModel.clearPasswordsForModel`

Topics
"Protect Models to Conceal Contents"

Introduced in R2014b

Simulink.ModelReference.ProtectedModel.-clearPasswordsForModel

Clear cached passwords for a protected model

Syntax

`Simulink.ModelReference.ProtectedModel.clearPasswordsForModel(model)`

Description

`Simulink.ModelReference.ProtectedModel.clearPasswordsForModel(model)` clears protected model passwords for `model` that have been cached during the current MATLAB session. If this function is not called, cached passwords are cleared at the end of a MATLAB session.

Examples

Clear cached passwords for a protected model

After using a protected model, clear passwords cached for the model during the MATLAB session.

```
Simulink.ModelReference.ProtectedModel.clearPasswordsForModel(model)
```

Input Arguments

`model` — Protected model name
string or character vector

Model name specified as a string or character vector

Example: '`rtwdemo_counter`'

Data Types: `char`

See Also

`Simulink.ModelReference.ProtectedModel.clearPasswords`

Topics
"Protect Models to Conceal Contents"

Introduced in R2014b

Simulink.ModelReference.ProtectedModel.HookInfo

Files and exported symbols generated by creation of protected model

Description

Information about files and symbols generated when creating a protected model. You can use this information for postprocessing of the generated files prior to packaging. To access the properties of this class, create a postprocessing function that accepts a `Simulink.ModelReference.ProtectedModel.HookInfo` object as input. When you protect your model, use the '`CustomPostProcessingHook`' option of the `Simulink.ModelReference.protect` function to specify the postprocessing function. Prior to packaging the protected model, the postprocessing function is called with the `Simulink.ModelReference.ProtectedModel.HookInfo` from the protected model as input.

Properties

`ExportedSymbols` — Exported Symbols
cell array of character vectors

A list of exported symbols generated by a protected model that you must not modify, returned as a cell array of character vectors.

For a protected model with a top model interface, the `HookInfo` object cannot provide information on exported symbols.

`NonSourceFiles` — Nonsource code files
cell array of character vectors

A list of nonsource files generated by protected model creation, returned as a cell array of character vectors. Nonsource files include MAT, RSP, and PRJ files.

`SourceFiles` — Source code files
cell array of character vectors

A list of source code files generated by protected model creation, returned as a cell array of character vectors. Source files include C, H, CPP, and HPP files.

Examples

Use Protected Model Information in Postprocessing Function

1 On the MATLAB path, create a postprocessing function `pm_postprocessing.m` that contains this code:

```
function pm_postprocessing(hookInfoObject)
```

```
    s1 = 'Exported Symbols: ';
    symbols = hookInfoObject.ExportedSymbols;
    s2 = 'Source Files:';
    srcfiles = hookInfoObject.SourceFiles;
    s3 = 'Non-Source Files:';
    nonsrcfiles = hookInfoObject.NonSourceFiles;
    disp([s1 symbols])
    disp([s2 srcfiles])
    disp([s3 nonsrcfiles])
```

This function displays a list of the exported symbols, source code files, and nonsource files generated by the model protection process.

2 Protect the model `sldemo_mdlref_counter` and specify the postprocessing function that you created. Before packaging the generated files, the model protection process calls the postprocessing function and inputs the `Simulink.ModelReference.ProtectedModel.HookInfo` object that was generated for the protected model.

```
Simulink.ModelReference.protect('sldemo_mdlref_counter',...
'Mode', 'CodeGeneration',...
'CustomPostProcessingHook',...
@(protectedMdlInf)pm_postprocessing(protectedMdlInf))
```

See Also

`Simulink.ModelReference.protect`

Topics

"Specify Custom Obfuscators for Protected Models"

Introduced in R2014a

Simulink.ModelReference.ProtectedModel.setPasswordForCertificate

Provide password for digital certificate

Syntax

```
Simulink.ModelReference.ProtectedModel.setPasswordForCertificate(
certificateFile,password)
```

Description

`Simulink.ModelReference.ProtectedModel.setPasswordForCertificate(certificateFile,password)` provides the required password to access the certificate file to digitally sign a protected model.

Examples

Sign a Protected Model by Using Password Protected Certificate

Protect a model, and then digitally sign it by using a password protected certificate.

Open and protect the model that you want to sign. For this example, protect the model `sldemo_mdlref_counter`.

```
sldemo_mdlref_counter
Simulink.ModelReference.protect('sldemo_mdlref_counter');
```

Locate the certificate file that you want to use to sign the protected model. Enter the password for the certificate.

```
Simulink.ModelReference.ProtectedModel.setPasswordForCertificate('certificate_file.pfx','password');
```

Sign the protected model by using the certificate file.

```
Simulink.ProtectedModel.sign('sldemo_mdlref_counter.slxp','certificate_file.pfx');
```

Input Arguments

`certificateFile` — Certificate file to use for signing
character vector | string scalar

Certificate file to use for signing the protected model, specified as a character vector or string scalar. The certificate must be a PKCS #12 file with the extension `.pfx` or `.p12`.

Example: `'my_cert.pfx'`

Example: `'InstitutionCertificate.p12'`

`password` — Password for certificate file
string or character vector

Password, specified as a string or character vector. If the certificate is encrypted for code generation, the password is required.

See Also

`Simulink.ProtectedModel.sign`

Topics

"Sign a Protected Model"

Introduced in R2020a

Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration

Add or provide encryption password for code generation from protected model

Syntax

```
Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration(model,
password)
```

Description

`Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration(model, password)` adds an encryption password for code generation if you create a protected model. If you use a protected model, the function provides the required password to generate code from the model.

Examples

Create a Protected Model with Encryption

Create a protected model with encryption for code generation.

```
Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration(...
'sldemo_mdlref_counter','password');
Simulink.ModelReference.protect('sldemo_mdlref_counter',...
'Mode','Code Generation','Encrypt',true,'Report',true);
```

A protected model named `sldemo_mdlref_counter.slxp` is created that requires an encryption password for code generation.

Generate Code from an Encrypted Protected Model

Use a protected model with encryption for code generation.

Provide the encryption password required for code generation from the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration(...
'sldemo_mdlref_counter','password');
```

After you have provided the encryption password, you can generate code from the protected model.

Input Arguments

`model` — Model name
string or character vector

Model name, specified as a string or character vector. It contains the name of a model or the path name of a Model block that references the protected model.

`password` — Password for protected model code generation
string or character vector

Password, specified as a string or character vector. If the protected model is encrypted for code generation, the password is required.

See Also

`Simulink.ModelReference.ProtectedModel.setPasswordForHDLCodeGeneration` | `Simulink.ModelReference.ProtectedModel.setPasswordForSimulation` | `Simulink.ModelReference.ProtectedModel.setPasswordForView` | `Simulink.ModelReference.protect`

Introduced in R2014b

Simulink.ModelReference.ProtectedModel.setPasswordForModify

Add or provide password for modifying protected model

Syntax

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(model,password)
```

Description

`Simulink.ModelReference.ProtectedModel.setPasswordForModify(model,password)` adds a password for a modifiable protected model. After the password has been created, the function provides the password for modifying the protected model.

Examples

Add Functionality to Protected Model

Create a modifiable protected model with support for code generation, then modify it to add Web view support.

Add the password for when a protected model is modified. If you skip this step, you are prompted to set a password when a modifiable protected model is created.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Create a modifiable protected model with support for code generation and Web view.

```
Simulink.ModelReference.protect('sldemo_mdlref_counter','Mode',...
'CodeGeneration', 'Modifiable',true, 'Report',true);
```

Provide the password to modify the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter', 'password');
```

Add support for Web view to the protected model that you created.

```
Simulink.ModelReference.modifyProtectedModel(...
'sldemo_mdlref_counter','Mode','CodeGeneration','Webview',true,...
'Report',true);
```

Input Arguments

`model` — Model name
string or character vector

Model name, specified as a string or character vector. It contains the name of a model or the path name of a Model block that references the protected model to be modified.

password — Password to modify protected model
string or character vector

Password, specified as a string or character vector. The password is required for modification of the protected model.

See Also

Simulink.ModelReference.modifyProtectedModel | Simulink.ModelReference.protect

Introduced in R2014b

Simulink.ModelReference.ProtectedModel.setPasswordForSimulation

Add or provide encryption password for simulation of protected model

Syntax

`Simulink.ModelReference.ProtectedModel.setPasswordForSimulation(model, password)`

Description

`Simulink.ModelReference.ProtectedModel.setPasswordForSimulation(model, password)` adds an encryption password for simulation if you create a protected model. If you use a protected model, the function provides the required password to simulate the model.

Examples

Create a Protected Model with Encryption

Create a protected model with encryption for simulation.

```
Simulink.ModelReference.ProtectedModel.setPasswordForSimulation(...
'sldemo_mdlref_counter','password');
Simulink.ModelReference.protect('sldemo_mdlref_counter',...
'Encrypt',true,'Report',true);
```

A protected model named `sldemo_mdlref_counter.slxp` is created that requires an encryption password for simulation.

Simulate an Encrypted Protected Model

Use a protected model with encryption for simulation.

Provide the encryption password required for simulation of the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForSimulation(...
'sldemo_mdlref_counter','password');
```

After you have provided the encryption password, you can simulate the protected model.

Input Arguments

`model` — Model name
string or character vector

Model name, specified as a string or character vector. It contains the name of a model or the path name of a Model block that references the protected model.

`password` — Password for protected model simulation
string or character vector

Password, specified as a string or character vector. If the protected model is encrypted for simulation, the password is required.

See Also

`Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration` | `Simulink.ModelReference.ProtectedModel.setPasswordForHDLCodeGeneration` | `Simulink.ModelReference.ProtectedModel.setPasswordForView` | `Simulink.ModelReference.protect`

Introduced in R2014b

Simulink.ModelReference.ProtectedModel.setPasswordForView

Add or provide encryption password for read-only view of protected model

Syntax

`Simulink.ModelReference.ProtectedModel.setPasswordForView(model,password)`

Description

`Simulink.ModelReference.ProtectedModel.setPasswordForView(model,password)` adds an encryption password for read-only view if you create a protected model. If you use a protected model, the function provides the required password for a read-only view of the model.

Examples

Create a Protected Model with Encryption

Create a protected model with encryption for read-only view.

```
Simulink.ModelReference.ProtectedModel.setPasswordForView(...
'sldemo_mdlref_counter','password');
Simulink.ModelReference.protect('sldemo_mdlref_counter',...
'Webview',true,'Encrypt',true,'Report',true);
```

A protected model named `sldemo_mdlref_counter.slxp` is created that requires an encryption password for read-only view.

View an Encrypted Protected Model

Use a protected model with encryption for read-only view.

Provide the encryption password required for the read-only view of the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForView(...
'sldemo_mdlref_counter','password');
```

After you have provided the encryption password, you have access to the read-only view of the protected model.

Input Arguments

`model` — Model name
string or character vector

Model name, specified as a string or character vector. It contains the name of a model or the path name of a Model block that references the protected model.

`password` — Password for read-only view of protected model
string or character vector

Password, specified as a string or character vector. If the protected model is encrypted for read-only view, the password is required.

See Also

`Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration` | `Simulink.ModelReference.ProtectedModel.setPasswordForHDLCodeGeneration` | `Simulink.ModelReference.ProtectedModel.setPasswordForSimulation` | `Simulink.ModelReference.protect`

Introduced in R2014b

Simulink.ProtectedModel.addTarget

Add code generation support for current target to protected model

Syntax

`Simulink.ProtectedModel.addTarget(model)`

Description

`Simulink.ProtectedModel.addTarget(model)` adds code generation support for the current `model` target to a protected model of the same name. Each target that the protected model supports is identified by the root of the **Code Generation > System Target file** (`SystemTargetFile`) parameter. For example, if the **System Target file** is `ert.tlc`, the target identifier is `ert`.

To add the current target:

- The model and the protected model of the same name must be on the MATLAB path.
- The protected model must have the `Modifiable` option enabled and have a password for modification.
- The target must be unique in the protected model.

If you add a target to a protected model that did not previously support code generation, the software switches the protected model `Mode` to `CodeGeneration` and `ObfuscateCode` to `true`.

Examples

Add a Target to a Protected Model

Add the currently configured model target to the protected model.

Load the model and save a local copy.

```
sldemo_mdlref_counter
save_system('sldemo_mdlref_counter','mdlref_counter.slx');
```

Add a required password for modifying a protected model. If you do not add a password, you are prompted to set a password when you create a modifiable, protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'mdlref_counter','password');
```

Create a modifiable, protected model with support for code generation.

```
Simulink.ModelReference.protect('mdlref_counter','Mode',...
'CodeGeneration', 'Modifiable',true, 'Report',true);
```

Get a list of targets that the protected model supports.

```
 st = Simulink.ProtectedModel.getSupportedTargets('mdlref_counter')
```

Configure the unprotected model to support a new target.

```
set_param('mdlref_counter', 'SystemTargetFile', 'ert.tlc');
save_system('mdlref_counter');
```

Add support to the protected model for the new target. You are prompted for the modification password.

```
Simulink.ProtectedModel.addTarget('mdlref_counter');
```

Verify that support for the new target has been added to the protected model.

```
st = Simulink.ProtectedModel.getSupportedTargets('mdlref_counter')
```

Input Arguments

`model` — Model name
string or character vector

Model name, specified as a string or character vector. It contains the name of a model or the path name of a Model block that references the protected model.

See Also

`Simulink.ModelReference.protect` | `Simulink.ProtectedModel.getConfigSet` | `Simulink.ProtectedModel.getCurrentTarget` | `Simulink.ProtectedModel.getSupportedTargets` | `Simulink.ProtectedModel.removeTarget` | `Simulink.ProtectedModel.setCurrentTarget`

Topics
"Create Protected Models with Multiple Targets"

Introduced in R2015a

Simulink.ProtectedModel.Callback

Callback code that executes in response to protected model events

Description

For a specific protected model functionality, the `Simulink.ProtectedModel.Callback` object specifies code to execute in response to an event. The callback code can be a character vector of MATLAB commands or a MATLAB script.

When you create a protected model, to specify callbacks, call the Simulink.ModelReference.protect function with the `'Callbacks'` option. The value of this option is a cell array of `Simulink.ProtectedModel.Callback` objects.

Creation

Syntax

```
Simulink.ProtectedModel.Callback(Event,AppliesTo,CallbackText)
Simulink.ProtectedModel.Callback(Event,AppliesTo,callbackFile)
```

Description

`Simulink.ProtectedModel.Callback(Event,AppliesTo,CallbackText)` creates a callback object for a specific protected model functionality and event. The `CallbackText` specifies MATLAB commands to execute for the callback.

`Simulink.ProtectedModel.Callback(Event,AppliesTo,callbackFile)` creates a callback object for a specific protected model functionality and event. The `CallbackFileName` specifies a MATLAB script to execute for the callback. The script must be on the MATLAB path.

Properties

`AppliesTo` — Protected model functionality
`'AUTO'` (default) | `'CODEGEN'` | `'SIM'` | `'VIEW'`

Protected model functionality that the event applies to, specified as one of these values:

- `'CODEGEN'`: Code generation.
- `'SIM'`: Simulation.
- `'VIEW'`: Read-only web view.
- `'AUTO'`: If the event is `'PreAccess'`, the callback executes for each functionality. If the event is `'Build'`, the callback executes for only `'CODEGEN'` functionality.

Example: `'SIM'`

`CallbackFileName` — Callback script to execute
character vector | string scalar

MATLAB script to execute in response to an event, specified as a character vector or string scalar. The script must be on the MATLAB path.

Example: `'pmCallback.m'`

`CallbackText` — Callback code to execute
character vector | string scalar

MATLAB commands to execute in response to an event, specified as a character vector or string scalar.

Example: `'A = [15 150];disp(A)'`

`Event` — Event that triggers callback
`'PreAccess'` | `'Build'`

Event that triggers the callback, specified as one of these options:

- `'PreAccess'`: Callback code is executed before simulation, build, or read-only viewing.
- `'Build'`: Callback code is executed before build. Valid for only `'CODEGEN'` functionality.

Example: `'PreAccess'`

`OverrideBuild` — Option to override protected model build
`false` (default) | `true`

Option to override the protected model build process, specified as a Boolean value. The override option applies only to a callback object that you define for a `'Build'` event for the `'CODEGEN'` functionality. You set this option by using the `setOverrideBuild` method.

Object Functions

setOverrideBuild   Override protected model build

Examples

Create Protected Model by Using a Callback

1. Create the callback object.

```
pmCallback = Simulink.ProtectedModel.Callback('PreAccess',...
'SIM','disp(''Hello world!'')')
```

2. Protect the model `sldemo_mdlref_counter` and specify the callback object.

```
Simulink.ModelReference.protect('sldemo_mdlref_counter',...
'Callbacks',{pmCallback})
```

3. Simulate the model `sldemo_mdlref_basic`, which references the protected model that you created.

```
sim('sldemo_mdlref_basic')
```

For each instance of the protected model reference in the top model, the output is displayed.

```
Hello world!
Hello world!
Hello world!
```

Create Protected Model That Uses a Callback Script

1. On the MATLAB path, create a callback script `pm_callback.m` that contains this code:

```
disp('Hello world!')
```

2. Create a callback object that uses the script for the callback code. Protect the model `sldemo_mdlref_counter` and specify the callback object.

```
pmCallback = Simulink.ProtectedModel.Callback('Build',...
'CODEGEN','pm_callback.m')
Simulink.ModelReference.protect('sldemo_mdlref_counter',...
'Mode', 'CodeGeneration','Callbacks',{pmCallback})
```

   The callback script executes during the code generation phase of the model protection process.

3. Generate code for the model `sldemo_mdlref_basic`, which references the protected model that you created.

```
slbuild('sldemo_mdlref_basic')
```

   During the code generation phase for the referenced protected model, code in `pm_callback.m` executes.

See Also

`Simulink.ModelReference.protect` | `Simulink.ProtectedModel.getCallbackInfo`

Topics

"Define Callbacks for Protected Models"
"Protect Models to Conceal Contents"
"Code Generation Requirements and Limitations"

Introduced in R2016a

setOverrideBuild

Package: `Simulink.ProtectedModel`

Override protected model build

Syntax

```
setOverrideBuild(callback, override)
```

Description

`setOverrideBuild(callback, override)` specifies whether a `Simulink.ProtectedModel.Callback` object can override the build process. This method is valid only for callbacks that execute in response to a `'Build'` event for `'CODEGEN'` functionality.

Examples

Create Code Generation Callback to Override Build Process

1. Create a callback object that uses a character vector of MATLAB commands for the callback code. Define the callback for a `'Build'` event for the `'CODEGEN'` functionality.

   ```
   pmCallback = Simulink.ProtectedModel.Callback('Build',...
   'CODEGEN','disp(''Hello world!'')')
   ```
2. Specify that the callback override the build process.

   ```
   setOverrideBuild(pmCallback, true);
   ```
3. Protect the model `sldemo_mdlref_counter` and specify the callback that you created.

   ```
   Simulink.ModelReference.protect('sldemo_mdlref_counter',...
   'Mode', 'CodeGeneration','Callbacks',{pmCallback})
   ```
4. Build the model `sldemo_mdlref_basic`, which references the protected model `sldemo_mdlref_counter`. When the top model begins to build the protected model, the callback that you created overrides the build process.

   ```
   slbuild('sldemo_mdlref_basic')
   ```

Input Arguments

`callback` — Protected model callback
`Simulink.ProtectedModel.Callback` object

Protected model callback that you want to override the protected model build process, specified as a `Simulink.ProtectedModel.Callback`. The callback object must be defined for a `'Build'` event for `'CODEGEN'` functionality.

`override` — Option to override protected model build process
false (default) | true

Option to override the protected model build process, specified as a Boolean value. This option applies to only a callback object defined for a `'Build'` event for the `'CODEGEN'` functionality.

Example: `pmcallback.setOverrideBuild(true)`

See Also

`Simulink.ModelReference.protect` | `Simulink.ProtectedModel.Callback`

Topics

"Define Callbacks for Protected Models"
"Protect Models to Conceal Contents"
"Code Generation Requirements and Limitations"

Introduced in R2016a

Simulink.ProtectedModel.CallbackInfo

Protected model information for use in callbacks

Description

A `Simulink.ProtectedModel.CallbackInfo` object contains information about a protected model that you can use in the code executed for a callback.

Creation

Syntax

`Simulink.ProtectedModel.getCallbackInfo(ModelName,Event,Functionality)`

Description

`Simulink.ProtectedModel.getCallbackInfo(ModelName,Event,Functionality)` creates a `Simulink.ProtectedModel.CallbackInfo` object for the callback that applies to the protected model for the event and functionality that you specify.

Properties

`CodeInterface` — Code interface generated by protected model
`'Top model'` | `'Model reference'`

Code interface that the protected model generates, specified as `'Top model'` or `'Model reference'`.

`Event` — Event that triggered callback
`'PreAccess'` | `'Build'`

Event that triggered the callback, specified as one of these values:

- `'PreAccess'`: Callback code executed before simulation, build, or read-only viewing.
- `'Build'`: Callback code executed before build. This option is valid only for the `'CODEGEN'` functionality.

`Functionality` — Protected model functionality
`'AUTO'` (default) | `'CODEGEN'` | `'SIM'` | `'VIEW'`

Protected model functionality that the event applies to, specified as one of these values:

- `'CODEGEN'`: Code generation.
- `'SIM'`: Simulation.
- `'VIEW'`: Read-only web view.
- `'AUTO'`: If the event is `'PreAccess'`, the callback executes for each functionality. If the event is `'Build'`, the callback executes for only `'CODEGEN'` functionality.

`ModelName` — Protected model name
character vector | string scalar

Protected model name, specified as a character vector or string scalar.

Example: `'myProtectedModel.slxp'`

`SubModels` — Models and submodels in the protected model container
cell array of character vectors

Names of all models and submodels in the protected model container, specified as a cell array of character vectors.

`Target` — Current target
character vector

Current target identifier for the protected model, specified as a character vector. This property is available for only code generation callbacks.

Example: `'ert'`

Object Functions

getBuildInfoForModel   Build information object for specified model

Examples

Use Protected Model Information in Simulation Callback

1. On the MATLAB path, create a callback script `pm_callback.m` that contains this code:

   ```
   s1 = 'Simulating protected model: ';
   cbinfobj = Simulink.ProtectedModel.getCallbackInfo(...
   'sldemo_mdlref_counter','PreAccess','SIM');
   disp([s1 cbinfobj.ModelName])
   ```

   The script uses the `Simulink.ProtectedModel.CallbackInfo` object to display the name of the protected model.
2. Create a callback object that uses the script for the callback code. Protect the model `sldemo_mdlref_counter` and specify the callback object.

   ```
   pmCallback = Simulink.ProtectedModel.Callback('PreAccess'...
   ,'SIM', 'pm_callback.m')
   Simulink.ModelReference.protect('sldemo_mdlref_counter',...
   'Callbacks',{pmCallback})
   ```
3. Simulate the protected model. When each instance of the protected model reference in the top model is simulated, the output from the callback is listed.

   ```
   sim('sldemo_mdlref_basic')
   ```

   ```
   Simulating protected model: sldemo_mdlref_counter
   Simulating protected model: sldemo_mdlref_counter
   Simulating protected model: sldemo_mdlref_counter
   ```

See Also

`Simulink.ModelReference.protect` | `Simulink.ProtectedModel.getCallbackInfo`

Topics
"Define Callbacks for Protected Models"
"Protect Models to Conceal Contents"
"Code Generation Requirements and Limitations"

Introduced in R2016a

Simulink.ProtectedModel.getCallbackInfo

Get `Simulink.ProtectedModel.CallbackInfo` object for use by callbacks

Syntax

```
cbinfobj = Simulink.ProtectedModel.getCallbackInfo(modelName,event,
functionality)
```

Description

`cbinfobj = Simulink.ProtectedModel.getCallbackInfo(modelName,event, functionality)` returns a `Simulink.ProtectedModel.CallbackInfo` object that provides information for protected model callbacks. The object contains information about the protected model, including:

- Model name.
- List of models and submodels in the protected model container.
- Callback event.
- Callback functionality.
- Code interface.
- Current target. This information is available only for code generation callbacks.

Examples

Use Protected Model Information in Code Generation Callback

On the MATLAB path, create a callback script, `pm_callback.m`, containing:

```
s1 = 'Code interface is: ';
cbinfobj = Simulink.ProtectedModel.getCallbackInfo(...
'sldemo_mdlref_counter','Build','CODEGEN');
disp([s1 cbinfobj.CodeInterface]);
```

When you create a protected model with a simulation callback, use the script.

```
pmCallback = Simulink.ProtectedModel.Callback('Build',...
'CODEGEN', 'pm_callback.m')
Simulink.ModelReference.protect('sldemo_mdlref_counter',...
'Mode', 'CodeGeneration','Callbacks',{pmCallback})
```

Build the protected model. Before the start of the protected model build process, the code interface is displayed.

```
slbuild('sldemo_mdlref_basic')
```

Input Arguments

`modelName` — Protected model name
string or character vector

Protected model name, specified as a string or character vector.

event — Event that triggered callback
'PreAccess' | 'Build'

Callback trigger event. Value is one of the following:

- 'PreAccess': Callback code executed before simulation, build, or read-only viewing.
- 'Build': Callback code executed before build. Valid only for 'CODEGEN' functionality.

functionality — Protected model functionality
'CODEGEN' | 'SIM' | 'VIEW' | 'AUTO'

Protected model functionality that the event applies to. Value is one of the following:

- 'CODEGEN': Code generation.
- 'SIM': Simulation.
- 'VIEW': Read-only Web view.
- 'AUTO': If the event is 'PreAccess', the callback executes for each functionality. If the event is 'Build', the callback executes only for 'CODEGEN' functionality.

If the value of functionality is blank, the default behavior is 'AUTO'.

Output Arguments

cbinfobj — Callback information object
Simulink.ProtectedModel.CallbackInfo

Callback information, specified as a Simulink.ProtectedModel.CallbackInfo object.

See Also

Simulink.ModelReference.protect | Simulink.ProtectedModel.CallbackInfo

Topics
“Define Callbacks for Protected Models”
“Protect Models to Conceal Contents”
“Code Generation Requirements and Limitations”

Introduced in R2016a

getBuildInfoForModel

Package: `Simulink.ProtectedModel`

Build information object for specified model

Syntax

`bldobj = getBuildInfoForModel(callbackInfo, model)`

Description

`bldobj = getBuildInfoForModel(callbackInfo, model)` returns the `RTW.BuildInfo` object that specifies the build toolchain and arguments for the model. The `model` name must be in the list of model names in the `SubModels` property of the `Simulink.ProtectedModel.CallbackInfo` object. You can call this method for only code generation callbacks in response to a `'Build'` event.

Examples

Get Build Information from a Code Generation Callback

1. On the MATLAB path, create a callback script, `pm_callback.m`, that contains this code:

```
cbinfobj = Simulink.ProtectedModel.getCallbackInfo(...
'sldemo_mdlref_counter','Build','CODEGEN');
bldinfo = cbinfobj.getBuildInfoForModel(cbinfobj.ModelName);
buildargs = getBuildArgs(bldinfo)
```

2. Create a callback object that uses the script.

```
pmCallback = Simulink.ProtectedModel.Callback('Build',...
'CODEGEN', 'pm_callback.m')
Simulink.ModelReference.protect('sldemo_mdlref_counter',...
'Mode', 'CodeGeneration','Callbacks',{pmCallback})
```

3. Build the protected model. Before the start of the protected model build, the build arguments are displayed.

```
slbuild('sldemo_mdlref_basic')
```

Input Arguments

`callbackInfo` — Callback information object
`Simulink.ProtectedModel.CallbackInfo` object

Callback information object, specified as a `Simulink.ProtectedModel.CallbackInfo` object. The callback object must be defined for a `'Build'` event for the `'CODEGEN'` functionality.

`model` — Model name
string or character vector

Model name, specified as a string or character vector. The `model` name must be in the list of model names in the `SubModels` property of the `Simulink.ProtectedModel.CallbackInfo` object.

Output Arguments

`bldobj` — Build toolchain and arguments
`RTW.BuildInfo` object

Build toolchain and arguments, returned as a `RTW.BuildInfo` object.

See Also
`Simulink.ModelReference.protect` | `Simulink.ProtectedModel.CallbackInfo`

Topics
"Define Callbacks for Protected Models"
"Protect Models to Conceal Contents"
"Code Generation Requirements and Limitations"

Introduced in R2016a

Simulink.ProtectedModel.getConfigSet

Get configuration set for current protected model target or for specified target

Syntax

```
configSet = Simulink.ProtectedModel.getConfigSet(protectedModel)
configSet = Simulink.ProtectedModel.getConfigSet(protectedModel,targetID)
```

Description

`configSet = Simulink.ProtectedModel.getConfigSet(protectedModel)` returns the configuration set object for the current, protected model target.

`configSet = Simulink.ProtectedModel.getConfigSet(protectedModel,targetID)` returns the configuration set object for a specified target that the protected model supports.

Examples

Get Configuration Set for Current Target

Get the configuration set for the currently configured, protected model target.

Load the model and save a local copy.

```
sldemo_mdlref_counter
save_system('sldemo_mdlref_counter','mdlref_counter.slx');
```

Add a required password for modifying a protected model. If you do not add a password, you are prompted to set a password when you create a modifiable, protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'mdlref_counter','password');
```

Create a modifiable, protected model with support for code generation.

```
Simulink.ModelReference.protect('mdlref_counter','Mode',...
'CodeGeneration', 'Modifiable',true, 'Report',true);
```

Get the configuration set for the currently configured target.

```
cs = Simulink.ProtectedModel.getConfigSet('mdlref_counter')
```

Get Configuration Set for Specified Target

Get the configuration set for a specified target that the protected model supports.

Load the model and save a local copy.

```
sldemo_mdlref_counter
save_system('sldemo_mdlref_counter','mdlref_counter.slx');
```

Add a required password for modifying a protected model. If you do not add a password, you are prompted to set a password when you create a modifiable, protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'mdlref_counter','password');
```

Create a modifiable, protected model with support for code generation.

```
Simulink.ModelReference.protect('mdlref_counter','Mode',...
'CodeGeneration', 'Modifiable',true, 'Report',true);
```

Configure the unprotected model to support a new target.

```
set_param('mdlref_counter', 'SystemTargetFile', 'ert.tlc');
save_system('mdlref_counter');
```

Add support to the protected model for the new target. You are prompted for the modification password.

```
Simulink.ProtectedModel.addTarget('mdlref_counter');
```

Verify that support for the new target has been added to the protected model.

```
st = Simulink.ProtectedModel.getSupportedTargets('mdlref_counter')
```

Get the configuration set for the added target.

```
cs = Simulink.ProtectedModel.getConfigSet('mdlref_counter','ert')
```

Input Arguments

`protectedModel` — Model name
string or character vector

Protected model name, specified as a string or character vector.

`targetID` — Target identifier
string or character vector

Identifier for selected target, specified as a string or character vector. The target identifier is the root of the **Code Generation > System Target file** (`SystemTargetFile`) parameter. For example, if the **System Target file** is `ert.tlc`, the target identifier is `ert`.

Output Arguments

`configSet` — Configuration object
`Simulink.ConfigSet`

Configuration set, specified as a `Simulink.ConfigSet` object

See Also

`Simulink.ModelReference.protect` | `Simulink.ProtectedModel.addTarget` | `Simulink.ProtectedModel.getCurrentTarget` | `Simulink.ProtectedModel.getSupportedTargets` | `Simulink.ProtectedModel.removeTarget` | `Simulink.ProtectedModel.setCurrentTarget`

Topics
"Create Protected Models with Multiple Targets"
"Reference Protected Models from Third Parties"

Introduced in R2015a

Simulink.ProtectedModel.getCurrentTarget

Get current protected model target

Syntax

```
currentTarget = Simulink.ProtectedModel.getCurrentTarget(protectedModel)
```

Description

`currentTarget = Simulink.ProtectedModel.getCurrentTarget(protectedModel)` returns the target identifier for the target that is currently configured for the protected model. At the start of a MATLAB session, the default current target is the last target added to the protected model. Otherwise, the current target is the last target that you used. You can change the current target using the `Simulink.ProtectedModel.setCurrentTarget` function.

When building the model, the software changes the target to match the parent if the currently selected target does not match the target of the parent model.

Examples

Get Currently Configured Target for Protected Model

Add a target to a protected model, and then get the currently configured target for the protected model.

Load the model and save a local copy.

```
sldemo_mdlref_counter
save_system('sldemo_mdlref_counter','mdlref_counter.slx');
```

Add a required password for modifying a protected model. If you do not add a password, you are prompted to set a password when you create a modifiable, protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'mdlref_counter','password');
```

Create a modifiable, protected model with support for code generation.

```
Simulink.ModelReference.protect('mdlref_counter','Mode',...
'CodeGeneration', 'Modifiable',true, 'Report',true);
```

Configure the unprotected model to support a new target.

```
 set_param('mdlref_counter', 'SystemTargetFile', 'ert.tlc');
 save_system('mdlref_counter');
```

Add support to the protected model for the new target. You are prompted for the modification password.

```
 Simulink.ProtectedModel.addTarget('mdlref_counter');
```

Verify that support for the new target has been added to the protected model.

```
st = Simulink.ProtectedModel.getSupportedTargets('mdlref_counter')
```

Get the currently configured target for the protected model.

```
ct = Simulink.ProtectedModel.getCurrentTarget('mdlref_counter')
```

Input Arguments

`protectedModel` — Model name
string or character vector

Protected model name, specified as a string or character vector.

Output Arguments

`currentTarget` — Current target
character vector

Current target for protected model, specified as a character vector.

See Also

`Simulink.ModelReference.protect` | `Simulink.ProtectedModel.addTarget` | `Simulink.ProtectedModel.getConfigSet` | `Simulink.ProtectedModel.getSupportedTargets` | `Simulink.ProtectedModel.removeTarget` | `Simulink.ProtectedModel.setCurrentTarget`

Topics

"Create Protected Models with Multiple Targets"
"Reference Protected Models from Third Parties"

Introduced in R2015a

Simulink.ProtectedModel.getSupportedTargets

Get list of targets that protected model supports

Syntax

```
supportedTargets = Simulink.ProtectedModel.getSupportedTargets(
protectedModel)
```

Description

`supportedTargets = Simulink.ProtectedModel.getSupportedTargets( protectedModel)` returns a list of target identifiers for the code generation targets supported by the specified protected model. The target identifier `sim` represents simulation support.

Examples

Get List of Supported Targets for a Protected Model

Add a target to a protected model, and then get a list of supported targets to verify the addition of the new target.

Load the model and save a local copy.

```
sldemo_mdlref_counter
save_system('sldemo_mdlref_counter','mdlref_counter.slx');
```

Add a required password for modifying a protected model. If you do not add a password, you are prompted to set a password when you create a modifiable, protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'mdlref_counter','password');
```

Create a modifiable, protected model with support for code generation.

```
Simulink.ModelReference.protect('mdlref_counter','Mode',...
'CodeGeneration', 'Modifiable',true, 'Report',true);
```

Configure the unprotected model to support a new target.

```
set_param('mdlref_counter', 'SystemTargetFile', 'ert.tlc');
save_system('mdlref_counter');
```

Add support to the protected model for the new target. You are prompted for the modification password.

```
Simulink.ProtectedModel.addTarget('mdlref_counter');
```

Verify that support for the new target has been added to the protected model.

```
st = Simulink.ProtectedModel.getSupportedTargets('mdlref_counter')
```

Input Arguments

protectedModel — Model name
string or character vector

Protected model name, specified as a string or character vector.

Output Arguments

supportedTargets — List of target identifiers
cell array of character vectors

List of target identifiers for the targets that the protected model supports, specified as a cell array of character vectors.

See Also

Simulink.ModelReference.protect | Simulink.ProtectedModel.addTarget | Simulink.ProtectedModel.getConfigSet | Simulink.ProtectedModel.getCurrentTarget | Simulink.ProtectedModel.removeTarget | Simulink.ProtectedModel.setCurrentTarget

Topics

"Create Protected Models with Multiple Targets"
"Reference Protected Models from Third Parties"

Introduced in R2015a

Simulink.ProtectedModel.open

Open protected model

Syntax

```
Simulink.ProtectedModel.open(model)
Simulink.ProtectedModel.open(model,type)
```

Description

`Simulink.ProtectedModel.open(model)` opens a protected model. If you do not specify how to view the protected model, the software first tries to open the Web view. If the Web view is not enabled for the protected model, the software then tries to open the report. If you did not create a report, the software reports an error.

`Simulink.ProtectedModel.open(model,type)` opens a protected model using the specified viewing method. If you specify ‘webview’, the software opens the Web view for the protected model. If you specify ‘report’, the software opens the protected model report. If the method that you specify is not enabled, the software reports an error. The protected model is not opened.

Examples

Open a Protected Model

Open a protected model without a specified method.

Load the model and save a local copy.

```
sldemo_mdlref_counter
save_system('sldemo_mdlref_counter','mdlref_counter.slx');
```

Create a protected model enabling support for code generation and reporting.

```
Simulink.ModelReference.protect('mdlref_counter','Mode',...
'CodeGeneration', 'Report',true);
```

Open the protected model without specifying how to view it.

```
Simulink.ProtectedModel.open('mdlref_counter')
```

The protected model does not have Web view enabled, so the protected model report is opened.

Open a Protected Model Web View

Open a protected model, specifying the Web view.

Load the model and save a local copy.

```
sldemo_mdlref_counter
save_system('sldemo_mdlref_counter','mdlref_counter.slx');
```

Create a protected model with support for code generation, Web view, and reporting.

```
Simulink.ModelReference.protect('mdlref_counter','Mode',...
'CodeGeneration', 'Webview',true,'Report',true);
```

Open the protected model and specify that you want to see the Web view.

```
Simulink.ProtectedModel.open('mdlref_counter','webview')
```

The protected model Web view is opened.

Input Arguments

model — Model name
string or character vector

Protected model name, specified as a string or character vector.

type — Open method
'webview' | 'report'

Method for viewing the protected model. If you specify 'webview', the software opens the Web view for the protected model. If you specify 'report', the software opens the protected model report.

See Also

`Simulink.ModelReference.protect`

Introduced in R2015a

Simulink.ProtectedModel.removeTarget

Remove support for specified target from protected model

Syntax

`Simulink.ProtectedModel.removeTarget(protectedModel,targetID)`

Description

`Simulink.ProtectedModel.removeTarget(protectedModel,targetID)` removes code generation support for the specified target from a protected model. You must provide the modification password to make this update. Removing a target does not require access to the unprotected model.

Note You cannot remove the `sim` target. If you do not want the protected model to support simulation, use the `Simulink.ModelReference.modifyProtectedModel` function to change the protected model mode to `ViewOnly`.

Examples

Remove Target Support from a Protected Model

Remove a supported target from a protected model.

Load the model and save a local copy.

```
sldemo_mdlref_counter
save_system('sldemo_mdlref_counter','mdlref_counter.slx');
```

Add a required password for modifying a protected model. If you do not add a password, you are prompted to set a password when you create a modifiable, protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'mdlref_counter','password');
```

Create a modifiable, protected model with support for code generation.

```
Simulink.ModelReference.protect('mdlref_counter','Mode',...
'CodeGeneration', 'Modifiable',true, 'Report',true);
```

Configure the unprotected model to support a new target.

```
set_param('mdlref_counter', 'SystemTargetFile', 'ert.tlc');
save_system('mdlref_counter');
```

Add support to the protected model for the new target. You are prompted for the modification password.

```
Simulink.ProtectedModel.addTarget('mdlref_counter');
```

Verify that support for the new target has been added to the protected model.

```
st = Simulink.ProtectedModel.getSupportedTargets('mdlref_counter')
```

Remove support for the `ert` target from the protected model. You are prompted for the modification password.

```
Simulink.ProtectedModel.removeTarget('mdlref_counter','ert');
```

Verify that support for the `ert` target has been removed from the protected model.

```
st = Simulink.ProtectedModel.getSupportedTargets('mdlref_counter')
```

Input Arguments

`protectedModel` — Model name
string or character vector

Protected model name, specified as a string or character vector.

`targetID` — Target to be removed
string or character vector

Identifier for target to be removed, specified as a string or character vector.

See Also

`Simulink.ModelReference.modifyProtectedModel` | `Simulink.ModelReference.protect` | `Simulink.ProtectedModel.addTarget` | `Simulink.ProtectedModel.getConfigSet` | `Simulink.ProtectedModel.getCurrentTarget` | `Simulink.ProtectedModel.getSupportedTargets` | `Simulink.ProtectedModel.setCurrentTarget`

Topics
"Create Protected Models with Multiple Targets"

Introduced in R2015a

Simulink.ProtectedModel.setCurrentTarget

Configure protected model to use specified target

Syntax

```
Simulink.ProtectedModel.setCurrentTarget(protectedModel, targetID)
```

Description

`Simulink.ProtectedModel.setCurrentTarget(protectedModel, targetID)` configures the protected model to use the target that the target identifier specifies.

Note If you include the protected model in a model reference hierarchy, the software tries to change the current target to match the target of the parent model. If the software cannot match the target of the parent, it reports an error.

Examples

Set Current Target for Protected Model

After you get a list of supported targets, set the current target for a protected model.

Load the model and save a local copy.

```
sldemo_mdlref_counter
save_system('sldemo_mdlref_counter','mdlref_counter.slx');
```

Add a required password for modifying a protected model. If you do not add a password, you are prompted to set a password when you create a modifiable, protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'mdlref_counter','password');
```

Create a modifiable, protected model with support for code generation.

```
Simulink.ModelReference.protect('mdlref_counter','Mode',...
'CodeGeneration', 'Modifiable',true, 'Report',true);
```

Get a list of targets that the protected model supports.

```
st = Simulink.ProtectedModel.getSupportedTargets('mdlref_counter')
```

Configure the unprotected model to support a new target.

```
set_param('mdlref_counter', 'SystemTargetFile', 'ert.tlc');
save_system('mdlref_counter');
```

Add support to the protected model for the new target. You are prompted for the modification password.

```
Simulink.ProtectedModel.addTarget('mdlref_counter');
```

Verify that support for the new target has been added to the protected model.

```
st = Simulink.ProtectedModel.getSupportedTargets('mdlref_counter')
```

Configure the protected model to use the new target.

```
Simulink.ProtectedModel.setCurrentTarget('mdlref_counter','ert');
```

Verify that the current target is the new target.

```
ct = Simulink.ProtectedModel.getCurrentTarget('mdlref_counter')
```

Input Arguments

`protectedModel` — Model name

string or character vector

Protected model name, specified as a string or character vector.

`targetID` — Target identifier

string or character vector

Identifier for selected target, specified as a string or character vector.

See Also

`Simulink.ModelReference.protect` | `Simulink.ProtectedModel.addTarget` | `Simulink.ProtectedModel.getConfigSet` | `Simulink.ProtectedModel.getCurrentTarget` | `Simulink.ProtectedModel.getSupportedTargets` | `Simulink.ProtectedModel.removeTarget`

Topics

"Create Protected Models with Multiple Targets"
"Reference Protected Models from Third Parties"

Introduced in R2015a

Simulink.ProtectedModel.sign

Attach digital signature to protected model

Syntax

`Simulink.ProtectedModel.sign(protectedModel,certificateFile)`

Description

`Simulink.ProtectedModel.sign(protectedModel,certificateFile)` attaches a digital signature with the certificate `certificateFile` to the protected model `protectedModel`.

Examples

Sign a Protected Model

Protect a model, and then digitally sign it with a certificate.

Open and protect the model that you want to sign. For this example, protect the model `sldemo_mdlref_counter`.

```
sldemo_mdlref_counter
Simulink.ModelReference.protect('sldemo_mdlref_counter');
```

Locate the certificate file that you want to use to sign the protected model. Sign the model by using the certificate file.

```
Simulink.ProtectedModel.sign('sldemo_mdlref_counter.slxp','certificate_file.pfx');
```

In the dialog box, enter the password for the certificate file.

Input Arguments

`protectedModel` — Name of protected model
character vector | string scalar

Name of the protected model that you want to sign, specified as a character vector or string scalar. The protected model has a `.slxp` extension.

Example: `'my_model.slxp'`

`certificateFile` — Certificate file to use for signing
character vector | string scalar

Certificate file to use for signing the protected model, specified as a character vector or string scalar. The certificate must be a PKCS #12 file with the extension `.pfx` or `.p12`.

Example: `'my_cert.pfx'`

Example: `'InstitutionCertificate.p12'`

See Also

`Simulink.ModelReference.protect`

Topics

"Sign a Protected Model"

Introduced in R2020a

slConfigUIGetVal

Return current value for any model configuration parameter

Syntax

`value = slConfigUIGetVal(hDlg,hSrc,'OptionName')`

Description

`value = slConfigUIGetVal(hDlg,hSrc,'OptionName')` returns the value currently set in the dialog for any model configuration parameter.

The `slConfigUIGetVal` function is used in the context of a user-written `SelectCallback` function, which is triggered when you:

- Change System Target Files.
- Build the model.

Examples

Get Configuration Option Value

The `slConfigUIGetVal` function returns the value of the **Configuration Parameters** > **Code Generation** > **Interface** > **Advanced parameters** > **Terminate function required** option.

```
function usertarget_selectcallback(hDlg,hSrc)

  disp(['*** Select callback triggered:',sprintf('\n'), ...
        '  Uncheck and disable "Terminate function required".']);

  disp(['Value of IncludeMdlTerminateFcn was ', ...
        slConfigUIGetVal(hDlg,hSrc,'IncludeMdlTerminateFcn')]);

  slConfigUISetVal(hDlg,hSrc,'IncludeMdlTerminateFcn','off');
  slConfigUISetEnabled(hDlg,hSrc,'IncludeMdlTerminateFcn',false);
  hSrc.refreshDialog;
```

Input Arguments

`hDlg` — handle for STF callback functions
handle

Handle created in the context of a `SelectCallback` function and used by the System Target File Callback Interface functions. Pass this variable but do not set it or use it for another purpose.

Example: `hDlg`

`hSrc` — handle for STF callback functions
handle

Handle created in the context of a `SelectCallback` function and used by the System Target File Callback Interface functions. Pass this variable and use it to refresh the Configuration Parameters dialog box. Do not set it or use it for another purpose.

Example: `hSrc`

`OptionName` — TLC variable
TLC variable

Quoted name of the TLC variable defined for a custom target configuration option.

Example: `'myTLCvariable'`

Output Arguments

`value` — Option value
option value

Current value of the specified option. The data type of the return value depends on the data type of the option.

See Also

`slConfigUISetEnabled` | `slConfigUISetVal`

Topics
"Define and Display Custom Target Options"
"Custom Target Optional Features"

Introduced in R2006b

slConfigUISetEnabled

Enable or disable any model configuration parameter

Syntax

```
slConfigUISetEnabled(hDlg,hSrc,'OptionName',value)
```

Description

`slConfigUISetEnabled(hDlg,hSrc,'OptionName',value)` is used in the context of a user-written `SelectCallback` function, which is triggered when the custom target is selected in the System Target File Browser in the Configuration Parameters dialog box. You use `slConfigUISetEnabled` to enable or disable a specified model configuration parameter. To refresh the Configuration Parameters dialog, use `hSrc.refreshDialog`.

Examples

Disable Model Configuration Option

The `slConfigUISetEnabled` function disables the **Configuration Parameters** > **Code Generation** > **Interface** > **Advanced parameters** > **Terminate function required** option.

```
function usertarget_selectcallback(hDlg,hSrc)

  disp(['*** Select callback triggered:',sprintf('\n'), ...
        '  Uncheck and disable "Terminate function required".']);

  disp(['Value of IncludeMdlTerminateFcn was ', ...
        slConfigUIGetVal(hDlg,hSrc,'IncludeMdlTerminateFcn')]);

  slConfigUISetVal(hDlg,hSrc,'IncludeMdlTerminateFcn','off');
  slConfigUISetEnabled(hDlg,hSrc,'IncludeMdlTerminateFcn',false);
  hSrc.refreshDialog;
```

Input Arguments

`hDlg` — Handle for STF callback

handle

Handle created in the context of a `SelectCallback` function and used by the System Target File Callback Interface functions. Pass this variable but do not set it or use it for another purpose.

Example: `hDlg`

`hSrc` — Handle for STF callback

handle

Handle created in the context of a `SelectCallback` function and used by the System Target File Callback Interface functions. Pass this variable and use it to refresh the Configuration Parameters dialog box. Do not set it or use it for another purpose.

Example: `hSrc`

`'OptionName',value` — TLC variable name and set enable
`false` | `true`

Quoted name and set enable of the TLC variable defined for a model configuration parameter.

Example: `'myConfigVariable',false`

See Also
`slConfigUIGetVal` | `slConfigUISetVal`

Topics
"Define and Display Custom Target Options"
"Custom Target Optional Features"

Introduced in R2006b

slConfigUISetVal

Set value for any model configuration parameter

Syntax

`slConfigUISetVal(hDlg,hSrc,'OptionName',value)`

Description

`slConfigUISetVal(hDlg,hSrc,'OptionName',value)` is used in the context of a user-written `SelectCallback` function, which is triggered when the custom target is selected in the System Target File Browser in the Configuration Parameters dialog box. You use `slConfigUISetVal` to set the value of a specified target option.

Examples

Set Configuration Option Value

The `slConfigUISetVal` function sets the value `'off'` for the **Configuration Parameters** > **Code Generation** > **Interface** > **Advanced parameters** > **Terminate function required** option.

```
function usertarget_selectcallback(hDlg,hSrc)

  disp(['*** Select callback triggered:',sprintf('\n'), ...
        '  Uncheck and disable "Terminate function required".']);

  disp(['Value of IncludeMdlTerminateFcn was ', ...
        slConfigUIGetVal(hDlg,hSrc,'IncludeMdlTerminateFcn')]);

  slConfigUISetVal(hDlg,hSrc,'IncludeMdlTerminateFcn','off');
  slConfigUISetEnabled(hDlg,hSrc,'IncludeMdlTerminateFcn',false);
  hSrc.refreshDialog;
```

Input Arguments

`hDlg` — Handle for STF callback

handle

Handle created in the context of a `SelectCallback` function and used by the System Target File Callback Interface functions. Pass this variable but do not set it or use it for another purpose.

Example: `hDlg`

`hSrc` — Handle for STF callback

handle

Handle created in the context of a `SelectCallback` function and used by the System Target File Callback Interface functions. Pass this variable and use it to refresh the Configuration Parameters dialog box. Do not set it or use it for another purpose.

Example: `hSrc`

`'OptionName',value` — TLC variable name and value
option value

Quoted name and value of the TLC variable defined for a model configuration parameter.

Example: `'myConfigVariable',1`

See Also

`slConfigUIGetVal` | `slConfigUISetEnabled`

Topics
"Define and Display Custom Target Options"
"Custom Target Optional Features"

Introduced in R2006b

switchTarget

Select target for model configuration set

Syntax

```
switchTarget(myConfigObj,systemTargetFile,[])
switchTarget(myConfigObj,systemTargetFile,targetOptions)
```

Description

`switchTarget(myConfigObj,systemTargetFile,[])` changes the selected system target file for the active configuration set.

`switchTarget(myConfigObj,systemTargetFile,targetOptions)` sets the configuration parameters specified by `targetOptions`.

Examples

Get ConfigSet, Default Options, and Switch Target

This example shows how to get the active configuration set for `model`, and change the system target file for the configuration set.

```
% Get configuration set for model
myConfigObj = getActiveConfigSet(model);
% Switch system target file
switchTarget(myConfigObj,'ert.tlc',[]);
```

Get ConfigSet, Set Options, Switch Target

This example shows how to get the active configuration set for the current model (`gcs`), set various `targetOptions`, then change the system target file selection.

```
% Get configuration set for current model
myConfigObj=getActiveConfigSet(gcs);

% Specify target options
targetOptions.TLCOptions = '-aVarName=1';
targetOptions.MakeCommand = 'make_rtw';
targetOptions.Description = 'my target';
targetOptions.TemplateMakefile = 'grt_default_tmf';

% Define a system target file
targetSystemFile='grt.tlc';

% Switch system target file
switchTarget(myConfigObj,targetSystemFile,targetOptions);
```

Use `targetOptions` to verify values (optional).

```
% Verify values (optional)
targetOptions

          TLCOptions: '-aVarName=1'
         MakeCommand: 'make_rtw'
         Description: 'my target'
    TemplateMakefile: 'grt_default_tmf'
```

Get ConfigSet, Set Options for MSVC Solution Build, Switch Target to MSVC ERT

This example shows how to get the active configuration set for `model`, then change the system target file to the ERT Create Visual C/C++ Solution File for Embedded Coder.

```
model='rtwdemo_rtwintro';
open_system(model);

% Get configuration set for model
myConfigObj = getActiveConfigSet(model);

% Specify target options for MSVC build
targetOptions.MakeCommand = 'make_rtw';
targetOptions.Description = ...
     'Create Visual C/C++ Solution File for Embedded Coder';
targetOptions.TemplateMakefile = 'RTW.MSVCBuild';

% Switch system target file
switchTarget(myConfigObj,'ert.tlc',targetOptions);
```

Get ConfigSet, Set Options for Toolchain Build, and Switch Target

Use options to select default ERT target file, instead of `set_param(model,'SystemTargetFile','ert.tlc')`.

```
% use switchTarget to select toolchain build of defaul ERT target
model='rtwdemo_rtwintro';
open_system(model);

% Get configuration set for model
myConfigObj = getActiveConfigSet(model);

% Specify target options for toolchain build approach
targetOptions.MakeCommand = '';
targetOptions.Description = 'Embedded Coder';
targetOptions.TemplateMakefile = '';

% Switch system target file
switchTarget(myConfigObj,'ert.tlc',targetOptions);
```

Input Arguments

`myConfigObj` — Configuration set object
`object`

A configuration set object of `ConfigSet` or configuration reference object of `Simulink.ConfigSetRef`. Call `getActiveConfigSet` to get the configuration set object.

Example: `myConfigObj = getActiveConfigSet(model);`

`systemTargetFile` — Name of system target file
character vector

Specify the name of the system target file (such as `ert.tlc` for Embedded Coder or `grt.tlc` for Simulink Coder) as the name appears in the **System Target File Browser**.

Example: `systemTargetFile = 'ert.tlc';`

`targetOptions` — Structure with field values that provide configuration parameter options
struct

Structure with fields that define a code generation target options. You can choose to modify certain configuration parameters by filling in values in a structure field. If you do not want to use options, specify an empty structure (`[]`).

Field Values in targetOptions

Specify the structure field values of the `targetOptions`. If you choose not to specify options, use an empty structure (`[]`).

Example: `targetOptions = [];`

`TemplateMakefile` — Character vector specifying file name of template makefile
character vector

Example: `targetOptions.TemplateMakefile = 'RTW.MSVCBuild';`

`TLCOptions` — Character vector specifying TLC argument
character vector

Example: `targetOptions.TLCOptions = '-aVarName=1';`

`MakeCommand` — Character vector specifying make command MATLAB language file
character vector

Example: `targetOptions.MakeCommand = 'make_rtw';`

`Description` — Character vector specifying description of the system target file
character vector

Example: `targetOptions.Description = 'Create Visual C/C++ Solution File for Embedded Coder';`

See Also

`ConfigSet` | `Simulink.ConfigSetRef` | `getActiveConfigSet`

Topics

“Select a System Target File Programmatically”
“Configure a System Target File”
“Set Target Language Compiler Options”

Introduced in R2009b

target Package

Manage target hardware information

Description

Use these classes to manage target hardware information. For example, register new target hardware for code generation or set up target connectivity for external mode and processor-in-the-loop (PIL) simulations.

Classes

target.AddOn	Describe add-on properties for target type
target.Alias	Create alternative identifier for target object
target.API	Describe API details
target.APIImplementation	Describe API implementation details
target.ApplicationExecutionTool	Capture system command information to run application from MATLAB computer
target.ApplicationStatus	Describe status of application on target hardware
target.Board	Provide hardware board details
target.Breakpoint	Provide breakpoint details for debugger
target.Command	Capture system command for execution on MATLAB computer
target.BuildDependencies	Describe C and C++ build dependencies to associate with target hardware
target.CommunicationChannel	Describe communication channel properties
target.CommunicationInterface	Describe data I/O details for target hardware
target.CommunicationProtocolStack	Describe communication protocol parameters
target.Connection	Base class for target connection properties
target.ConnectionProperties	Describe target-specific connection properties
target.DebugIOTool	Debug byte stream I/O tool service interface
target.ExecutionService	Describe implementation of execution service for target application
target.ExecutionTool	MATLAB service interface for tool that manages application execution on target hardware
target.ExternalMode	Represent external mode protocol stack
target.ExternalModeConnectivity	Base class for external mode connectivity options
target.Function	Provide function signature information
target.HostProcessExecutionTool	Capture system command information to run target application from MATLAB computer
target.LanguageImplementation	Provide C and C++ compiler implementation details
target.MainFunction	Provide C and C++ dependencies for main function of target hardware application
target.MATLABDependencies	Describe MATLAB class and function dependencies
target.Object	Base class for target types
target.PILProtocol	Describe PIL protocol implementation for target hardware
target.Port	Describe connection via target hardware port
target.PortConnection	Describe target connection port
target.Processor	Provide target processor information

target.ProfilingFreezingOverhead	Capture freezing and unfreezing instrumentation overhead
target.ProfilingFunctionOverhead	Capture function instrumentation overhead
target.ProfilingTaskOverhead	Capture task instrumentation overhead
target.RS232Channel	Describe serial communication channel
target.SystemCommandExecutionTool	Capture system command information to run target application from MATLAB computer
target.TargetConnection	Provide details about connecting MATLAB computer to target hardware
target.TCPChannel	Describe TCP communication properties
target.Timer	Provide timer details for processor
target.Tools	Describe properties of tools for target hardware
target.UDPChannel	Describe UDP communication
target.XCP	Describe XCP protocol stack for target hardware
target.XCPExternalModeConnectivity	Represent connectivity options in external mode protocol stack
target.XCPPlatformAbstraction	Specify XCP platform abstraction layer for target hardware
target.XCPTCPIPTransport	Represent XCP TCP/IP transport protocol layer
target.XCPTransport	Base class for XCP transport protocol layer
target.XCPSerialTransport	Represent XCP serial transport protocol layer

Functions

target.add	Add target object to internal database
target.create	Create target object
target.export	Export target object data
target.get	Retrieve target objects from internal database
target.remove	Remove target object from internal database
target.upgrade	Upgrade existing definitions of hardware devices

See Also

Topics

"Register New Hardware Devices"
"Customise Connectivity for XCP External Mode Simulations"
"Set Up PIL Connectivity by Using target Package"

Introduced in R2019a

target.add

Package: `target`

Add target object to internal database

Syntax

```
objectsAdded = target.add(targetObject)
objectsAdded = target.add(targetObject, Name,Value)
```

Description

`objectsAdded = target.add(targetObject)` adds the specified target object to an internal database and returns a vector that contains the added objects. By default, the target data is available only for the current MATLAB session.

`objectsAdded = target.add(targetObject, Name,Value)` uses name-value arguments to controls persistence over MATLAB sessions and command-line output.

Examples

Specify Hardware Implementation

To specify a hardware implementation that persists over MATLAB sessions, use the `target.create` and `target.add` functions.

```
myLangImp = target.create('LanguageImplementation', ...
                          'Name', 'MyLanguageImplementation', ...
                          'Copy', 'ARM Compatible-ARM Cortex');

myProc = target.create('Processor','Name','MyProcessor');
myProc.LanguageImplementations = myLangImp;
objectsAdded = target.add(myProc, ...
                          'UserInstall',true, ...
                          'SuppressOutput',true);
```

Input Arguments

`targetObject` — Target object
object

Specify the target object that you want to add to the internal database.

Example: `target.add(`*`myTargetObject`*`);`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `target.add(myTargetObject, 'UserInstall', true);`

`UserInstall` — Target data persistence
false (default) | true

Control persistence of target data in an internal database:

- `true` –– Target data persists in internal database over multiple MATLAB sessions.
- `false` –– Target data is in internal database only for the current MATLAB session.

Example: `target.add(myTargetObject, 'UserInstall', true);`

Data Types: `logical`

`SuppressOutput` — Control command-line output
false (default) | true

Control command-line output of function:

- `true` –– Suppress command-line output from the function.
- `false` –– Provide information about the objects that the function adds to internal database.

Example: `target.add(myTargetObject, 'SuppressOutput', true);`

Data Types: `logical`

Output Arguments

`objectsAdded` — Objects added
object vector

Target objects that the function adds to internal database.

See Also

`target.create` | `target.get` | `target.remove`

Topics
"Register New Hardware Devices"

Introduced in R2019a

target.AddOn class

Package: `target`

Describe add-on properties for target type

Description

Use the `target.AddOn` class to capture custom properties that you can associate with these types of objects:

- `target.CommunicationChannel`
- `target.CommunicationProtocolStack`
- `target.Board`
- `target.Processor`
- `target.ConnectionProperties`

To extend the objects, assign the `target.AddOn` object to the `AddOns` property.

To create a `target.AddOn` object, use the `target.create` function.

Properties

`Name` — Add-on object name
character vector | string

Name of the reusable add-on object.

Example: `arduinoAddOn.Name = 'ArduinoBoardProperties';`

Attributes:

GetAccess	public
SetAccess	public

Methods

Public Methods

addProperty	Add a custom property to target.AddOn object

Examples

Part Number and Programmer for Arduino Board Definition

Add device-specific properties to a `target.Board` definition. Add information about the Arduino® part number and programmer to an Arduino board definition.

Create a board for Arduino Mega 2560.

```
mega = target.create('Board', ...
                     'Manufacturer', 'Arduino', ...
                     'Name', 'Mega 2560');
```

Create a `target.AddOn` object that specifies the Arduino board part number and programmer.

```
arduinoAddOn = target.create('AddOn');
arduinoAddOn.Name = 'ArduinoBoardProperties';
arduinoAddOn.addProperty('ArduinoPartNumber', 'String');
arduinoAddOn.addProperty('ArduinoProgrammer', 'String');
mega.AddOns = arduinoAddOn;
```

Specify the part number and programmer values.

```
mega.set('ArduinoPartNumber', 'm2560');
mega.set('ArduinoProgrammer', 'wiring');
```

You can parameterize the `avrdude` command for the deployment of an Arduino application.

```
command= target.create('Command');
command.String = 'avrdude';
command.Arguments = ('-p$(BOARD.Processor.ArduinoPartNumber)' ...
                     '-c$(PROCESSOR.ArduinoProgrammer)' ...
                     '-Uflash:w:$(EXE):i');
mega.Tools.DeployTools(1).Commands = [command];
```

See Also
`target.create`

Introduced in R2020b

addProperty

Class: `target.AddOn`
Package: `target`

Add a custom property to `target.AddOn` object

Syntax

`myAddOn.addProperty(propertyName, propertyType)`

Description

`myAddOn.addProperty(propertyName, propertyType)` adds the property `propertyName` of type `propertyType` to the object *`myAddOn`*.

Input Arguments

`propertyName` — Property name
character vector | string

Name of property that you want to add to object.

Example: `exampleAddOn.addProperty('myPrptyName', 'myPrptyType')`

`propertyType` — Property type
double | string | object

Type of property that you want to add to object.

Example: `exampleAddOn.addProperty('myPrptyName', 'myPrptyType')`

Examples

Add Property to `target.AddOn` Object

For workflow examples that use this method, see "Examples" on page 2-0.

See Also

`target.AddOn`

Introduced in R2020b

target.Alias class

Package: `target`

Create alternative identifier for target object

Description

Use the `target.Alias` class to create alternative identifiers for target objects. For example, if a target object has a long class identifier, you can create a `target.Alias` object that provides a short identifier for the target object.

To create a `target.Alias` object, use the `target.create` function.

Properties

`Name` — Alternative identifier

character vector | string

Alternative identifier for target object.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`For` — Target object

object

Original target object.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

Examples

Create Short Identifier for Target Object

For an example that uses this class, see “Create Alternative Identifier for Target Object”.

See Also

`target.create`

Topics

“Register New Hardware Devices”

Introduced in R2019a

target.API class

Package: `target`

Describe API details

Description

An API defines a set of entry-point functions for interaction with a software application or service. Use a `target.API` object to provide API details for target definition. Use this class with `target.APIImplementation` to describe how an API is used and built on target hardware.

To create a `target.API` object, use the `target.create` function.

Properties

Name — API name
character vector | string

Name of the API.

Example: `timerApi.Name = 'Linux Timer API';`

Attributes:

GetAccess	public
SetAccess	public

Language — API language
`target.Language` object

Programming language of the API implementation.

Attributes:

GetAccess	public
SetAccess	public

Functions — API Functions
`target.Function` object vector

Vector of `target.Function` objects that describe the set of entry-point functions that make up the API.

Attributes:

GetAccess	public
SetAccess	public

Examples

Describe `rtiostream` C API

This example provides implementation details for the `rtiostream` C API .

```
apiImp = target.create('APIImplementation', 'Name', ...
                       'x86 rtiostream Implementation');
apiImp.API = target.create('API', 'Name', 'rtiostream');
apiImp.BuildDependencies = target.create('BuildDependencies');
apiImp.BuildDependencies.SourceFiles = ...
      {fullfile('$(MATLAB_ROOT)', 'toolbox', ...
                'coder', 'rtiostream','src', ...
                'rtiostreamtcpip', 'rtiostream_tcpip.c')};
apiImp.MainFunction = target.create('MainFunction', ...
                      'Name', 'TCP RtIOStream Main');
apiImp.MainFunction.Arguments = {'-blocking', '1', '-port', '0'};
```

See Also

target.APIImplementation | target.Function | target.LanguageImplementation | target.create

Introduced in R2020b

target.APIImplementation class

Package: `target`

Describe API implementation details

Description

An API defines a set of entry-point functions for interaction with a software application or service. Use a `target.APIImplementation` object to provide details about how an API is used and built on target hardware.

To create a `target.APIImplementation` object, use the `target.create` function.

Properties

`Name` — API implementation name

character vector | string

Name of the `APIImplementation` object, which `target.get` uses as an identifier in data retrieval.

Attributes:

GetAccess	public
SetAccess	public

`API` — API description

`target.API` object

Description of the API implementation definition.

Attributes:

GetAccess	public
SetAccess	public

`BuildDependencies` — API implementation dependencies

`target.Dependencies` object

Source files, header files, and other dependencies that are needed for building and running the API on the target hardware.

Attributes:

GetAccess	public
SetAccess	public

`MainFunction` — main function requirements for API implementation

`target.MainFunction` object

Capture run-time dependencies such as `main` function arguments, initialization code, and `main` function build dependencies.

Attributes:

GetAccess	public
SetAccess	public

Examples

Describe rtiostream C API

This example provides implementation details for the `rtiostream` C API .

```
apiImp = target.create('APIImplementation', 'Name', ...
                       'x86 rtiostream Implementation');
apiImp.API = target.create('API', 'Name', 'rtiostream');
apiImp.BuildDependencies = target.create('BuildDependencies');
apiImp.BuildDependencies.SourceFiles = ...
      {fullfile('$(MATLAB_ROOT)', 'toolbox', ...
                'coder', 'rtiostream','src', ...
                'rtiostreamtcpip', 'rtiostream_tcpip.c')};
apiImp.MainFunction = target.create('MainFunction', ...
                      'Name', 'TCP RtIOStream Main');
apiImp.MainFunction.Arguments = {'-blocking', '1', '-port', '0'};
```

See Also
`target.API` | `target.BuildDependencies` | `target.MainFunction` | `target.create`

Introduced in R2020b

target.ApplicationExecutionTool class

Package: `target`

Capture system command information to run application from MATLAB computer

Description

Use the `target.ApplicationExecutionTool` class to capture system command information that is required to run an application from your development computer.

Class Attributes

`Abstract`	`true`
`HandleCompatible`	`true`

For information on class attributes, see "Class Attributes".

Properties

`Name` — Execution tool name
character vector | string

Name of the execution tool.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`Id` — Object identifier
character vector | string

Value of the `Name` property.

Attributes:

`GetAccess`	`public`
`SetAccess`	`private`

See Also

`target.Command` | `target.HostProcessExecutionTool` | `target.SystemCommandExecutionTool` | `target.create`

Topics

"Set Up PIL Connectivity by Using target Package"

Introduced in R2020b

target.ApplicationStatus class

Package: `target`

Describe status of application on target hardware

Description

Use the `target.ApplicationStatus` enumeration class to describe the status of your target application. The enumeration class contains these members.

Member	Application Status
`Running`	Application running on target hardware
`Stopped`	Application not running on target hardware
`Unknown`	Not known

Creation

`target.ApplicationStatus.`*`MemberName`* creates an object of the enumeration class.

Examples

PIL Target Connectivity with Debugger

For an example that uses the `target.ApplicationStatus` class, see “Use Debugger for PIL Target Connectivity”.

See Also

`target.ExecutionTool`

Topics

“Define Enumeration Classes”

Introduced in R2021a

target.Board class

Package: `target`

Provide hardware board details

Description

Use a `target.Board` object to provide MATLAB with data about your target hardware board, for example, CPU, communication, and tool information.

To create a `target.Board` object, use the `target.create` function.

Properties

`Name` — Board name
character vector | string

Name of the `target.Board` object, which `target.get` uses as an identifier in data retrieval.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`Processors` — Available processors
`target.Processor` object array

Array of `target.Processor` objects that provide descriptions of available processors for the board.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`CommunicationInterfaces` — Available communication interfaces
`target.CommunicationInterfaces` object array

Array of `target.CommunicationInterface` objects that provide descriptions of available communication interfaces for the board.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`CommunicationProtocolStacks` — Available communication protocols supported by board
`target.CommunicationProtocolStack` object array

Array of `target.CommunicationProtocolStack` that provide descriptions of the communication protocols for the board.

Attributes:

GetAccess	public
SetAccess	public

Tools — Tooling for interaction with board
target.Tools object

Collection of tooling descriptions associated with the board. For example, ApplicationExecutionTool to enable execution of applications on the target hardware.

Attributes:

GetAccess	public
SetAccess	public

Examples

Create Board Description

Create a description of a target hardware board. This code from “Set Up PIL Connectivity by Using target Package” shows how to create the description.

Create a board object that provides MATLAB with a description of processor attributes.

```
hostTarget = target.create('Board', 'Name', 'Host Intel processor');
```

Specify the processor for the board, for example, by reusing a supported processor.

```
hostTarget.Processors = target.get('Processor', ...
                                   'Intel-x86-64 (Linux 64)');
```

Create Communication Interface for Target Hardware

Create a communication interface for the target hardware board. This code snippet from “Set Up PIL Connectivity by Using target Package” shows how to create the interface.

```
comms = target.create('CommunicationInterface');

comms.Name = 'Linux TCP Interface';
comms.Channel = 'TCPChannel';
comms.APIImplementations = target.create('APIImplementation', ...
                                         'Name', 'x86 RTIOStream Implementation');
comms.APIImplementations.API = target.create('API', 'Name', 'RTIO Stream');

...

hostTarget.CommunicationInterfaces = comms;
```

Specify PIL Protocol Information

Specify PIL protocol information. This code snippet from “Set Up PIL Connectivity by Using target Package” shows how to specify the information.

```
pilProtocol = target.create('PILProtocol');
pilProtocol.Name = 'Linux PIL Protocol';
pilProtocol.SendBufferSize = 50000;
```

```
pilProtocol.ReceiveBufferSize = 50000;
hostTarget.CommunicationProtocolStacks = pilProtocol;
```

See Also

`target.Processor` | `target.create`

Topics

"Set Up PIL Connectivity by Using target Package"

Introduced in R2020b

target.Breakpoint class

Package: `target`

Provide breakpoint details for debugger

Description

Use the `target.Breakpoint` class to provide breakpoint details for a debugger.

To create a `target.Breakpoint` object, use the `target.create` function.

Properties

`Function` — Breakpoint function
string

Function that contains the breakpoint.

Attributes:

GetAccess	public
SetAccess	public

`File` — Breakpoint file
string

Path to the file that contains the breakpoint.

Attributes:

GetAccess	public
SetAccess	public

`Line` — Breakpoint line number
scalar

Line number of the breakpoint.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `int32`

Examples

PIL Target Connectivity with Debugger

For an example that uses the `target.Breakpoint` class, see “Use Debugger for PIL Target Connectivity”.

See Also
`target.DebugIOTool` | `target.create`

Topics
"Use Debugger for PIL Target Connectivity"

Introduced in R2021a

target.BuildDependencies class

Package: `target`

Describe C and C++ build dependencies to associate with target hardware

Description

Use the `target.BuildDependencies` object to:

- Describe C and C++ build dependencies, for example, source files and include paths.
- Associate the dependencies with your target hardware.

For example, you can use a `target.BuildDependencies` object to describe the build dependencies for a `target.APIImplementation` object.

Properties

`SourceFiles` — Source file dependencies
cell array of character vectors | string array

Specify path to source files.

Attributes:

GetAccess	public
SetAccess	public

`IncludeFiles` — Include file dependencies
cell array of character vectors | string array

Include file dependencies.

Attributes:

GetAccess	public
SetAccess	public

`IncludePaths` — Header file include path dependencies
cell array of character vectors | string array

Header file include path dependencies.

Attributes:

GetAccess	public
SetAccess	public

`Defines` — Macro definition dependencies
cell array of character vectors | string array

Macro definition dependencies.

Attributes:

GetAccess	public
SetAccess	public

StaticLibraries — Static library dependencies
cell array of character vectors | string array

Static library dependencies.

Attributes:

GetAccess	public
SetAccess	public

SharedLibraries — Shared library dependencies
cell array of character vectors | string array

Shared library dependencies.

Attributes:

GetAccess	public
SetAccess	public

Examples

Describe Implementation Build Dependencies for rtiostream C API

This example shows how you can describe the implementation build dependencies for the rtiostream C API.

```
apiImp = target.create('APIImplementation', 'Name', ...
                       'x86 rtiostream Implementation');
apiImp .API = target.create('API', 'Name', 'rtiostream');
apiImp .BuildDependencies = target.create('BuildDependencies');
apiImp .BuildDependencies.SourceFiles = ...
           {fullfile('$(MATLAB_ROOT)', 'toolbox', ...
                     'coder', 'rtiostream','src', ...
                     'rtiostreamtcpip', 'rtiostream_tcpip.c')};
apiImp.MainFunction = target.create('MainFunction', ...
                      'Name', 'TCP RtIOStream Main');
apiImp.MainFunction.Arguments = {'-blocking', '1', '-port', '0'};
```

See Also
target.APIImplementation | target.APIImplementation | target.create

Introduced in R2020b

target.Command class

Package: `target`

Capture system command for execution on MATLAB computer

Description

Use the `target.Command` class to capture a system command for execution on your development computer.

To create a `target.Command` object, use the `target.create` function. Create the object and then use separate steps to specify properties. Or, create the object and specify properties in a single step.

```
commandObject = target.create('Command', ...
                              stringPropertyValue, ...
                              argumentsPropertyValue)
```

Properties

`String` — Command string

string

Name of the application or script to call.

Attributes:

GetAccess	public
SetAccess	public

`Arguments` — Application or script arguments

string array | cell array of character vectors

String array or cell array of character vectors where each element represents a separate argument to the application or script defined in the `String` property.

Attributes:

GetAccess	public
SetAccess	public

Examples

Create `target.Command` Objects

Create this `target.Command` object by providing an application file path and arguments for the application.

```
cmdObj= target.create('Command');
cmdObj.String = 'pathToavrdude';
cmdObj.Arguments = {'-p$(BOARD.Processor.ArduinoPartNumber)' ...
                    '-c$(PROCESSOR.ArduinoProgrammer)' ...
                    '-Uflash:w:$(EXE):i'};
```

You can create the object in a single step.

```
cmdObj = target.create('Command', ...
                       'pathToavrdude', ...
                       {'-p$(BOARD.Processor.ArduinoPartNumber)' ...
                       '-c$(PROCESSOR.ArduinoProgrammer)' ...
                       '-Uflash:w:$(EXE):i'});
```

You can also specify the command and arguments by using a string. For example, to create a `target.Command` object for the command `echo` with arguments `-a` and `-b`, run:

```
cmdObj = target.create('Command', 'echo -a -b');
```

See Also

`target.ApplicationExecutionTool` | `target.HostProcessExecutionTool` | `target.SystemCommandExecutionTool` | `target.create`

Topics

"Set Up PIL Connectivity by Using target Package"

Introduced in R2020b

target.CommunicationChannel class

Package: `target`

Describe communication channel properties

Description

Use a `target.CommunicationChannel` object to describe communication channel properties for an I/O connection between two systems. You can use the object as part of a `target.Connection` object. target.`RS232Channel`, `target.TCPChannel`, and `target.UDPChannel` are examples of predefined communication channels.

Properties

Name — `target.CommunicationChannel` object name
character vector | string

Name of `target.CommunicationChannel` object.

Attributes:

GetAccess	public
SetAccess	public

Examples

Create Connection by Using TCP Communication Channel

This code from "Set Up PIL Connectivity by Using target Package" shows how to specify the connection between your development computer and target hardware. In the example, the target application runs on your development computer as a separate process and uses a TCP communication channel through `localhost`.

```
connection = target.create('TargetConnection');
connection.Name = 'Host Process Connection';
connection.Target = hostTarget;
connection.CommunicationChannel = target.create('TCPChannel');
connection.CommunicationChannel.Name = ...
                   'External Process TCPCommunicationChannel';
connection.CommunicationChannel.IPAddress = 'localhost';
connection.CommunicationChannel.Port = '0';
```

Note Using name-value arguments, you can create the connection object with this command:

```
timer = target.create('TargetConnection', ...
                      'Name', 'Host Process Connection', ...
                      'CommunicationType', 'TCPChannel', ...
                      'IPAddress', 'localhost', ...
                      'Port', '0')
```

See Also

`target.RS232Channel` | `target.TCPChannel` | `target.UDPChannel` | `target.create`

Topics
"Set Up PIL Connectivity by Using target Package"

Introduced in R2020b

target.CommunicationInterface class

Package: `target`

Describe data I/O details for target hardware

Description

Use the `target.CommunicationInterface` class to describe data transfer for your target hardware. Associate the communication channel for data transfer and the device driver API implementation with a `target.CommunicationInterface` object.

Properties

Name — `target.CommunicationInterface` object name
character vector | string

Name of `target.CommunicationInterface` object.

Attributes:

GetAccess	public
SetAccess	public

`Channel` — Communication channel
character vector | string

Type of communication channel that connects to the target hardware. For example, if you define PIL connectivity by using a `target.TargetConnection` over a `target.RS232Channel`, set this property to `'RS232Channel'`.

Attributes:

GetAccess	public
SetAccess	public

`APIImplementations` — Device driver API implementations for communication interface
`target.APIImplementations` object array

Details of the API implementations that use the communication interface channel to support data transfer to and from the target hardware. For example, an `rtiostream` API implementation for PIL connectivity.

Attributes:

GetAccess	public
SetAccess	public

Examples

Communication Interface for Target Hardware

Create the communication interface for the target hardware. This code snippet from “Set Up PIL Connectivity by Using target Package” shows how to create the interface.

```
comms = target.create('CommunicationInterface');
comms.Name = 'Linux TCP Interface';
comms.Channel = 'TCPChannel';
comms.APIImplementations = target.create('APIImplementation', ...
                                         'Name', 'x86 rtiostream Implementation');
comms.APIImplementations.API = target.create('API', 'Name', 'rtiostream');
comms.APIImplementations.BuildDependencies = target.create('BuildDependencies');
comms.APIImplementations.BuildDependencies.SourceFiles = ...
                                                        {fullfile('$(MATLABROOT)', ...
                                                        'toolbox', ...
                                                        'coder', ...
                                                        'rtiostream', ...
                                                        'src', ...
                                                        'rtiostreamtcpip', ...
                                                        'rtiostream_tcpip.c')};
comms.APIImplementations.MainFunction = target.create('MainFunction', ...
                                                      'Name', 'TCP RtIOStream Main');
comms.APIImplementations.MainFunction.Arguments = {'-blocking', '1', '-port', '0'};
hostTarget.CommunicationInterfaces = comms;
```

See Also

`target.APIImplementation` | `target.CommunicationChannel` | `target.create`

Topics

“Set Up PIL Connectivity by Using target Package”

Introduced in R2020b

target.CommunicationProtocolStack class

Package: `target`

Describe communication protocol parameters

Description

To describe target-specific parameters for different protocols, you can associate a `target.CommunicationProtocolStack` object with a `target.Board` object. One example of `target.CommunicationProtocolStack` is `target.PILProtocol`.

To create a `target.CommunicationProtocolStack` object, use the `target.create` function.

Properties

Name — API name
character vector | string

Name of the API.

Attributes:

GetAccess	public
SetAccess	public

Examples

Specify PIL Communication Protocol Object for `target.Board`

Specify PIL protocol information. This code snippet from "Set Up PIL Connectivity by Using target Package" shows how to specify the information.

```
pilProtocol = target.create('PILProtocol');
pilProtocol.Name = 'Linux PIL Protocol';
pilProtocol.SendBufferSize = 50000;
pilProtocol.ReceiveBufferSize = 50000;
hostTarget.CommunicationProtocolStacks = pilProtocol;
```

See Also

`target.PILProtocol` | `target.create`

Topics
"Set Up PIL Connectivity by Using target Package"

Introduced in R2020b

target.Connection class

Package: `target`

Base class for target connection properties

Description

Capture target connection properties by using a class derived from `target.Connection`. For example, use the `target.TargetConnection` class to describe a target connection from the development computer, inheriting properties from the `target.Connection` class.

Class Attributes

`Abstract`	`true`
`HandleCompatible`	`true`

For information on class attributes, see “Class Attributes”.

Properties

`Name` — Connection object
character vector | string

Name of the connection object.

Attributes:

`GetAccess`	`public`
`SetAccess`	`private`

`CommunicationChannel` — Communication channel description
`target.CommunicationChannel` object

Use this property to associate the connection with a specific `target.CommunicationChannel` that is used to determine common connection properties between the connected systems. For example, `target.RS232Channel`.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

See Also

`target.CommunicationChannel` | `target.TargetConnection` | `target.create`

Topics

“Set Up PIL Connectivity by Using target Package”

Introduced in R2020b

target.ConnectionProperties class

Package: target

Describe target-specific connection properties

Description

Use a target.ConnectionProperties object to describe connection properties that are required for the target hardware to connect to another system. The class is used as a base class for specific connection properties such as target.Port. You can extend the class to provide custom properties by using the target.AddOn class.

Properties

AddOns — Custom property add-on objects

target.AddOns object array

To provide additional custom properties, associate one or more target.AddOn objects with the target.ConnectionProperties object.

Attributes:

GetAccess	public
SetAccess	public

Methods

Public Methods

get	Get name of custom add-on property
set	Set property value

See Also

target.create

Introduced in R2020b

get

Class: `target.ConnectionProperties`
Package: `target`

Get name of custom add-on property

Syntax

```
value = get(propertyName)
```

Description

`value = get(propertyName)` returns the name of a custom add-on property or `target.ConnectionProperties` property.

Input Arguments

`propertyName` — Property name
string

Name of a custom add-on property or `target.ConnectionProperties` class property.

Output Arguments

`Value` — Property value
string

Property value that corresponds to the `propertyName`.

See Also

`target.ConnectionProperties`

Introduced in R2020b

set

Class: `target.ConnectionProperties`
Package: `target`

Set property value

Syntax

`set(propertyName,propertyValue)`

Description

`set(propertyName,propertyValue)` sets the name of a custom add-on property or `target.ConnectionProperties` property to a specified value.

Input Arguments

`propertyName` — Property name
string

Name of a custom add-on property or `target.ConnectionProperties` class property.

`propertyValue` — Property value
string

Value for `propertyName`.

See Also

`target.ConnectionProperties`

Introduced in R2020b

target.create

Package: `target`

Create target object

Syntax

```
targetObject = target.create(targetType)
targetObject = target.create(targetType,Name,Value)
```

Description

`targetObject = target.create(targetType)` creates and returns an object of the specified class.

`targetObject = target.create(targetType,Name,Value)` configures the object using one or more name-value arguments.

Note You can create an object and specify properties in one step for these classes:

- `target.Command`
- `target.TargetConnection`
- `target.Timer`

Examples

Create New Hardware Implementation

For workflow examples that use this function, see:

- "Specify Hardware Implementation for New Device"
- "Create Hardware Implementation by Modifying Existing Implementation"
- "Create Hardware Implementation by Reusing Existing Implementation"

Input Arguments

targetType — Target type

character vector | string

Specify class of object. For example, specifying:

- `'Processor'` creates a `target.Processor` object.
- `'LanguageImplementation'` creates a `target.LanguageImplementation` object.

- 'Alias' creates a target.Alias object.

For the full list of supported types, see target.

Example: 'Processor'

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: myProc = target.create('Processor', 'Name', 'myProcessor', 'Manufacturer', 'myProcessorManufacturer');

Copy — Copy existing target feature object
character vector | string

Create a target object by copying values from an existing target object. For example:

```
myLangImp = target.create('LanguageImplementation', ...
                          'Name', 'myLanguageImplementation', ...
                          'Copy', 'ARM Compatible-ARM Cortex');
```

***propertyName* — Property name**
character vector | string

Create the target object with properties that are set to values that you specify.

Output Arguments

targetObject — Target object
object

The object that is created and returned. For example, the object is a:

- target.Processor object if targetType is 'Processor'
- target.LanguageImplementation object if targetType is 'LanguageImplementation'
- target.Alias object if targetType is 'Alias'

See Also

target.add | target.get | target.remove

Topics

"Register New Hardware Devices"

Introduced in R2019a

target.DebugIOTool class

Package: `target`

Debug byte stream I/O tool service interface

Description

To provide a service interface for a tool that starts and tracks an application on the target hardware through a debugger, define a subclass that derives from the `target.DebugIOTool` class . The tool controls the debugger's interaction with the executing application and reads and writes to memory through the debugger. The `target.DebugIOTool` class inherits from `target.ExecutionTool`.

Properties

`Breakpoints` — Application breakpoints
target.Breakpoint object array

Application breakpoints that you must set in the debugger.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

Methods

Public Methods

close	Close target application harness
getApplicationStatus	Get status of target application
getStandardError	Return standard error stream from target application
getStandardOutput	Return standard output stream from target application
loadApplication	Load target application into harness
open	Open target application harness
read	Read specified byte stream from variable in memory
startApplication	Start execution of target application
stopApplication	Stop execution of target application
unloadApplication	Unload target application from harness
write	Write byte stream to variable in memory

Examples

PIL Target Connectivity with Debugger

For an example that uses the `target.DebugIOTool` class, see “Use Debugger for PIL Target Connectivity”.

See Also

`target.ExecutionService` | `target.ExecutionTool`

Topics
"Use Debugger for PIL Target Connectivity"
"DebugIOTool Template"

Introduced in R2021a

read

Class: `target.DebugIOTool`
Package: `target`

Read specified byte stream from variable in memory

Syntax

```
[variableData, errFlag] = myDebugIOTool.read(byteStream, variable)
```

Description

`[variableData, errFlag] = myDebugIOTool.read(byteStream, variable)`, through the debugger, reads the specified byte stream from the variable in memory. The method returns the byte stream and an error flag.

Input Arguments

`byteStream` — Byte stream

string

Byte stream to write to memory.

Example: `[vd,ef] = myDebugIOTool.read('myByteStream', myVariable)`

`variable` — Variable in memory

vector

Variable in memory.

Example: `[vd,ef] = myDebugIOTool.read('myByteStream', myVariable)`

Data Types: `uint64`

Output Arguments

`variableData` — Byte stream

vector

Byte stream from variable in memory.

`errFlag` — Error flag

true | false

Outcome of the write operation:

- `true` –– Error occurred during write operation.
- `false` –– Write operation completed.

See Also
`target.DebugIOTool` | `write`

Introduced in R2021a

write

Class: `target.DebugIOTool`
Package: `target`

Write byte stream to variable in memory

Syntax

```
errFlag = myDebugIOTool.write(byteStream, variable)
```

Description

`errFlag = myDebugIOTool.write(byteStream, variable)`, through the debugger, writes the specified byte stream to the variable in memory. The method returns an error flag.

Input Arguments

`byteStream` — Byte stream
string

Byte stream to write to memory.

Example: `ef = myDebugIOTool.write('myByteStream', myVariable)`

`variable` — Variable in memory
vector

Destination variable in memory.

Example: `ef = myDebugIOTool.write('myByteStream', myVariable)`

Data Types: `uint64`

Output Arguments

`errFlag` — Error flag
true | false

Outcome of the write operation:

- `true` –– Error occurred during write operation.
- `false` –– Write operation completed.

See Also
`target.DebugIOTool` | `write`

Introduced in R2021a

target.ExecutionService class

Package: `target`

Describe implementation of execution service for target application

Description

Use the `target.ExecutionService` class, which inherits from `target.ApplicationExecutionTool`, to describe the execution service implementation for running an application on the target computer. The implementation uses MATLAB code.

To create a `target.ExecutionService` object, use the `target.create` function.

Properties

`Name` — Execution service name

character vector | string

The name of the target application execution service.

Attributes:

GetAccess	public
SetAccess	public

`APIImplementation` — API implementation

target.APIImplementation object

A `target.APIImplementation` object that describes the MATLAB data model service implementation.

Attributes:

GetAccess	public
SetAccess	public

`HostOperatingSystemRequirements` — Development computer operating system requirements

'All' (default) | 'Unix' | 'Linux' | 'MacOS' | 'Windows'

MathWorks system requirements for operating system of development computer.

Attributes:

GetAccess	public
SetAccess	public

`Id` — Object identifier

character vector | string

Value of the `Name` property.

Attributes:

`GetAccess`	`public`
`SetAccess`	`private`

Examples

Use `target.ExecutionService` to Describe Application Execution

For an example that uses the `target.ExecutionService` class to describe the application execution within a debugger, see "Use Debugger for PIL Target Connectivity".

See Also

`target.ApplicationExecutionTool` | `target.ExecutionTool` | `target.create`

Topics

"Use Debugger for PIL Target Connectivity"

Introduced in R2021a

target.ExecutionTool class

Package: `target`

MATLAB service interface for tool that manages application execution on target hardware

Description

Use the `target.ExecutionTool` class to provide a MATLAB service interface for a tool that manages the application execution on the target hardware.

Properties

`Application` — Target application

string

Name of application to run on target hardware, which is set by MATLAB at runtime.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`ApplicationCommandLineArguments` — Command-line arguments

string array

Command-line arguments for the target application, which is set by MATLAB at runtime.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

Methods

Public Methods

close	Close target application harness
getApplicationStatus	Get status of target application
getStandardError	Return standard error stream from target application
getStandardOutput	Return standard output stream from target application
loadApplication	Load target application into harness
open	Open target application harness
startApplication	Start execution of target application
stopApplication	Stop execution of target application
unloadApplication	Unload target application from harness

Examples

Use `target.ExecutionTool` to Manage Application Execution

For an example that uses the `target.ExecutionTool` class to manage the application execution on the target hardware, see “Use Debugger for PIL Target Connectivity”.

See Also
`target.DebugIOTool`

Topics
"Use Debugger for PIL Target Connectivity"

Introduced in R2021a

close

Class: `target.ExecutionTool`
Package: `target`

Close target application harness

Syntax

```
errFlag = myExecutionTool.close()
```

Description

`errFlag = myExecutionTool.close()` closes the harness for the target application (that is managed by the associated execution tool) if it exists. An example of an application harness is a simulator or debugger. The method returns an error flag.

Output Arguments

`errFlag` — Error flag
true | false

Outcome of the operation:

- `true` –– An error occurred while closing the application harness.
- `false` –– Method completed without error. Application harness closed or does not exist.

See Also

`open` | `target.ExecutionTool`

Introduced in R2021a

getApplicationStatus

Class: `target.ExecutionTool`
Package: `target`

Get status of target application

Syntax

```
[applicationStatus, errFlag] = myExecutionTool.getApplicationStatus()
```

Description

`[applicationStatus, errFlag] = myExecutionTool.getApplicationStatus()` returns the status of the target application (managed by the execution tool) and an outcome flag.

Output Arguments

`applicationStatus` — Target application status
`target.ApplicationStatus` member

The status of the target application that is being managed by the execution tool.

`errFlag` — Error flag
true | false

Outcome of the operation:

- `true` –– An error occurred while trying to obtain status of target application.
- `false` –– Status of target application obtained without error.

See Also
`target.ExecutionTool`

Introduced in R2021a

getStandardError

Class: `target.ExecutionTool`
Package: `target`

Return standard error stream from target application

Syntax

```
[stdErrorStream, errFlag] = myExecutionTool.getStandardError()
```

Description

`[stdErrorStream, errFlag] = myExecutionTool.getStandardError()` returns, as a string, the standard error stream from the target application that is being managed by the associated execution tool. The method also returns an error flag.

Output Arguments

`stdErrorStream` — Standard error stream
string

The standard error stream from the beginning of the execution of the target application.

`errFlag` — Error flag
true | false

Outcome of the operation:

- `true` –– Error occurred while retrieving standard error stream from target application.
- `false` –– Standard error stream from target application returned.

See Also

`getStandardOutput` | `target.ExecutionTool`

Introduced in R2021a

getStandardOutput

Class: `target.ExecutionTool`
Package: `target`

Return standard output stream from target application

Syntax

```
[stdOutputStream, errFlag] = myExecutionTool.getStandardOutput()
```

Description

`[stdOutputStream, errFlag] = myExecutionTool.getStandardOutput()` returns, as a string, the standard output stream from the target application that is being managed by the associated execution tool. The method also returns an error flag.

Output Arguments

`stdOutputStream` — Standard output stream
string

The standard output stream from the beginning of the execution of the target application.

`errFlag` — Outcome flag
true | false

Outcome of the operation:

- `true` –– Error occurred while retrieving standard output stream from target application.
- `false` –– Standard output stream from target application returned.

See Also
`getStandardError` | `target.ExecutionTool`

Introduced in R2021a

loadApplication

Class: `target.ExecutionTool`
Package: `target`

Load target application into harness

Syntax

```
outcomeFlag = myExecutionTool.loadApplication()
```

Description

`outcomeFlag = myExecutionTool.loadApplication()` loads the target application (managed by the associated execution tool) into the application harness if the harness exists. An example of an application harness is a simulator or debugger. The method returns an error flag.

Output Arguments

`errFlag` — Error flag
true | false

Outcome of the operation:

- `true` –– An error occurred while loading the target application into the harness.
- `false` –– Target application loaded into the harness or the harness does not exist.

See Also

`target.ExecutionTool` | `unloadApplication`

Introduced in R2021a

open

Class: `target.ExecutionTool`
Package: `target`

Open target application harness

Syntax

```
errFlag = myExecutionTool.open()
```

Description

`errFlag = myExecutionTool.open()` opens the harness for the target application (that is managed by the associated execution tool) if the harness exists. An example of a target application harness is a simulator or debugger. The method returns an error flag.

Output Arguments

`errFlag` — Error flag

true | false

Outcome of the operation:

- `true` –– Error occurred while opening application harness.
- `false` –– Application harness opened or does not exist.

See Also

`close` | `target.ExecutionTool`

Introduced in R2021a

startApplication

Class: `target.ExecutionTool`
Package: `target`

Start execution of target application

Syntax

```
errFlag = myExecutionTool.startApplication()
```

Description

`errFlag = myExecutionTool.startApplication()` starts the execution of the target application (that is managed by the associated execution tool) and returns an error flag.

Output Arguments

`errFlag` — Error flag
true | false

Outcome of the operation:

- `true` –– Error occurred while trying to start target application.
- `false` –– Target application started.

See Also

`stopApplication` | `target.ExecutionTool`

Introduced in R2021a

stopApplication

Class: `target.ExecutionTool`
Package: `target`

Stop execution of target application

Syntax

```
errFlag = myExecutionTool.stopApplication()
```

Description

`errFlag = myExecutionTool.stopApplication()` stops the execution of the target application (that is managed by the associated execution tool) and returns an error flag.

Output Arguments

`errFlag` — Error flag
true | false

Outcome of the operation:

- `true` –– Error occurred while trying to stop target application.
- `false` –– Target application stopped.

See Also

`startApplication` | `target.ExecutionTool`

Introduced in R2021a

unloadApplication

Class: `target.ExecutionTool`
Package: `target`

Unload target application from harness

Syntax

```
outcomeFlag = myExecutionTool.unloadApplication()
```

Description

`outcomeFlag = myExecutionTool.unloadApplication()` unloads the target application (managed by the associated execution tool) from the application harness if the harness exists. An example of an application harness is a simulator or debugger. The method returns an error flag.

Output Arguments

`errFlag` — Error flag
true | false

Outcome of the operation:

- `true` –– An error occurred while unloading the target application from the harness.
- `false` –– Target application unloaded from the harness or the harness does not exist.

See Also

`loadApplication` | `target.ExecutionTool`

Introduced in R2021a

target.export

Package: `target`

Export target object data

Syntax

```
target.export(targetObject)
target.export(targetObject,Name,Value)
```

Description

`target.export(targetObject)` exports the specified target object data to a MATLAB function, `registerTargets.m`. Use the generated file to share target feature data between sessions and computers. When you run `registerTargets.m`, it recreates the target object and adds the object to an internal database.

`target.export(targetObject,Name,Value)` exports the specified target object data using one or more name-value pair arguments.

Examples

Share Hardware Device Data

You can share hardware device data across computers and users.

Specify two hardware devices.

```
langImp1 = target.create('LanguageImplementation', ...
                         'Name', 'MyLanguageImplementation1', ...
                         'Copy', 'ARM Compatible-ARM Cortex');

langImp2 = target.create('LanguageImplementation', ...
                         'Name', 'MyLanguageImplementation2', ...
                         'Copy', 'Atmel-AVR');

myProc1 = target.create('Processor','Name','MyProcessor1');
myProc1.LanguageImplementations = [langImp1, langImp2];
objectsAdded1 = target.add(myProc1, ...
                         'UserInstall',true, ...
                         'SuppressOutput',true);

myProc2 = target.create('Processor','Name','MyProcessor2');
objectsAdded2 = target.add(myProc2, ...
                         'UserInstall',true, ...
                         'SuppressOutput',true);
```

Run the `target.export` function.

```
target.export([myProc1, myProc2], 'FileName', 'exportMyProcFunction')
```

The function generates `exportMyProcFunction.m` in the current working folder. Use the generated function to share hardware device data across computers and users. For example, on another computer, run this command:

```
addedObjects = exportMyProcFunction;
```

The generated function recreates and adds the objects to an internal database.

If you want the hardware device data to persist over MATLAB sessions, run:

```
addedObjects = exportMyProcFunction('UserInstall',true);
```

Input Arguments

`targetObject` — Target object

object vector

Specify the target objects that you want to export.

Example: `target.export(myTargetObject);`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `target.export(myTargetObject,'FileName', 'exportMyTargetFn');`

`FileName` — File name

character vector | string

Specify name of MATLAB function file that contains exported target data.

`Overwrite` — Overwrite flag

false (default) | true

- `true` –– Overwrite MATLAB function file if it exists.
- `false` –– Generate error if MATLAB function file exists. File is not overwritten.

See Also

`target.add` | `target.create`

Topics

"Register New Hardware Devices"

Introduced in R2019b

target.ExternalMode class

Package: target

Represent external mode protocol stack

Description

Use the target.ExternalMode class, which is a subclass of target.CommunicationProtocolStack, to specify the external mode protocol stack for your target hardware.

To create a target.ExternalMode object, use the target.create function.

Properties

Connectivities — External mode connectivity options

target.ExternalModeConnectivity object array

Provide connectivity options for the external mode protocol stack. The array can contain only one target.ExternalModeConnectivity object for a specific transport protocol. For example, the array can contain one object for XCP on TCP/IP and another object for XCP on Serial.

Attributes:

GetAccess	public
SetAccess	public

Examples

Specify External Mode Protocol Stack for Target Hardware

This code snippet from "Customise Connectivity for XCP External Mode Simulations" shows how to specify the external mode protocol stack for your target hardware.

```
extModeTCPConnectivity = ...
             target.create('XCPExternalModeConnectivity', ...
             'Name', 'External Mode TCP Connectivity', ...
             'XCP', xcpTCPIPConfiguration);

externalMode = target.create('ExternalMode', ...
                'Name', 'External Mode', ...
                'Connectivities', extModeTCPConnectivity);

board.CommunicationProtocolStacks = externalMode;
```

See Also

target.Board | target.CommunicationProtocolStack | target.ExternalModeConnectivity | target.XCP | target.XCPExternalModeConnectivity | target.create

Topics

"Customise Connectivity for XCP External Mode Simulations"

Introduced in R2021a

target.ExternalModeConnectivity class

Package: `target`

Base class for external mode connectivity options

Description

The `target.ExternalModeConnectivity` class is a base class for representing connectivity options that are available in the external mode protocol stack. The `target.XCPExternalModeConnectivity` class is derived from this base class.

Class Attributes

`Abstract`	`true`
`HandleCompatible`	`true`

For information on class attributes, see “Class Attributes”.

See Also

`target.Board` | `target.CommunicationProtocolStack` | `target.ExternalMode` | `target.XCPExternalModeConnectivity` | `target.create`

Topics

“Customise Connectivity for XCP External Mode Simulations”

Introduced in R2021a

target.Function class

Package: `target`

Provide function signature information

Description

Use the `target.Function` class, which inherits functionality from `target.Data`, to provide function signature information.

To create a `target.Function` object, use the `target.create` function.

Properties

`ReturnType` — Data type returned
character vector | string

Data type of value that the associated function returns.

Attributes:

GetAccess	public
SetAccess	public

`Name` — Function name
character vector | string

Name of function.

Attributes:

GetAccess	public
SetAccess	public

Examples

Create Function Signature

Create the function signature for a timer by using the `target.Function` class.

Create the signature for a function that returns the data type `uint64` and the function name `timestamp_x86`.

```
timerSignature = target.create('Function');
timerSignature.Name = 'timestamp_x86';
timerSignature.ReturnType = 'uint64';
```

See Also
`target.create`

Introduced in R2020b

target.get

Package: `target`

Retrieve target objects from internal database

Syntax

```
targetObject = target.get(targetType, targetObjectId)
tFOList = target.get(targetType)
tFOList = target.get(targetType, Name, Value)
```

Description

`targetObject = target.get(targetType, targetObjectId)` retrieves a target object from an internal database.

`tFOList = target.get(targetType)` returns a list of *`targetType`* objects that are stored in the internal database.

`tFOList = target.get(targetType, Name, Value)` returns a list of *`targetType`* objects that have properties that match the name-value pairs.

Examples

Remove Target Object

This example shows how you can remove a `target.LanguageImplementation` object associated with an object identifier, *`myLanguageImplementationID`*.

Retrieve the object from the internal database.

```
objectToRemove = target.get('LanguageImplementation', myLanguageImplementationID);
```

Remove the object.

```
target.remove(objectToRemove);
```

Create Board Description

This example shows how to create a `target.Board` object that provides MATLAB with a description of processor attributes. It uses `target.get` to retrieve the description of a supported processor.

Create a board object.

```
hostTarget = target.create('Board', 'Name', 'Host Intel processor');
```

Specify the processor for the board by reusing a supported processor.

```
hostTarget.Processors = target.get('Processor', ...
                                   'Intel-x86-64 (Linux 64)');
```

Input Arguments

targetType — Target type
character vector | string

Specify the class of the object that you want to retrieve. For example, to retrieve:

- A `target.Processor` object, specify `'Processor'` .
- A `target.LanguageImplementation` object, specify `'LanguageImplementation'`.

For the list of supported classes, see `target Package`.

targetObjectId — Target object identifier
character vector | string

Specify the unique identifier of the object that you want to retrieve, that is, the `Id` property value of the object.

Output Arguments

targetObject — Target object
object

Retrieved target object. For example:

- If `targetType` is `'Processor'`, the returned object is a `target.Processor` object.
- If `targetType` is `'LanguageImplementation'`, the returned object is a `target.LanguageImplementation` object.

For the list of supported classes, see `target Package`.

tFOList — Target object list
object vector

List of retrieved target objects.

See Also

`target.add` | `target.create` | `target.remove`

Topics

"Register New Hardware Devices"

Introduced in R2019a

target.HostProcessExecutionTool class

Package: `target`

Capture system command information to run target application from MATLAB computer

Description

Use the `target.HostProcessExecutionTool` class to capture system command information that is required to run the target application from your development computer.

Properties

`Name` — Execution tool name
character vector | string

Name of the execution tool.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`StartCommand` — Command list to run application
`target.Command` object

A `target.Command` object that provides a system command for running the application. The command in the list starts the application process.

This property must not be empty.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`StopCommand` — Command list to stop application
`target.Command` object

A `target.Command` object that provides a system command to stop the application execution.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`Id` — Object identifier
character vector | string

Value of the `Name` property.

Attributes:

`GetAccess`	`public`
`SetAccess`	`private`

See Also

`target.ApplicationExecutionTool` | `target.SystemCommandExecutionTool` | `target.create`

Topics

"Set Up PIL Connectivity by Using target Package"

Introduced in R2020b

target.LanguageImplementation class

Package: `target`

Provide C and C++ compiler implementation details

Description

Use the `target.LanguageImplementation` class to provide implementation details about the C and C++ compiler for your target hardware. For example, byte ordering.

To create a `target.LanguageImplementation` object, use the `target.create` function.

Properties

`AtomicFloatSize` — Largest atomic float size
integer

Size in bits of the largest floating-point data type that you can atomically load and store on the hardware

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

Data Types: `int32`

`AtomicIntegerSize` — Largest atomic integer size
integer

Size in bits of the largest integer that you can atomically load and store on the hardware

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

Data Types: `int32`

`Endianess` — Byte ordering
`'Little'` (default) | `'Big'` | `'Unspecified'`

Byte ordering implemented by target hardware.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`DataTypes` — Data type definitions
object

`target.DataTypes` object that provides C and C++ data type definitions through properties described in this table.

Property Name	Purpose
`Char`	`target.datatype.Char` object for `char`.
`Short`	`target.datatype.Short` object for `short`.
`Int`	`target.datatype.Int` object for `int`.
`Long`	`target.datatype.Long` object for `long`.
`LongLong`	`target.datatype.LongLong` object for `longlong`.
`Half`	`target.datatype.Half` object for a half precision data type that the target uses.
`Float`	`target.datatype.Float` object for `float`.
`Double`	`target.datatype.Double` object for `double`.
`Pointer`	`target.datattype.Pointer` object for `pointer`.
`SizeT`	`target.datatype.SizeT` object for `size_t`.
`PtrDiffT`	`target.datatype.PtrDiffT` object for `ptrdiff_t`.

Attributes:

GetAccess	public
SetAccess	private

Id — Object identifier

character vector | string

Value of `Name` property.

Attributes:

GetAccess	public
SetAccess	private

Name — Name

character vector | string

Name of the target language implementation

Attributes:

GetAccess	public
SetAccess	public

WordSize — Native word size

integer

Native word size for the target hardware.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `int32`

Examples

Create New Hardware Implementation

For examples that use this class, see:

- "Specify Hardware Implementation for New Device"
- "Create Hardware Implementation by Modifying Existing Implementation"
- "Create Hardware Implementation by Reusing Existing Implementation"

See Also

`target.Processor` | `target.create`

Topics

"Register New Hardware Devices"

Introduced in R2019a

target.MainFunction class

Package: `target`

Provide C and C++ dependencies for `main` function of target hardware application

Description

Use the `target.MainFunction` class to provide `main` function dependencies for an application `main` function that runs on your target hardware. For example, C and C++ initialization and termination code, `include` preprocessor directives, and specification of `main` function arguments for the application.

To create a `target.MainFunction` object, use the `target.create` function.

Properties

`Name` — Dependency collection
character vector | string

Name of the collection of `main` dependencies.

Attributes:

```
GetAccess                                  public
SetAccess                                  public
```

`Dependencies` — Build dependencies
`target.BuildDependencies` object

Compiler build tool dependencies of the `main` function, which include header files, source files, and libraries.

Attributes:

```
GetAccess                                  public
SetAccess                                  public
```

`Arguments` — Command-line arguments
string array

Capture run-time command-line argument dependencies.

Attributes:

```
GetAccess                                  public
SetAccess                                  public
```

`IncludeFiles` — #include files
string array

Array of header files that must be included in a target `main` function by using the preprocessor directive `#include "path-spec"`.

Attributes:

GetAccess	public
SetAccess	public

SystemIncludeFiles — System #include files
string array

Array of header files that must be included in a target `main` function by using the preprocessor directive `#include <path-spec>`.

Attributes:

GetAccess	public
SetAccess	protected

InitializationCode — Target main initialization
character vector | string

Formatted string of C or C++ code that the `main` function uses to initialize target resources.

Attributes:

GetAccess	public
SetAccess	public

TerminationCode — Target main termination
character vector | string

Formatted string of C or C++ code that the `main` function uses to terminate target resources.

Attributes:

GetAccess	public
SetAccess	public

Examples

Specify Target-Specific main Function Dependencies

Create a `target.MainFunction` object and associate it with a `target.Board object`, which captures the `main` function dependencies for an Arduino board. Workflows, such as processor-in-the-loop (PIL), can use this information when generating a `main` function for an application that runs on the target hardware.

```
board = target.create('Board', 'Name', 'Arduino Board')
mainFunction = target.create('MainFunction');
mainFunction.Name = 'Arduino Main Dependencies';

mainFunction.IncludeFiles = { 'Arduino.h' };
mainFunction.InitializationCode = fileread('arduino_main_initialization.c');

board.MainFunctions = mainFunction;
```

In the code snippet, `arduino_main_initialization.c` contains C code. For example:

```
/* Initialize system */
init();
```

Specify main Function Run-Time Arguments

This code snippet from “Set Up PIL Connectivity by Using target Package” shows how you can create and use a `target.MainFunction` object to specify `main` function arguments that are required for an API implementation.

```
comms = target.create('CommunicationInterface');
comms.Name = 'Linux TCP Interface';
comms.Channel = 'TCPChannel';
comms.APIImplementations = target.create('APIImplementation', ...
                                         'Name', 'x86 rtiostream Implementation');
comms.APIImplementations.API = target.create('API', 'Name', 'rtiostream');
comms.APIImplementations.BuildDependencies = target.create('BuildDependencies');
comms.APIImplementations.BuildDependencies.SourceFiles = ...
                                                        {fullfile('$(MATLABROOT)', ...
                                                        'toolbox', ...
                                                        'coder', ...
                                                        'rtiostream', ...
                                                        'src', ...
                                                        'rtiostreamtcpip', ...
                                                        'rtiostream_tcpip.c')};
comms.APIImplementations.MainFunction = target.create('MainFunction', ...
                                                      'Name', 'TCP RtIOStream Main');
comms.APIImplementations.MainFunction.Arguments = {'-blocking', '1', '-port', '0'};
hostTarget.CommunicationInterfaces = comms;
```

See Also

`target.APIImplementation` | `target.Board` | `target.create`

Topics

“Set Up PIL Connectivity by Using target Package”

Introduced in R2020b

target.MATLABDependencies class

Package: `target`

Describe MATLAB class and function dependencies

Description

Use the `target.MATLABDependencies` class to describe MATLAB class and function dependencies.

To create a `target.MATLABDependencies` object, use the `target.create` function.

Properties

`Classes` — MATLAB classes
string array

MATLAB classes that make up the implementation dependencies.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`Functions` — MATLAB functions
string array

MATLAB functions that make up the implementation dependencies.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

Examples

MATLAB Dependencies of `target.DebugIOTool` Implementation

For an example that describes the MATLAB dependencies of an implementation of `target.DebugIOTool`, see “Use Debugger for PIL Target Connectivity”.

See Also
`target.API` | `target.APIImplementation` | `target.BuildDependencies` | `target.create`

Topics
“Use Debugger for PIL Target Connectivity”

Introduced in R2021a

target.Object class

Package: `target`

Base class for target types

Description

`target.Object` is an abstract base class that enables target types to inherit common functionality.

Class Attributes

`Abstract`	`true`
`HandleCompatible`	`true`

For information on class attributes, see "Class Attributes".

Properties

`IsValid` — Data validity

true | false

Attributes:

`GetAccess`	`public`
`SetAccess`	`private`

Data Types: `logical`

Methods

Public Methods

validate   Validate data integrity of target feature object

Examples

Validate Hardware Device Data

For an example that uses this class, see "Validate Hardware Device Data".

See Also

Topics

"Register New Hardware Devices"

Introduced in R2019b

validate

Class: `target.Object`
Package: `target`

Validate data integrity of target feature object

Syntax

```
myTargetFeature.validate()
```

Description

`myTargetFeature.validate()` runs a procedure to validate the data integrity of the object *`myTargetFeature`*. If the validation procedure fails, the method produces an error.

Examples

Validate Hardware Device Data

For an example that uses this method, see “Validate Hardware Device Data”.

See Also

Topics
“Register New Hardware Devices”

Introduced in R2019b

target.PILProtocol class

Package: `target`

Describe PIL protocol implementation for target hardware

Description

Use the `target.PILProtocol` class, which inherits functionality from `target.CommunicationProtocolStack`, to describe the processor-in-the-loop (PIL) communication protocol implementation for your target hardware. For example, use this class to provide buffering information for data transfer and timeout information for I/O with the associated `target.Board` object.

To create a `target.PILProtocol` object, use the `target.create` function.

Properties

`Name` — PIL protocol object name
character vector | string

Name of the PIL protocol object.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`SendBufferSize` — Send buffer size
scalar integer

Size of send buffer for caching communication data.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`ReceiveBufferSize` — Receive buffer size
scalar integer

Size of receive buffer for caching communication data.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`SendTimeout` — Send timeout
scalar integer

Timeout that is applied to a data send command, specified in seconds.

Attributes:

GetAccess	public
SetAccess	public

ReceiveTimeout — Receive timeout
scalar integer

Timeout that is applied to a data receive command, specified in seconds.

Attributes:

GetAccess	public
SetAccess	public

OpenTimeout — Open timeout
scalar integer

Timeout that is applied when opening PIL communications, specified in seconds.

Attributes:

GetAccess	public
SetAccess	public

ByteInputOutputOnly — Bytes only
scalar integer

Specify whether PIL communication sends and receives only bytes.

Attributes:

GetAccess	public
SetAccess	public

Examples

Specify PIL Protocol Information

Specify PIL protocol information. This code snippet from "Set Up PIL Connectivity by Using target Package" shows how to specify the information.

```
pilProtocol = target.create('PILProtocol');
pilProtocol.Name = 'Linux PIL Protocol';
pilProtocol.SendBufferSize = 50000;
pilProtocol.ReceiveBufferSize = 50000;
hostTarget.CommunicationProtocolStacks = pilProtocol;
```

See Also
target.CommunicationProtocolStack | target.create

Topics
"Set Up PIL Connectivity by Using target Package"

Introduced in R2020b

target.Port class

Package: `target`

Describe connection via target hardware port

Description

Use the `target.Port` class, which inherits from `target.ConnectionProperties`, to describe a connection via a port of the target hardware. For example, the serial `COM` port.

Properties

`AddOns` — Custom property add-on objects

`target.AddOns` object array

To provide additional custom properties, associate one or more `target.AddOn` objects with the `target.Port` object.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`PortNumber` — Port number

string

Number of the port associated with the connection.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

Methods

Public Methods

get	Get name of custom add-on property
set	Set property value

See Also

`target.ConnectionProperties` | `target.create`

Introduced in R2021a

target.PortConnection class

Package: `target`

Describe target connection port

Description

Use the `target.PortConnection` class, which inherits functionality from `target.ConnectionProperties`, to describe a target connection port. For example, a serial `COM` port.

To create a `target.PortConnection` object, use the `target.create` function.

Properties

`Port` — Port
character vector | string

Port number.

Attributes:

GetAccess	public
SetAccess	public

See Also

`target.ConnectionProperties` | `target.create`

Introduced in R2020b

target.Processor class

Package: `target`

Provide target processor information

Description

Use the `target.Processor` class to provide information about your target processor. For example, name, manufacturer, and language implementation.

To create a `target.Processor` object, use the `target.create` function.

Properties

Id — Object identifier
character vector | string

The object identifier is the hyphenated combination of the `Manufacturer` and `Name` property values. If the `Manufacturer` property is empty, the object identifier is the `Name` property value.

Attributes:

GetAccess	public
SetAccess	private

`LanguageImplementations` — Language implementation
object

Associated `target.LanguageImplementation` object.

Attributes:

GetAccess	public
SetAccess	public

Name — Processor name
character vector | string

Name of the target processor.

Example: `'Cortex-A53'`

Attributes:

GetAccess	public
SetAccess	public

`Manufacturer` — Processor manufacturer
character vector | string

Optional description of the target processor manufacturer.

Example: `'ARM Compatible'`

Attributes:

GetAccess	public
SetAccess	public

Timers — Timers
target.Counter object array

Provide timer information.

Attributes:

GetAccess	public
SetAccess	public

Overheads — Profiling instrumentation overheads
vector

Specify instrumentation overhead values for removal from execution-time measurements.

Attributes:

GetAccess	public
SetAccess	public

Examples

Create New Hardware Implementation

For examples that use this class, see:

- "Specify Hardware Implementation for New Device"
- "Create Hardware Implementation by Modifying Existing Implementation"
- "Create Hardware Implementation by Reusing Existing Implementation"

Create Timer Object

This example shows how you can create a timer object for your development computer.

Create the function signature for a timer. In this example, the function returns a `uint64` data type and the function name `timestamp_x86`.

```
timerSignature = target.create('Function');
timerSignature.Name = 'timestamp_x86';
timerSignature.ReturnType = 'uint64';
```

Capture the function in an API object.

```
timerApi = target.create('API');
timerApi.Functions = timerSignature;
timerApi.Language = target.Language.C;
timerApi.Name = 'Linux Timer API';
```

Capture the dependencies of the function, that is, the source and header files that are required to run the function.

```
timerDependencies = target.create('BuildDependencies');
timerDependencies.IncludeFiles = {'host_timer_x86.h'};
timerDependencies.IncludePaths = ...
               {'$(MATLAB_ROOT)/toolbox/coder/profile/src'};
timerDependencies.SourceFiles = {'host_timer_x86.c'};
```

Create an object that combines the API and dependencies.

```
timerImplementation = target.create('APIImplementation');
timerImplementation.API = timerApi;
timerImplementation.BuildDependencies = timerDependencies;
timerImplementation.Name = 'Linux Timer Implementation';
```

Create the timer object and associate it with the timer information.

```
timer = target.create('Timer');
timer.APIImplementation = timerImplementation;
timer.Name = 'Linux Timer';
```

Note Using name-value arguments, you can create the timer object with this command.

```
timer = target.create('Timer', 'Name', 'Linux Timer', ...
          'FunctionName', 'timestamp_x86', ...
          'FunctionReturnType', 'uint64', ...
          'FunctionLanguage', target.Language.C, ...
          'SourceFiles', {'host_timer_x86.c'}, ...
          'IncludeFiles', {'host_timer_x86.h'}, ...
          'IncludePaths', {'$(MATLAB_ROOT)/toolbox/coder/profile/src'})
```

Assign the timer and add-ons to the processor object.

```
processor = target.get('Processor', 'Intel-x86-64 (Linux 64)');
processor.Timers = timer;
```

See Also

`target.LanguageImplementation` | `target.create`

Topics

“Register New Hardware Devices”
“Remove Instrumentation Overheads by Using target Package”

Introduced in R2019a

target.ProfilingFreezingOverhead class

Package: `target`

Capture freezing and unfreezing instrumentation overhead

Description

Use a `target.ProfilingFreezingOverhead` object to capture the instrumentation overhead for freezing and unfreezing a timer.

To create a `target.ProfilingFreezingOverhead` object, use the `target.create` function.

Properties

`Value` — Instrumentation overhead
scalar

Specify instrumentation overhead for freezing and unfreezing a timer.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

Data Types: `int`

`Counter` — Timer
target.Timer object

Object that provides the timer details for your processor.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

Data Types: `int`

`MinimumBenchmarkIterations` — Instrumentation overhead
100 (default) | scalar

Specify the minimum number of iterations that benchmark program performs to compute instrumentation overhead values.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

Data Types: `int`

Examples

Specify Function Instrumentation Overhead

Manually specify the function instrumentation overhead for a timer.

Retrieve the `target.Processor` and `target.Timer` objects.

```
processor = target.get('Processor', 'myProcessorObjectId');
timer = target.get('Timer', 'myTimerObjectId');
```

Create a `target.ProfilingFreezingOverhead` object.

```
freezingOverhead = target.create('ProfilingFreezingOverhead', ...
                                 'Name', 'Timer Freezing Overhead');
freezingOverhead.Value = 30;
freezingOverhead.Counter = timer;
```

See Also

`target.ProfilingFunctionOverhead` | `target.ProfilingTaskOverhead`

Topics

"Remove Instrumentation Overheads by Using target Package"

Introduced in R2021a

target.ProfilingFunctionOverhead class

Package: `target`

Capture function instrumentation overhead

Description

Use a `target.ProfilingFunctionOverhead` object to capture the instrumentation overhead for profiling a function.

To create a `target.ProfilingFunctionOverhead` object, use the `target.create` function.

Properties

`Value` — Instrumentation overhead
scalar

Specify instrumentation overhead for profiling a function.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `int`

`Counter` — Timer
target.Timer object

Object that provides the timer details for your processor.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `int`

`MinimumBenchmarkIterations` — Instrumentation overhead
100 (default) | scalar

Specify the minimum number of iterations that benchmark program performs to compute instrumentation overhead values.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `int`

Examples

Specify Function Instrumentation Overhead

Manually specify the function instrumentation overhead for a timer.

Retrieve the `target.Processor` and `target.Timer` objects.

```
processor = target.get('Processor', 'myProcessorObjectId');
timer = target.get('Timer', 'myTimerObjectId');
```

Create a `target.ProfilingFunctionOverhead` object.

```
functionOverhead = target.create('ProfilingFunctionOverhead', ...
                              'Name', 'Timer Function Overhead');
functionOverhead.Value = 20;
functionOverhead.Counter = timer;
```

See Also

`target.ProfilingFreezingOverhead` | `target.ProfilingTaskOverhead`

Topics

"Remove Instrumentation Overheads by Using target Package"

Introduced in R2021a

target.ProfilingTaskOverhead class

Package: `target`

Capture task instrumentation overhead

Description

Use a `target.ProfilingTaskOverhead` object to capture the instrumentation overhead for profiling a task.

To create a `target.ProfilingTaskOverhead` object, use the `target.create` function.

Properties

`Value` — Instrumentation overhead
scalar

Specify instrumentation overhead for profiling a task.

Attributes:

```
GetAccess                                    public
SetAccess                                    public
```

Data Types: `int`

`Counter` — Timer
target.Timer object

Object that provides the timer details for your processor.

Attributes:

```
GetAccess                                    public
SetAccess                                    public
```

Data Types: `int`

`MinimumBenchmarkIterations` — Instrumentation overhead
100 (default) | scalar

Specify the minimum number of iterations that benchmark program performs to compute instrumentation overhead values.

Attributes:

```
GetAccess                                    public
SetAccess                                    public
```

Data Types: `int`

Examples

Specify Task Instrumentation Overhead

Manually specify the task instrumentation overhead for a timer.

Retrieve the `target.Processor` and `target.Timer` objects.

```
processor = target.get('Processor', 'myProcessorObjectId');
timer = target.get('Timer', 'myTimerObjectId');
```

Create a `target.ProfilingTaskOverhead` object.

```
taskOverhead = target.create('ProfilingTaskOverhead', ...
                             'Name', 'Timer Task Overhead');
taskOverhead.Value = 10;
taskOverhead.Counter = timer;
```

See Also

`target.ProfilingFreezingOverhead` | `target.ProfilingFunctionOverhead`

Topics

"Remove Instrumentation Overheads by Using target Package"

Introduced in R2021a

target.remove

Package: `target`

Remove target object from internal database

Syntax

```
target.remove(targetObject)
target.remove(targetType, targetObjectId)
target.remove(targetObject, Name,Value)
```

Description

`target.remove(targetObject)` removes the target object from an internal database.

`target.remove(targetType, targetObjectId)` removes the target object specified by class and identifier.

`target.remove(targetObject, Name,Value)` uses name-value arguments to remove associated objects and suppress command-line output.

Examples

Remove Target Object From Internal Database

You can specify and add a hardware device implementation to an internal database.

```
armv8 = target.create('LanguageImplementation', ...
              'Name', 'Armv8-A LP64', ...
              'Copy', 'ARM Compatible-ARM Cortex');

a53 = target.create('Processor', ...
                    'Name', 'Cortex-A53', ...
                    'Manufacturer', 'ARM Compatible');

a53.LanguageImplementations = armv8;

target.add(a53)
```

When a target object is no longer required, you can use the function to remove the object from the internal database.

To remove only the `target.Processor` object, run:

```
target.remove(a53)
```

Or:

```
target.remove('Processor', 'ARM Compatible-Cortex-A53');
```

To remove the `target.Procesor` object and its associated `target.LanguageImplementation` object and suppress the command-line output, run:

```
target.remove(a53, ...
              'IncludeAssociations', true, ...
               'SuppressOutput',true);
```

Input Arguments

`targetObject` — Target object
object

Specify the target object that you want to remove.

`targetType` — Target type
character vector | string

Specify the class of the target object that you want to remove. For example:

- If the class is `target.Processor`, specify `'Processor'`.
- If the class is `target.LanguageImplementation`, specify `'LanguageImplementation'`.

Example: `'Processor'`

`targetObjectId` — Target object identifier
character vector | string

Specify the unique identifier of the object that you want to remove, that is, the `Id` property value of the object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `target.remove(myTargetObject, 'IncludeAssociations', true);`

`IncludeAssociations` — Remove associated objects
false (default) | true

Remove associated objects from internal database:

- `true` –– Function removes `targetObject` and associated target objects from the internal database. If an associated object is referenced by another target object, the function does not remove the associated object.
- `false` –– Function removes only `targetObject` from the internal database.

Example: `target.remove(myTargetObject, 'IncludeAssociations', true);`

Data Types: `logical`

`SuppressOutput` — Control command-line output
false (default) | true

Control command-line output of function:

- `true` –– Suppress command-line output from the function.
- `false` –– Provide information about the objects that the function removes from the internal database.

Example: `target.remove(`*`myTargetObject`*`, 'SuppressOutput', true);`

Data Types: `logical`

See Also

`target.add` | `target.create` | `target.get`

Topics

"Register New Hardware Devices"

Introduced in R2019a

target.RS232Channel class

Package: `target`

Describe serial communication channel

Description

Use the `target.RS232Channel` class, which inherits functionality from `target.CommunicationChannel`, to describe properties of the serial communication channel.

To create a `target.RS232Channel` object, use the `target.create` function.

Properties

`BaudRate` — Baud value
scalar integer

Baud value of the serial connection.

Attributes:

GetAccess	public
SetAccess	public

See Also
`target.CommunicationChannel` | `target.create`

Introduced in R2020b

target.SystemCommandExecutionTool class

Package: `target`

Capture system command information to run target application from MATLAB computer

Description

Use the `target.SystemCommandExecutionTool` to capture system command information that is required to run the target application from your development computer.

Properties

`Name` — Execution tool name
character vector | string

Name of the execution tool.

Attributes:

GetAccess	public
SetAccess	public

`StartCommand` — Command list to run application
`target.Command` object

A `target.Command` object that provides a system command for running the application. The command in the list starts the application process.

This property must not be empty.

Attributes:

GetAccess	public
SetAccess	public

`StopCommand` — Command list to stop application
`target.Command` object

A `target.Command` object that provides a system command to stop the application execution.

Attributes:

GetAccess	public
SetAccess	public

`Id` — Object identifier
character vector | string

Value of the `Name` property.

Attributes:

`GetAccess`	`public`
`SetAccess`	`private`

See Also

`target.ApplicationExecutionTool` | `target.Command` | `target.HostProcessExecutionTool` | `target.create`

Topics

"Set Up PIL Connectivity by Using target Package"

Introduced in R2020b

target.TargetConnection class

Package: target

Provide details about connecting MATLAB computer to target hardware

Description

Use the target.TargetConnection class, which inherits functionality from target.Connection, to provide details about connecting your MATLAB computer to target hardware. For example, the communication channel and connection properties that are required for communication with your target hardware.

To create a target.TargetConnection object, use the target.create function. Create the object and then use separate steps to specify properties. Or, using name-value arguments, create the object and specify properties in a single step.

Properties

Name — Connection object
character vector | string

Name of connection object.

Attributes:

GetAccess	public
SetAccess	public

CommunicationChannel — Communication channel type
target.CommunicationChannel object

Associate a target.CommunicationChannel object with your connection, which describes the type of channel that is used. For example, to specify serial or TCP channel properties, use target.RS232Channel or target.TCPChannel respectively.

If you use name-value arguments to create a target.TargetConnection object, for the CommunicationChannel property, specify these arguments.

Name	Description
'CommunicationType'	Required. Type of predefined communication channel. Specifies one of these values:
'IPAddress'	Optional. If predefined communication channel is 'TCPChannel' or 'UDPChannel', specifies IPAddress property of target.TCPChannel or target.UDPChannel object.
'Port'	Optional. If predefined communication channel is 'TCPChannel' or 'UDPChannel', specifies Port property of target.TCPChannel or target.UDPChannel object.

Name	Description
'BaudRate'	Optional. If predefined communication channel is 'RS232Channel', specifies BaudRate property of target.RS232Channel object.
'Parity'	Optional. If predefined communication channel is 'RS232Channel', specifies Parity property of target.RS232Channel object.

Attributes:

GetAccess	public
SetAccess	public

Target — Connected target

target.Board object

Associate a target.Board object with your connection, which describes the target hardware that is connected to the MATLAB computer.

Attributes:

GetAccess	public
SetAccess	public

ConnectionProperties — MATLAB computer connection properties

target.ConnectionProperties object

Associate a target.ConnectionProperties object with your connection that describes the MATLAB computer connection properties that are used to connect to the target hardware. For example, to specify the serial port, use a target.Port object.

If you use name-value arguments to create a target.TargetConnection object and the predefined communication channel type is 'RS232Channel', specifying the argument 'Port' sets the ConnectionProperties property to target.Port.

Attributes:

GetAccess	public
SetAccess	public

TargetConnectionProperties — Target connection properties

target.ConnectionProperties object

Associate a target.ConnectionProperties object with your connection that describes the target hardware connection properties that are used to connect to the MATLAB computer. For example, to specify the serial port, use a target.Port object.

If you use name-value arguments to create a target.TargetConnection object and the predefined communication channel type is 'RS232Channel', specifying the argument 'Port' sets the TargetConnectionProperties property to target.Port.

Attributes:

GetAccess	public
SetAccess	public

Examples

Specify Connection Between Development Computer and Target Hardware

This code from “Set Up PIL Connectivity by Using target Package” shows how to specify the connection between your development computer and target hardware. In the example, the target application runs on your development computer as a separate process and uses a TCP communication channel through `localhost`.

```
connection = target.create('TargetConnection');
connection.Name = 'Host Process Connection';
connection.Target = hostTarget;
connection.CommunicationChannel = target.create('TCPChannel');
connection.CommunicationChannel.Name = ...
                   'External Process TCPCommunicationChannel';
connection.CommunicationChannel.IPAddress = 'localhost';
connection.CommunicationChannel.Port = '0';
```

Note Using name-value arguments, you can create the connection object with this command:

```
timer = target.create('TargetConnection', ...
                      'Name', 'Host Process Connection', ...
                      'CommunicationType', 'TCPChannel', ...
                      'IPAddress', 'localhost', ...
                      'Port', '0')
```

See Also

`target.Board` | `target.CommunicationChannel` | `target.ConnectionProperties` | `target.create`

Introduced in R2020b

target.TCPChannel class

Package: `target`

Describe TCP communication properties

Description

Use the `target.TCPChannel`, which inherits functionality from `target.CommunicationChannel`, to describe TCP communication properties.

To create a `target.TCPChannel` object, use the `target.create` function.

Properties

`IPAddress` — IP address

character vector | string

IP address of the TCP server.

Attributes:

GetAccess	public
SetAccess	public

`Port` — Port

scalar integer

TCP port.

Attributes:

GetAccess	public
SetAccess	public

Examples

Create Connection by Using TCP Communication Channel

This code from "Set Up PIL Connectivity by Using target Package" shows how to specify the connection between your development computer and target hardware. In the example, the target application runs on your development computer as a separate process and uses a TCP communication channel through `localhost`.

```
connection = target.create('TargetConnection');
connection.Name = 'Host Process Connection';
connection.Target = hostTarget;
connection.CommunicationChannel = target.create('TCPChannel');
connection.CommunicationChannel.Name = ...
                   'External Process TCPCommunicationChannel';
connection.CommunicationChannel.IPAddress = 'localhost';
connection.CommunicationChannel.Port = '0';
```

Note Using name-value arguments, you can create the connection object with this command:

```
timer = target.create('TargetConnection', ...
                      'Name', 'Host Process Connection', ...
                      'CommunicationType', 'TCPChannel', ...
                      'IPAddress', 'localhost', ...
                      'Port', '0')
```

See Also

target.CommunicationChannel | target.TargetConnection | target.create

Topics

“Set Up PIL Connectivity by Using target Package”

Introduced in R2020b

target.Timer class

Package: `target`

Provide timer details for processor

Description

Use the `target.Timer` class to provide timer details for your processor. For example, information about the C or C++ function interface and implementation, frequency, and timer count direction. To provide information about instrumenting C or C++ code for profiling, you can associate the timer details with a `target.Processor` object.

To create a `target.Timer` object, use the `target.create` function. Create the object and then use separate steps to specify properties. Or, using name-value arguments, create the object and specify properties in a single step.

Properties

`Name` — Timer name

character vector | string

Name of timer.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`Direction` — Count direction

'Up' | 'Down'

Direction of timer count.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`APIImplementation` — API implementation

`target.APIImplementation` object

Information about the API implementation, which is used to determine the current time.

If you use name-value arguments to create a `target.Timer` object, for the `APIImplementation` property, specify these arguments.

Name	Description
`'FunctionName'`	Required. `Name` property of `target.Function` object.
`'FunctionReturnType'`	Required. `ReturnType` property of `target.Function` object.

Name	Description
'IncludeFiles'	Required. IncludeFiles property of target.BuildDependencies object
'FunctionLanguage'	Optional. Language property of target.API object.
'SourceFiles'	Optional. SourceFiles property of target.BuildDependencies object.
'IncludePaths'	Optional. IncludePaths property of target.BuildDependencies object

Attributes:

GetAccess	public
SetAccess	public

Frequency — Frequency
scalar

Frequency of the unit returned by the timer function. This value can be used to convert timer function output to seconds. The helper class `target.unit.Frequency` contains some common frequency units.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `uint64`

Examples

Create Timer Object

Create a timer object for your development computer.

Create the function signature for a timer. In this example, the function returns a `uint64` data type and the function name `timestamp_x86`.

```
timerSignature = target.create('Function');
timerSignature.Name = 'timestamp_x86';
timerSignature.ReturnType = 'uint64';
```

Capture the function in an API object.

```
timerApi = target.create('API');
timerApi.Functions = timerSignature;
timerApi.Language = target.Language.C;
timerApi.Name = 'Linux Timer API';
```

Capture the dependencies of the function, that is, the source and header files that are required to run the function.

```
timerDependencies = target.create('BuildDependencies');
timerDependencies.IncludeFiles = {'host_timer_x86.h'};
timerDependencies.IncludePaths = ...
               {'$(MATLAB_ROOT)/toolbox/coder/profile/src'};
timerDependencies.SourceFiles = {'host_timer_x86.c'};
```

Create an object that combines the API and dependencies.

```
timerImplementation = target.create('APIImplementation');
timerImplementation.API = timerApi;
timerImplementation.BuildDependencies = timerDependencies;
timerImplementation.Name = 'Linux Timer Implementation';
```

Create the timer object and associate it with the timer information.

```
timer = target.create('Timer');
timer.APIImplementation = timerImplementation;
timer.Name = 'Linux Timer';
```

Note Using name-value arguments, you can create the timer object with this command.

```
timer = target.create('Timer', 'Name', 'Linux Timer', ...
          'FunctionName', 'timestamp_x86', ...
          'FunctionReturnType', 'uint64', ...
          'FunctionLanguage', target.Language.C, ...
          'SourceFiles', {'host_timer_x86.c'}, ...
          'IncludeFiles', {'host_timer_x86.h'}, ...
          'IncludePaths', {'$(MATLAB_ROOT)/toolbox/coder/profile/src'})
```

Assign the timer and add-ons to the processor object.

```
processor = target.get('Processor', 'Intel-x86-64 (Linux 64)');
processor.Timers = timer;
```

Create Timer Object by Modifying Existing Object

You can create a new timer object by copying an existing timer object and modifying specific property values of the copy.

```
newTimer = target.create('Timer', ...
                         'Copy', 'Linux Timer', ...
                         'Name', 'NewTimerName', ...
                         'FunctionName', 'newFunctioName');
```

See Also

`target.APIImplementation` | `target.Processor` | `target.create`

Introduced in R2020b

target.Tools class

Package: `target`

Describe properties of tools for target hardware

Description

Use the `target.Tools` class to capture properties of tools that enable your development computer to interact with the target hardware.

Properties

`DebugTools` — Supported debuggers

target.ApplicationExecutionTool object vector

Descriptions of debugging tools that are supported for the target hardware.

Attributes:

GetAccess	public
SetAccess	public

`ExecutionTools` — Supported execution tools

target.ApplicationExecutionTool object vector

Descriptions of supported tools for executing applications on the target hardware.

Attributes:

GetAccess	public
SetAccess	public

Examples

PIL Target Connectivity with Debugger

For an example that uses the `target.Tools` class, see “Use Debugger for PIL Target Connectivity”.

See Also

`target.Board` | `target.create`

Topics

“Use Debugger for PIL Target Connectivity”

Introduced in R2020b

target.UDPChannel class

Package: `target`

Describe UDP communication

Description

Use the `target.UDPChannel` class, which inherits functionality from `target.CommunicationChannel`, to describe User Datagram Protocol (UDP) communication.

To create a `target.UDPChannel` object, use the `target.create` function.

Properties

`IPAddress` — IP address
character vector | string

IP address of the UDP server.

Attributes:

GetAccess	public
SetAccess	public

`Port` — Port
scalar integer

UDP port.

Attributes:

GetAccess	public
SetAccess	public

See Also

`target.CommunicationChannel` | `target.TargetConnection` | `target.create`

Introduced in R2020b

target.upgrade

Package: `target`

Upgrade existing definitions of hardware devices

Syntax

```
target.upgrade(upgraderForRegistrationMechanism, pathToRegistrationFile)
target.upgrade(___, Name,Value)
```

Description

`target.upgrade(upgraderForRegistrationMechanism, pathToRegistrationFile)` uses an upgrade procedure to create objects from data definitions in current artifacts. The function creates `registerUpgradedTargets.m` in the current working folder.

To register the upgraded data definitions with MATLAB, run `registerUpgradedTargets()`.

For persistence across MATLAB sessions, run `registerUpgradedTargets('UserInstall', true)`.

`target.upgrade(___, Name,Value)` specifies additional options using one or more name-value pair arguments.

Examples

Upgrade Hardware Device Data Definitions

For a workflow example that uses the function, see "Upgrade Data Definitions for Hardware Devices".

Input Arguments

`upgraderForRegistrationMechanism` — Upgrade procedure
'rtwTargetInfo' | 'sl_customization'

Select upgrade procedure for current registration mechanism.

`pathToRegistrationFile` — Full file name
character vector | string

Specify file that contains current registration mechanism.

Example: `target.upgrade('rtwTargetInfo', 'myPath/mySubfolder/rtwTargetInfo.m')`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `target.upgrade('rtwTargetInfo', 'myPath/mySubfolder/rtwTargetInfo.m','ExportFileName','myNewFile')`

`ExportToMATLABFunction` — Export to MATLAB
true (default) | false

- `true` –– Generate a MATLAB function that registers the upgraded definitions using `target.add`.
- `false` –– Do not generate a function.

`ExportFileName` — Generated function file name
'registerUpgradedTargets.m' (default) | string

If `ExportToMATLABFunction` is `true`, the argument specifies the file name of the generated MATLAB function . Otherwise, the argument is ignored.

`Overwrite` — Overwrite existing file
false (default) | true

- `true` –– If the file specified by `ExportFileName` exists, overwrite the file.
- `false` –– If the file specified by `ExportFileName` exists, the function produces an error.

If `ExportToMATLABFunction` is false, the argument is ignored.

See Also

`target.add` | `target.create`

Topics
"Register New Hardware Devices"

Introduced in R2019b

target.XCP class

Package: `target`

Describe XCP protocol stack for target hardware

Description

Use the `target.XCP` class to describe the XCP protocol stack for the target hardware.

To create a `target.XCP` object, use the `target.create` function.

Properties

`XCPTransport` — XCP transport protocol

target.XCPTransport object

Specify the XCP transport protocol layer through a `target.XCPTCPIPTransport` or a `target.XCPSerialTransport` object.

Attributes:

GetAccess	public
SetAccess	public

`XCPPlatformAbstraction` — XCP platform abstraction

target.XCPPlatformAbstraction object

Optional property to specify the XCP platform abstraction layer. If you do not specify a value, the software uses the default platform abstraction layer.

Attributes:

GetAccess	public
SetAccess	public

Examples

XCP Protocol Stack for Target Hardware

This code snippet from “Customise Connectivity for XCP External Mode Simulations” shows how you can provide a description of the XCP protocol stack for your target hardware.

```
xcpTCPIPConfiguration = target.create('XCP', ...
                'Name', 'XCP TCP/IP Configuration', ...
                'XCPTransport', xcpTCPIPTransport, ...
                'XCPPlatformAbstraction', xcpPlatformAbstraction);
```

See Also

`target.XCPPlatformAbstraction` | `target.XCPTransport` | `target.create`

Topics

"Customise Connectivity for XCP External Mode Simulations"

Introduced in R2021a

target.XCPExternalModeConnectivity class

Package: `target`

Represent connectivity options in external mode protocol stack

Description

Use the `target.XCPExternalModeConnectivity` class, which is derived from `target.ExternalModeConnectivity`, to represent XCP connectivity options in the external mode protocol stack.

To create a `target.XCPExternalModeConnectivity` object, use the `target.create` function.

Properties

XCP — XCP configuration
target.XCP object

Specify XCP protocol stack for target hardware.

Examples

Specify External Mode Protocol Stack for Target Hardware

This code snippet from "Customise Connectivity for XCP External Mode Simulations" shows how to specify the external mode protocol stack for your target hardware.

```
extModeTCPConnectivity = ...
             target.create('XCPExternalModeConnectivity', ...
             'Name', 'External Mode TCP Connectivity', ...
             'XCP', xcpTCPIPConfiguration);

externalMode = target.create('ExternalMode', ...
                'Name', 'External Mode', ...
                'Connectivities', extModeTCPConnectivity);

board.CommunicationProtocolStacks = externalMode;
```

See Also

`target.ExternalMode` | `target.create`

Topics
"Customise Connectivity for XCP External Mode Simulations"

Introduced in R2021a

target.XCPPlatformAbstraction class

Package: `target`

Specify XCP platform abstraction layer for target hardware

Description

Use the `target.XCPPlatformAbstraction` class to specify the implementation of the "XCP Platform Abstraction Layer" for your target hardware. The layer provides:

- The implementation of a static memory allocator –– see "Memory Allocator".
- Other target hardware-specific functionality –– see "Other Platform Abstraction Layer Functionality".

To create a `target.XCPPlatformAbstraction` object, use the `target.create` function.

Properties

`BuildDependencies` — Platform abstraction layer build dependencies
target.BuildDependencies object

Specify preprocessor directives, source files, and header files that are required for the implementation of the XCP platform abstraction layer.

Attributes:

GetAccess	public
SetAccess	public

Examples

Specify Communication Protocol Stack for Target Hardware

This code snippet from "Customise Connectivity for XCP External Mode Simulations" shows how to specify and use a custom implementation of the XCP platform abstraction layer.

```
xcpPlatformAbstraction = target.create('XCPPlatformAbstraction', ...
                'Name', 'XCP Platform Abstraction');

xcpPlatformAbstraction.BuildDependencies.Defines = {'XCP_CUSTOM_PLATFORM'};
customPlatformAbstractionPath = 'pathToImplementationFolder';
xcpPlatformAbstraction.BuildDependencies.SourceFiles = ...
                {fullfile(customPlatformAbstractionPath, 'myXCPPlatform.c')};
xcpPlatformAbstraction.BuildDependencies.IncludePaths = ...
                {customPlatformAbstractionPath};

xcpTCPIPTransport = target.create('XCPTCPIPTransport', ...
                'Name', 'XCP TCPIP Transport');

xcpTCPIPConfiguration = target.create('XCP', ...
                'Name', 'XCP TCP/IP Configuration', ...
```

```
    'XCPTransport', xcpTCPIPTransport, ...
    'XCPPlatformAbstraction', xcpPlatformAbstraction);
```

See Also

`target.XCP` | `target.create`

Topics

"Customise Connectivity for XCP External Mode Simulations"

Introduced in R2021a

target.XCPTCPIPTransport class

Package: `target`

Represent XCP TCP/IP transport protocol layer

Description

Use the `target.XCPTCPIPTransport` class, which inherits functionality from `target.XCPTransport`, to represent the XCP TCP/IP transport protocol layer for your target hardware.

Examples

XCP Protocol Stack for Target Hardware

This code snippet shows how you can use the `target.XCPTCPIPTransport` class to represent the XCP TCP/IP transport protocol layer for your target hardware.

```
xcpTCPIPTransport = ...
                    target.create('XCPTCPIPTransport', ...
                    'Name', 'XCP TCPIP Transport');
```

See Also

`target.XCP` | `target.XCPTransport` | `target.create`

Topics

"Customise Connectivity for XCP External Mode Simulations"

Introduced in R2021a

target.XCPTransport class

Package: `target`

Base class for XCP transport protocol layer

Description

The `target.XCPTransport` class is a base class for representing the XCP transport protocol layer on the target hardware. The `target.XCPTCPIPTransport` and `target.XCPSerialTransport` classes are derived from this class.

Class Attributes

`Abstract`	`true`
`HandleCompatible`	`true`

For information on class attributes, see “Class Attributes”.

See Also

`target.XCPSerialTransport` | `target.XCPTCPIPTransport` | `target.create`

Topics

“Customise Connectivity for XCP External Mode Simulations”

Introduced in R2021a

target.XCPSerialTransport class

Package: `target`

Represent XCP serial transport protocol layer

Description

Use the `target.XCPSerialTransport` class, which inherits functionality from `target.XCPTransport`, to represent the XCP serial transport protocol layer for your target hardware.

Examples

XCP Serial Transport

This code snippet shows how you can use the `target.XCPSerialTransport` class to represent the XCP serial transport protocol layer for your target hardware.

```
xcpSerialTransport = ...
                     target.create('XCPSerialTransport', ...
                     'Name', 'XCP Serial Transport');
```

See Also

`target.XCP` | `target.XCPTransport` | `target.create`

Topics

"Customise Connectivity for XCP External Mode Simulations"

Introduced in R2021a

updateFilePathsAndExtensions

Update files in build information with missing paths and file extensions

Syntax

`updateFilePathsAndExtensions(buildinfo,extensions)`

Description

`updateFilePathsAndExtensions(buildinfo,extensions)` specifies the file name extensions (file types) to include in search and update processing.

Using paths from the build information, the `updateFilePathsAndExtensions` function checks whether file references in the build information require an updated path or file extension. Use this function to:

- Maintain build information for a toolchain that requires the use of file extensions.
- Update multiple customized instances of build information for a given .

If you use `updateFilePathsAndExtensions`, you call it after you add files to the build information. This approach minimizes the potential performance impact of the required disk I/O.

Examples

Update File Paths and Extensions in Build Information

In your working folder, create the folder path `etcproj/etc` , add files `etc.c`, `test1.c`, and `test2.c` to the folder `etc`. For this example, the working folder is `w:\work\BuildInfo`. From the working folder, update build information `myBuildInfo` with missing paths or file extensions.

```
myBuildInfo = RTW.BuildInfo;
addSourcePaths(myBuildInfo,fullfile(pwd, ...
   'etcproj','/etc'),'test');
addSourceFiles(myBuildInfo,{'etc' 'test1' ...
   'test2'},'','test');
before = getSourceFiles(myBuildInfo,true,true);
```

```
>> before

before =

    '\etc'    '\test1'    '\test2'
```

```
updateFilePathsAndExtensions(myBuildInfo);
after = getSourceFiles(myBuildInfo,true,true);
```

```
>> after{:}
```

```
ans =

    'w:\work\BuildInfo\etcproj\etc\etc.c'


ans =

    'w:\work\BuildInfo\etcproj\etc\test1.c'


ans =

    'w:\work\BuildInfo\etcproj\etc\test2.c'
```

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo`
object

`extensions` — File name extensions to include in search and update processing
`'.c'` (default) | cell array of character vectors | string

The *`extensions`* argument specifies the file name extensions (file types) to include in search and update processing. The function checks files and updates paths and extensions based on the order in which you list the extensions in the cell array. For example, if you specify `{'.c' '.cpp'}` and a folder contains `myfile.c` and `myfile.cpp`, an instance of `myfile` is updated to `myfile.c`.

Example: `'.c' '.cpp'`

See Also

`addIncludeFiles` | `addIncludePaths` | `addSourceFiles` | `addSourcePaths` | `updateFileSeparator`

Topics
"Customize Post-Code-Generation Build Processing"

Introduced in R2006a

updateFileSeparator

Update file separator character for file lists in build information

Syntax

```
updateFileSeparator(buildinfo,separator)
```

Description

`updateFileSeparator(buildinfo,separator)` changes instances of the current file separator (/ or \) in the build information to the specified file separator.

The default value for the file separator matches the value returned by the MATLAB command `filesep`. For template makefile (TMF) approach builds, you can override the default by defining a separator with the `MAKEFILE_FILESEP` macro in the template makefile. If the `GenerateMakefile` parameter is set, the code generator overrides the default separator and updates the build information after evaluating the `PostCodeGenCommand` configuration parameter.

Examples

Update File Separator in Build Information

Update object `myBuildInfo` to apply the Windows file separator.

```
myBuildInfo = RTW.BuildInfo;
updateFileSeparator(myBuildInfo,'\');
```

Input Arguments

`buildinfo` — `RTW.BuildInfo` object
object

`RTW.BuildInfo` object that contains information for compiling and linking generated code.

`separator` — File separator character for path specifications in the build information
`'\'` | `'/'`

The separator argument specifies the file separator `\` (Windows) or `/` (UNIX) to use in file path specifications in the build information.

Example: '\'

See Also

`addIncludeFiles` | `addIncludePaths` | `addSourceFiles` | `addSourcePaths` | `updateFilePathsAndExtensions`

Topics

"Customize Post-Code-Generation Build Processing"

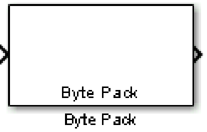
Introduced in R2006a

Embedded Coder Blocks

Byte Pack

Convert input signals to `uint8` vector

Library: Embedded Coder / Embedded Targets / Host Communication



Description

The Byte Pack block receives input signals of one or more data types and converts the data to one `uint8` vector for output. Use block parameters to specify data types of the input signals and the alignment of the data in the vector that the block outputs. Because the UDP protocol transmits data in `uint8` format, you can use this block to reformat data for UDP transmission by connecting the output of this block to the input of a UDP Send block.

Ports

Input

`Port_1` — Signals to convert

signal of type `double` (default) | array of signal data

Array of input signals of one more data types.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `enumerated` | `bus`

Output

`Port_1` — Converted signal data

vector

Converted block input, returned as a vector of `uint8` data.

Data Types: `uint8`

Parameters

`Input port data types (cell array)` — Data types of block input signals

{'double'} (default) | cell array of Simulink data types

Specify the Simulink data types of input signals that the block receives in a cell array. In the cell array, specify data types in the order in which the block input port receives the signal data. For example, if the block receives data in the order `uint32`, `uint32`, `uint16`, `double`, `uint8`, `double`, and `single`, specify this cell array:

```
{'uint32','uint32','uint16','double','uint8','double','single'}
```

The block then provides the number of block inputs.

`Byte alignment` — Byte boundary for data type alignment
1 (default) | 2 | 4 | 8

Specify how to align the data types of input data to form the `uint8` vector output in bytes. Alignment can occur on 1, 2, 4, or 8-byte boundaries. Based on the alignment value, each signal data value begins on multiples of the alignment value. The alignment algorithms for each element in the output vector begin on a byte boundary specified by the alignment value. Byte alignment sets the boundaries relative to the starting point of the vector.

To get the tightest packing without holes between data types in the various combinations of data types, select `1`.

You can have multiple data types of varying lengths. In such cases, a 2-byte alignment can produce 1–byte gaps between `uint8` or `int8` values and another data type. In the pack implementation, the block copies data to the output data buffer 1 byte at a time.

For example, assume that you specify this cell array for **Input port data types (cell array)**:

```
{'uint32','uint32','uint16','double','uint8','double','single'}
```

Assume that you set **Byte alignment** to `2`. Each new value begins 2 bytes from the previous data boundary. When the signals are scalar values, the:

- First signal value in the vector starts at 0 bytes.
- Second signal value starts at 2 bytes.
- Third signal value starts at 4 bytes.
- Fourth signal value starts at 6 bytes.
- Fifth signal value starts at 8 bytes.
- Sixth signal value starts at 10 bytes
- Seventh signal value starts at 12 bytes.

The packing algorithm leaves a 1-byte gap between the `uint8` data value and the double value.

See Also

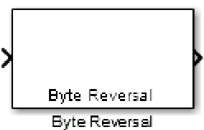
Byte Reversal | Byte Unpack | UDP Send

Introduced in R2011a

Byte Reversal

Reverse order of bytes in input word

Library: Embedded Coder / Embedded Targets / Host Communication

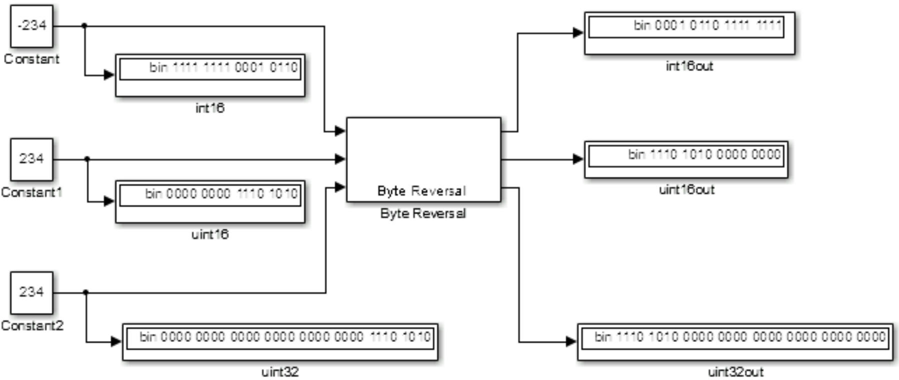


Description

The Byte Reversal block changes the order of the bytes in data that you input to the block. Use this block when a process communicates between target computers that use different endianness, such as between Intel® processors that are little endian and other processors that are big endian. Texas Instruments™ processors are little-endian by default.

To exchange data between processors that have different endianness, place a Byte Reversal block just before the send block and immediately after the receive block.

This model shows byte reversal for three inputs. The input and output ports match for each path.



Ports

Input

`Port_1` — Data to convert

scalar, vector, matrix

Data for which the block changes the byte ordering. The number of input ports used by the block depends on the value that you specify for the **Number of inputs** parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `enumerated` | `bus`

Output

Port_1 — Converted data
scalar, vector, matrix

Data with modified byte ordering. The number of output ports used by the block depends on the value that you specify for the **Number of inputs** parameter. Each input port maps to the matching output port. Data received on input port 1 is sent through output port 1, and so on.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point | enumerated | bus

Parameters

Number of inputs — Number of block inputs
1 (default) | integer

Specify the number of block inputs. The block creates that number of input ports and output ports. Each input port maps to the matching output port. Data received on input Port_1 is sent through output Port_1, and so on.

Reversing the byte ordering does not change the data type. Input and output retain matching data types.

See Also

Byte Pack | Byte Unpack

Introduced in R2011a

Byte Unpack

Convert uint8 vector to input signals
Library: Embedded Coder / Embedded Targets / Host Communication

Description

The Byte Unpack block receives a uint8 vector and converts the vector to output signals of different Simulink data types based on content of the input vector. You can use block parameters to specify dimensions and data types of the output signals and the alignment of the data in the individual vectors that the block outputs. Because the UDP protocol transmits data in uint8 format, you can use this block to reformat data that is received as a UDP packet for use in a model by connecting the input of this block to the output of a UDP Receive block.

Ports

Input

Port_1 — Signal to convert
vector

Input vector of uint8.

Data Types: uint8

Output

Port_1 — Converted signal data
signal of type double (default) | array of signal data

Array of input signals of one more data types.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point | enumerated | bus

Parameters

Output port dimensions (cell array) — Dimensions of block output signals
{[1]} (default) | cell array of data dimension specifications

Specify the dimensions of the output signals that the block outputs in a cell array. Each element in the array specifies the dimension that the MATLAB size function returns for the corresponding signal. Specify dimensions that align with data converted by the corresponding Byte Pack block in the model.

For example, assume the corresponding Byte Pack block specifies these input port data types:

```
{'uint32','uint32','uint16','double','uint8','double','single'}
```

To specify scalar and matrix output, you might set the **Output port dimensions (cell array)** parameter to:

```
{1,1,[2,4],[4,4],[2,2],1,[3,3]}
```

To apply the same dimension across the output signals, you can specify one dimension value.

`Output port data types (cell array)` — Data types of block output signals
{'double'} (default) | cell array of Simulink data types

Specify the Simulink data types of the individual input signals received by the corresponding Byte Pack block in the model in a cell array.

For example, if the corresponding Byte Pack block specifies these input port data types, specify the same cell array for this parameter.

```
{'uint32','uint32','uint16','double','uint8','double','single'}
```

To apply the same data type to each output signal, you need to specify the data type only once.

`Byte Alignment` — Byte boundary for data type alignment
1 (default) | 2 | 4 | 8

Specify how to align the data types of output data to form the `uint8` vector input in bytes. Specify the alignment value that matches the value specified for the corresponding Byte Pack block in the model.

For example, if the corresponding Byte Pack block sets the byte alignment to 2, set this parameter to 2.

See Also

Byte Pack | Byte Reversal | UDP Receive

Introduced in R2011a

CAN Pack

Pack individual signals into CAN message

Library: Simulink Real-Time / CAN / CAN MSG blocks
Vehicle Network Toolbox / CAN Communication
Embedded Coder / Embedded Targets / Host Communication

Description

The CAN Pack block loads signal data into a message at specified intervals during the simulation.

To use this block, you must have a license for Simulink software.

The CAN Pack block supports:

- Simulink Accelerator™ rapid accelerator mode. You can speed up the execution of Simulink models.
- Model referencing. Your model can include other Simulink models as modular components.

For more information, see "Design Your Model for Effective Acceleration".

Tip

- To work with J1939 messages, use the blocks in the J1939 Communication block library instead of this block.

Ports

Input

`input` — CAN message input
single | double | int8 | int16 | int32 | int64 | uint32 | uint64 | boolean

CAN Pack block has one input port by default. The number of block inputs is dynamic and depends on the number of signals you specify for the block. For example, if your block has four signals, it has four block inputs.

The block supports the following input signal data types: single, double, int8, int16, int32, int64, uint8, uint16, uint32, uint64, and boolean. The block does not support fixed-point data types.

Code generation to deploy models to targets. If your signal information consists of signed or unsigned integers greater than 32 bits long, code generation is not supported.

Output

`output` — CAN message output
single | double | int8 | int16 | int32 | int64 | uint32 | uint64 | boolean

This block has one output port, CAN Msg. The CAN Pack block takes the specified input parameters and packs the signals into a message.

Parameters

`Data input as` — Select your data signal

`raw data` (default) | `manually specified signals` | `CANdb specified signals`

- `raw data`: Input data as a uint8 vector array. If you select this option, you only specify the message fields. all other signal parameter fields are unavailable. This option opens only one input port on your block.

  The conversion formula is:

  ```
  raw_value = (physical_value - Offset) / Factor
  ```

  where `physical_value` is the original value of the signal and `raw_value` is the packed signal value.
- `manually specified signals`: Allows you to specify data signal definitions. If you select this option, use the **Signals** table to create your signals. The number of block inputs depends on the number of signals you specify.
- `CANdb specified signals`: Allows you to specify a CAN database file that contains message and signal definitions. If you select this option, select a CANdb file. The number of block inputs depends on the number of signals specified in the CANdb file for the selected message.

Programmatic Use
Block Parameter: `DataFormat`

`CANdb file` — CAN database file

character vector

This option is available if you specify that your data is input through a CANdb file in the **Data is input as** list. Click **Browse** to find the CANdb file on your system. The message list specified in the CANdb file populates the **Message** section of the dialog box. The CANdb file also populates the **Signals** table for the selected message.

File names that contain non-alphanumeric characters such as equal signs, ampersands, and so on are not valid CAN database file names. You can use periods in your database name. Before you use the CAN database files, rename them with non-alphanumeric characters.

Programmatic Use
Block Parameter: `CANdbFile`

`Message list` — CAN message list

array of character vectors

This option is available if you specify that your data is input through a CANdb file in the **Data is input as** field and you select a CANdb file in the **CANdb file** field. Select the message to display signal details in the **Signals** table.

Programmatic Use
Block Parameter: `MsgList`

`Name` — CAN message name

`CAN Msg` (default) | character vector

Specify a name for your CAN message. The default is `CAN Msg`. This option is available if you choose to input raw data or manually specify signals. This option is not available if you choose to use signals from a CANdb file.

Programmatic Use
Block Parameter: `MsgName`

`Identifier type` — CAN identifier type
`Standard (11-bit identifier)` (default) | `Extended (29-bit identifier)`

Specify whether your CAN message identifier is a `Standard` or an `Extended` type. The default is `Standard`. A standard identifier is an 11-bit identifier and an extended identifier is a 29-bit identifier. This option is available if you choose to input raw data or manually specify signals. For `CANdb specified signals`, the **Identifier type** inherits the type from the database.

Programmatic Use
Block Parameter: `MsgIDType`

`CAN Identifier` — CAN message identifier
`0` (default) | `0 .. 536870911`

Specify your CAN message ID. This number must be a positive integer from 0 through 2047 for a standard identifier and from 0 through 536870911 for an extended identifier. You can also specify hexadecimal values by using the `hex2dec` function. This option is available if you choose to input raw data or manually specify signals.

Programmatic Use
Block Parameter: `MsgIdentifier`

`Length (bytes)` — CAN message length
`8` (default) | `0 .. 8`

Specify the length of your CAN message from 0 to 8 bytes. If you are using `CANdb specified signals` for your data input, the CANdb file defines the length of your message. If not, this field defaults to `8`. This option is available if you choose to input raw data or manually specify signals.

Programmatic Use
Block Parameter: `MsgLength`

`Remote frame` — CAN message as remote frame
`off` (default) | `on`

Specify the CAN message as a remote frame.

Programmatic Use
Block Parameter: `Remote`

`Output as bus` — CAN message as bus
`off` (default) | `on`

Select this option for the block to output CAN messages as a Simulink bus signal. For more information on Simulink bus objects, see "Composite Signals".

Programmatic Use
Block Parameter: `BusOutput`

`Add signal` — Add CAN signal
character vector

Add a signal to the signal table.

Programmatic Use
Block Parameter: `AddSignal`

`Delete signal` — Remove CAN signal

character vector

Remove a signal from the signal table.

Programmatic Use
Block Parameter: `DeleteSignal`

`Signals` — Signals table

table

This table appears if you choose to specify signals manually or define signals by using a CANdb file.

If you are using a CANdb file, the data in the file populates this table and you cannot edit the fields. To edit signal information, switch to manually specified signals.

If you have selected to specify signals manually, create your signals in this table. Each signal that you create has these values:

Name

Specify a descriptive name for your signal. The Simulink block in your model displays this name. The default is `Signal [row number]`.

Start bit

Specify the start bit of the data. The start bit is the least significant bit counted from the start of the message data. The start bit must be an integer from 0 through 63.

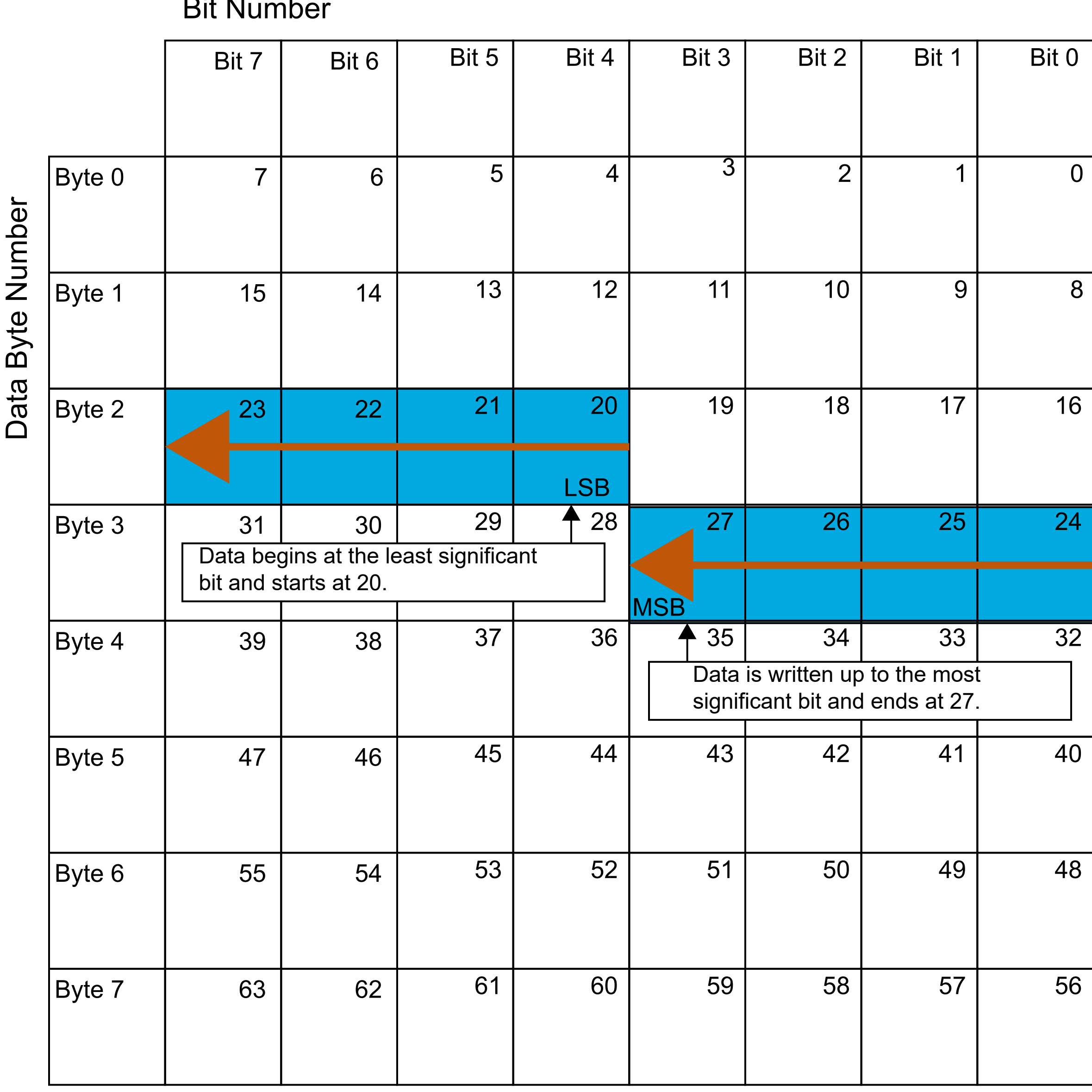
Length (bits)

Specify the number of bits the signal occupies in the message. The length must be an integer from 1 through 64.

Byte order

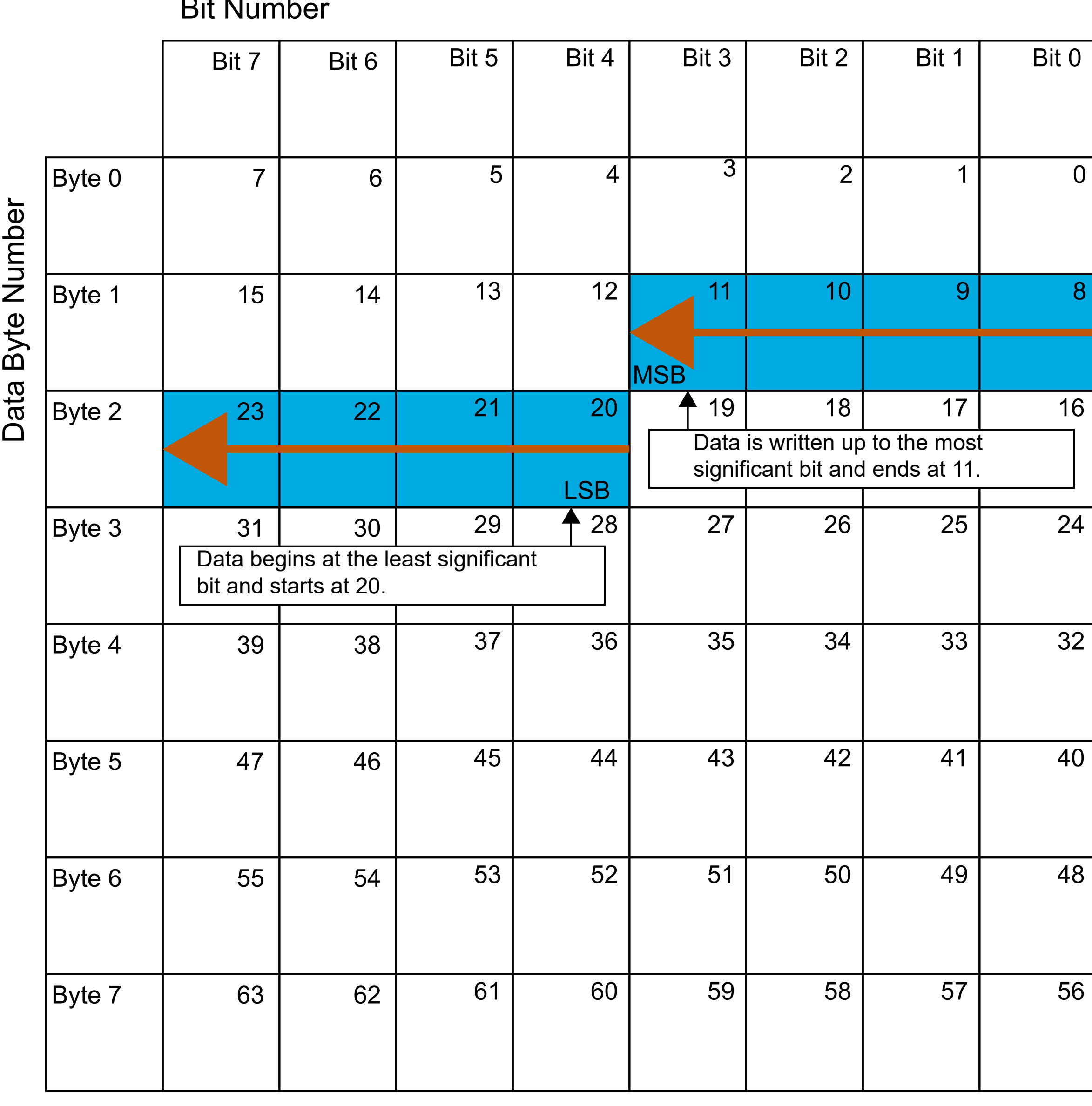
Select either of these options:

- `LE`: Where the byte order is in little-endian format (Intel). In this format you count bits from the least significant bit, to the most significant bit. For example, if you pack one byte of data in little-endian format, with the start bit at 20, the data bit table resembles this figure.



Little-Endian Byte Order Counted from the Least-Significant Bit to the Highest Address

- BE: Where byte order is in big-endian format (Motorola®). In this format you count bits from the least-significant bit to the most-significant bit. For example, if you pack one byte of data in big-endian format, with the start bit at 20, the data bit table resembles this figure.



Big-Endian Byte Order Counted from the Least Significant Bit to the Lowest Address

Data type

Specify how the signal interprets the data in the allocated bits. Choose from:

- `signed` (default)
- `unsigned`
- `single`
- `double`

Multiplex type

Specify how the block packs the signals into the CAN message at each time step:

- `Standard`: The signal is packed at each time step.
- `Multiplexor`: The `Multiplexor` signal, or the mode signal is packed. You can specify only one `Multiplexor` signal per message.
- `Multiplexed`: The signal is packed if the value of the `Multiplexor` signal (mode signal) at run time matches the configured **Multiplex value** of this signal.

For example, a message has these signals with the following types and values.

Signal Name	Multiplex Type	Multiplex Value
Signal-A	Standard	Not applicable
Signal-B	Multiplexed	1
Signal-C	Multiplexed	0
Signal-D	Multiplexor	Not applicable

In this example:

- The block packs Signal-A (Standard signal) and Signal-D (Multiplexor signal) in every time step.
- If the value of Signal-D is 1 at a particular time step, then the block packs Signal-B along with Signal-A and Signal-D in that time step.
- If the value of Signal-D is 0 at a particular time step, then the block packs Signal-C along with Signal-A and Signal-D in that time step.
- If the value of Signal-D is not 1 or 0, the block does not pack either of the Multiplexed signals in that time step.

Multiplex value

This option is available only if you have selected the **Multiplex type** to be `Multiplexed`. The value you provide must match the `Multiplexor` signal value at run time for the block to pack the `Multiplexed` signal. The **Multiplex value** must be a positive integer or zero.

Factor

Specify the **Factor** value to apply to convert the physical value (signal value) to the raw value packed in the message. For more information, see the **Data input as** parameter conversion formula.

Offset

Specify the **Offset** value to apply to convert the physical value (signal value) to the raw value packed in the message. For more information, see the **Data input as** parameter conversion formula.

Min, **Max**

Define a range of signal values. The default settings are `-Inf` (negative infinity) and `Inf`, respectively. For **CANdb specified signals**, these settings are read from the CAN database. For **manually specified signals**, you can specify the minimum and maximum physical value of the signal. By default, these settings do not clip signal values that exceed the settings.

Programmatic Use

Block Parameter: `SignalInfo`

See Also

CAN Unpack

Topics
"Design Your Model for Effective Acceleration"

Introduced in R2009a

CAN Unpack

Unpack individual signals from CAN messages

Library: Simulink Real-Time / CAN / CAN MSG blocks
Vehicle Network Toolbox / CAN Communication
Embedded Coder / Embedded Targets / Host Communication

Description

The CAN Unpack block unpacks a CAN message into signal data using the specified output parameters at every time step. Data is output as individual signals.

To use this block, you also need a license for Simulink software.

The CAN Unpack block supports:

- The use of Simulink Accelerator Rapid Accelerator mode. Using this feature, you can speed up the execution of Simulink models.
- The use of model referencing. Using this feature, your model can include other Simulink models as modular components.

For more information on these features, see “Design Your Model for Effective Acceleration”.

Tip

- To process every message coming through a channel, it is recommended that you use the CAN Unpack block in a function trigger subsystem. See “Using Triggered Subsystems”.
- To work with J1939 messages, use the blocks in the J1939 Communication block library instead of this block.

Ports

Input

`input` — CAN message input
single | double | int8 | int16 | int32 | int64 | uint32 | uint64 | boolean

This block has one input port, CAN Msg. The CAN Unpack block takes the specified input parameters and unpacks the signals into a message.

The block supports the following input signal data types: single, double, int8, int16, int32, int64, uint8, uint16, uint32, uint64, and boolean. The block does not support fixed-point data types.

Code generation to deploy models to targets. Code generation is not supported if your signal information consists of signed or unsigned integers greater than 32 bits long.

Output

`output` — CAN message output
single | double | int8 | int16 | int32 | int64 | uint32 | uint64 | boolean

The CAN Unpack block has one output port by default. The number of output ports is dynamic and depends on the number of signals that you specify for the block to output. For example, if your block has four signals, the block has four output ports.

Selecting an **Output ports** option adds an output port to your block. For more information, see parameters `Output identifier`, `Output timestamp`, `Output error`, `Output remote`, `Output length`, and `Output status`.

For manually or CANdb specified signals, the default output signal data type is double. To specify other types, use a Signal Specification block. This allows the block to support the following output signal data types: single, double, int8, int16, int32, int64, uint8, uint16, uint32, uint64, and boolean. The block does not support fixed-point types.

Parameters

`Data to output as` — Select your data signal
`raw data` (default) | `manually specify signals` | `CANdb specified signals`

- `raw data`: Output data as a uint8 vector array. If you select this option, you specify only the message fields. The other signal parameter fields are unavailable. This option opens only one output port on your block.

  The conversion formula is:

  ```
  physical_value = raw_value * Factor + Offset
  ```

  where `raw_value` is the unpacked signal value and `physical_value` is the scaled signal value.
- `manually specified signals`: You can specify data signals. If you select this option, use the `Signals` table to create your signals message manually. The number of output ports on your block depends on the number of signals that you specify. For example, if you specify four signals, your block has four output ports.
- `CANdb specified signals`: You can specify a CAN database file that contains data signals. If you select this option, select a CANdb file. The number of output ports on your block depends on the number of signals specified in the CANdb file. For example, if the selected message in the CANdb file has four signals, your block has four output ports.

Programmatic Use
Block Parameter: `DataFormat`

`CANdb file` — CAN database file
character vector

This option is available if you specify that your data is input via a CANdb file in the **Data to be output as** list. Click **Browse** to find the CANdb file on your system. The messages and signal definitions specified in the CANdb file populate the **Message** section of the dialog box. The signals specified in the CANdb file populate **Signals** table. File names that contain non-alphanumeric characters such as equal signs, ampersands, and so forth are not valid CAN database file names. You can use periods in your database name. Rename CAN database files with non-alphanumeric characters before you use them.

Programmatic Use
Block Parameter: `CANdbFile`

`Message list` — CAN message list

array of character vectors

This option is available if you specify in the **Data to be output as** list that your data is to be output as a CANdb file and you select a CANdb file in the **CANdb file** field. You can select the message that you want to view. The **Signals** table then displays the details of the selected message.

Programmatic Use
Block Parameter: `MsgList`

`Name` — CAN message name

`CAN Msg` (default) | character vector

Specify a name for your CAN message. The default is `CAN Msg`. This option is available if you choose to output raw data or manually specify signals.

Programmatic Use
Block Parameter: `MsgName`

`Identifier type` — CAN identifier type

`Standard (11-bit identifier)` (default) | `Extended (29-bit identifier)`

Specify whether your CAN message identifier is a `Standard` or an `Extended` type. The default is `Standard`. A standard identifier is an 11-bit identifier and an extended identifier is a 29-bit identifier. This option is available if you choose to output raw data or manually specify signals. For CANdb-specified signals, the **Identifier type** inherits the type from the database.

Programmatic Use
Block Parameter: `MsgIDType`

`CAN Identifier` — CAN message identifier

`0` (default) | `0 .. 536870911`

Specify your CAN message ID. This number must be a integer from 0 through 2047 for a standard identifier and from 0 through 536870911 for an extended identifier. If you specify `-1`, the block unpacks the messages that match the length specified for the message. You can also specify hexadecimal values by using the `hex2dec` function. This option is available if you choose to output raw data or manually specify signals.

Programmatic Use
Block Parameter: `MsgIdentifier`

`Length (bytes)` — CAN message length

`8` (default) | `0 .. 8`

Specify the length of your CAN message from 0 to 8 bytes. If you are using `CANdb specified signals` for your output data, the CANdb file defines the length of your message. If not, this field defaults to `8`. This option is available if you choose to output raw data or manually specify signals.

Programmatic Use
Block Parameter: `MsgLength`

`Add signal` — Add CAN signal

character vector

Add a signal to the signal table.

Programmatic Use
Block Parameter: `AddSignal`

`Delete signal` — Remove CAN signal
character vector

Remove a signal from the signal table.

Programmatic Use
Block Parameter: `DeleteSignal`

`Signals` — Signals table
table

If you choose to specify signals manually or define signals by using a CANdb file, this table appears.

If you are using a CANdb file, the data in the file populates this table and you cannot edit the fields. To edit signal information, switch to specified signals.

If you have selected to specify signals manually, create your signals manually in this table. Each signal that you create has these values:

Name

Specify a descriptive name for your signal. The Simulink block in your model displays this name. The default is `Signal [row number]`.

Start bit

Specify the start bit of the data. The start bit is the least significant bit counted from the start of the message. The start bit must be an integer from 0 through 63.

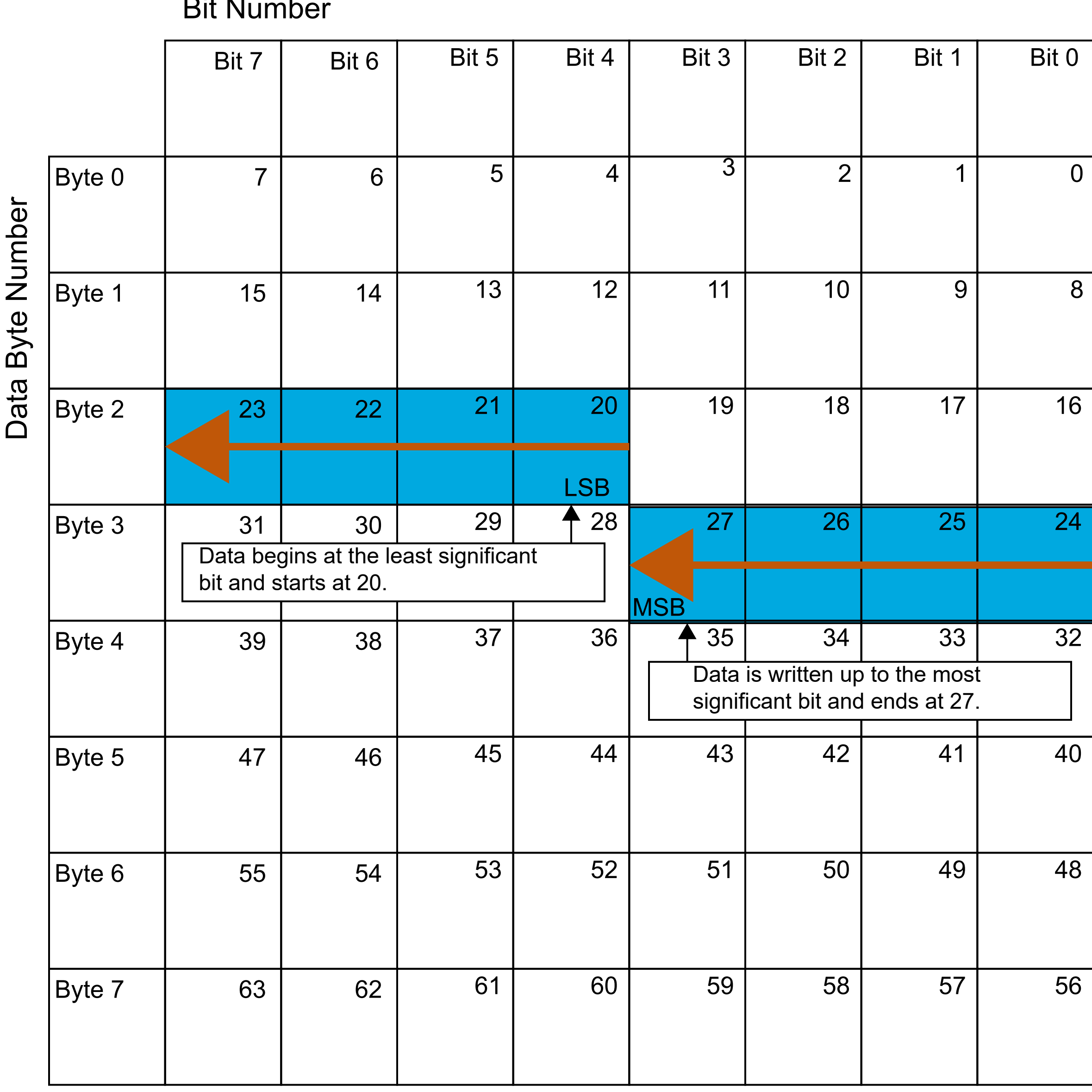
Length (bits)

Specify the number of bits the signal occupies in the message. The length must be an integer from 1 through 64.

Byte order

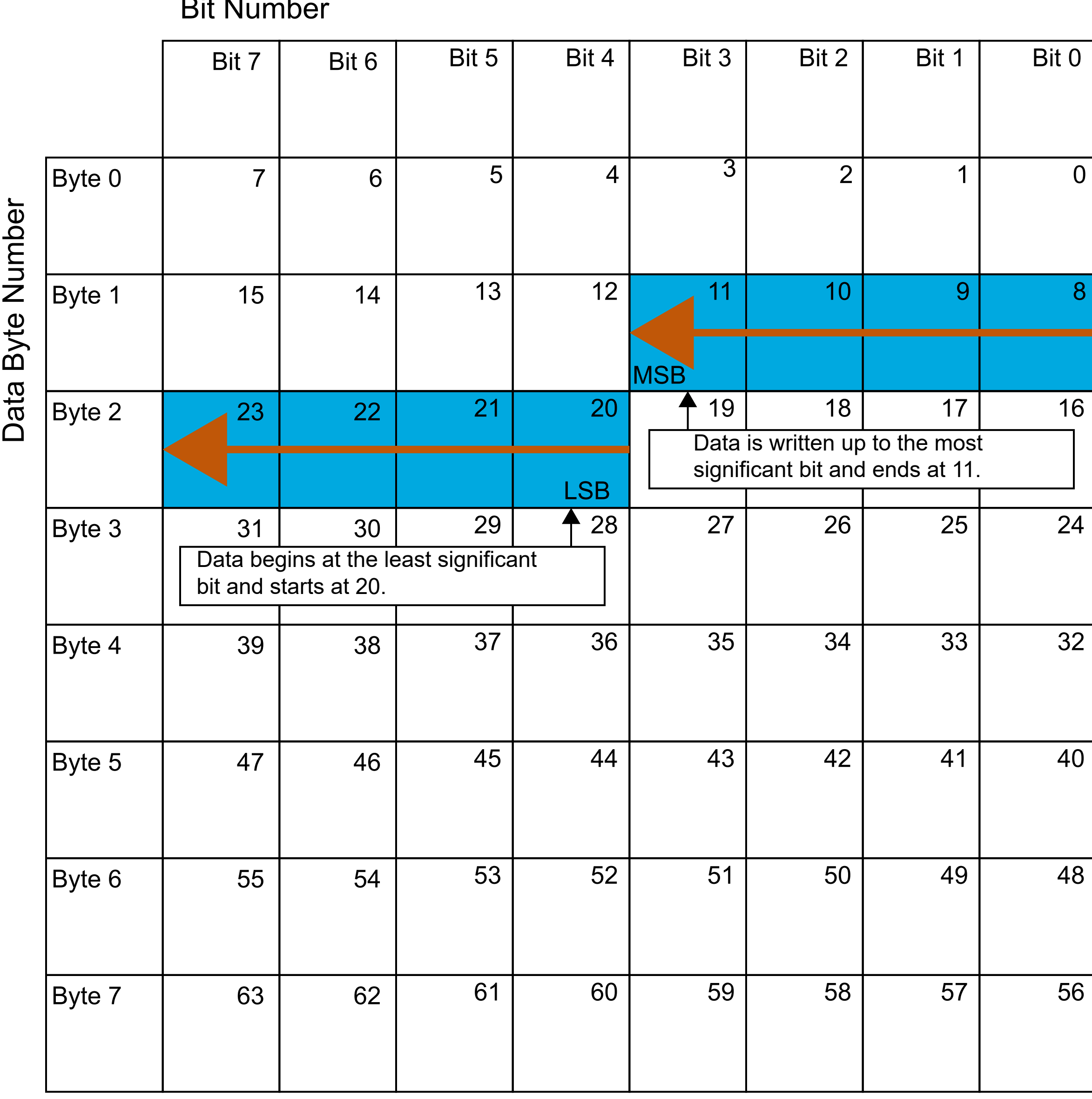
Select either of these options:

- `LE`: Where the byte order is in little-endian format (Intel). In this format you count bits from the least-significant bit to the most-significant bit. For example, if you pack one byte of data in little-endian format, with the start bit at 20, the data bit table resembles this figure.



Little-Endian Byte Order Counted from the Least Significant Bit to the Highest Address

- BE: Where the byte order is in big-endian format (Motorola). In this format you count bits from the least-significant bit to the most-significant bit. For example, if you pack one byte of data in big-endian format, with the start bit at 20, the data bit table resembles this figure.



Big-Endian Byte Order Counted from the Least Significant Bit to the Lowest Address

Data type

Specify how the signal interprets the data in the allocated bits. Choose from:

- `signed` (default)
- `unsigned`
- `single`
- `double`

Multiplex type

Specify how the block unpacks the signals from the CAN message at each time step:

- `Standard`: The signal is unpacked at each time step.
- `Multiplexor`: The `Multiplexor` signal or the mode signal is unpacked. You can specify only one `Multiplexor` signal per message.
- `Multiplexed`: The signal is unpacked if the value of the `Multiplexor` signal (mode signal) at run time matches the configured **Multiplex value** of this signal.

For example, a message has four signals with these values.

Signal Name	Multiplex Type	Multiplex Value
Signal-A	Standard	Not applicable
Signal-B	Multiplexed	1
Signal-C	Multiplexed	0
Signal-D	Multiplexor	Not applicable

In this example:

- The block unpacks Signal-A (Standard signal) and Signal-D (Multiplexor signal) in every time step.
- If the value of Signal-D is 1 at a particular time step, then the block unpacks Signal-B along with Signal-A and Signal-D in that time step.
- If the value of Signal-D is 0 at a particular time step, then the block unpacks Signal-C along with Signal-A and Signal-D in that time step.
- If the value of Signal-D is not 1 or 0, the block does not unpack either of the Multiplexed signals in that time step.

Multiplex value

This option is available only if you have selected the **Multiplex type** to be `Multiplexed`. The value you provide must match the `Multiplexor` signal value at run time for the block to unpack the `Multiplexed` signal. The **Multiplex value** must be a positive integer or zero.

Factor

Specify the **Factor** value applied to convert the unpacked raw value to the physical value (signal value). For more information, see the **Data input as** parameter conversion formula.

Offset

Specify the **Offset** value applied to convert the physical value (signal value) to the unpacked raw value. For more information, see the **Data input as** parameter conversion formula.

Min, **Max**

Define a range of raw signal values. The default settings are `-Inf` (negative infinity) and `Inf`, respectively. For **CANdb specified signals**, these settings are read from the CAN database. For **manually specified signals**, you can specify the minimum and maximum physical value of the signal. By default, these settings do not clip signal values that exceed them.

Programmatic Use
Block Parameter: `SignalInfo`

`Output identifier` — CAN message identifier

character vector

Select this option to output a CAN message identifier. The data type of this port is **uint32**.

Programmatic Use
Block Parameter: `IDPort`

`Output timestamp` — Enable CAN message timestamp
`off` (default) | `on`

Select this option to output the message timestamp. This value indicates when the message was received, measured as the number of seconds elapsed since the model simulation began. This option adds a new output port to the block. The data type of this port is **double**.

Programmatic Use
Block Parameter: `TimestampPort`

`Output error` — Enable CAN message error status
`off` (default) | `on`

Select this option to output the message error status. This option adds a new output port to the block. An output value of `1` on this port indicates that the incoming message is an error frame. If the output value is `0`, there is no error. The data type of this port is **uint8**.

Programmatic Use
Block Parameter: `ErrorPort`

`Output remote` — Enable CAN message remote frame status
`off` (default) | `on`

Select this option to output the message remote frame status. This option adds a new output port to the block. The data type of this port is **uint8**.

Programmatic Use
Block Parameter: `RemotePort`

`Output length` — Enable CAN message length
`off` (default) | `on`

Select this option to output the length of the message in bytes. This option adds a new output port to the block. The data type of this port is **uint8**.

Programmatic Use
Block Parameter: `LengthPort`

`Output status` — Enable status output
`off` (default) | `on`

Select this option to output the message received status. The status is `1` if the block receives a new message and `0` if it does not. This option adds a new output port to the block. The data type of this port is **uint8**.

Programmatic Use
Block Parameter: `StatusPort`

See Also

CAN Pack

Topics

"Design Your Model for Effective Acceleration"

Introduced in R2009a

CAN FD Pack

Pack individual signals into message for CAN FD bus

Library: Vehicle Network Toolbox / CAN FD Communication
Simulink Real-Time / CAN / CAN-FD MSG blocks
Embedded Coder Support Package for Texas Instruments C2000 Processors / Target Communication

Description

The CAN FD Pack block loads signal data into a message at specified intervals during the simulation.

To use this block, you also need a license for Simulink software.

The CAN FD Pack block supports:

- The use of Simulink Accelerator mode. Using this feature, you can speed up the execution of Simulink models. For more information, see “Design Your Model for Effective Acceleration”.

Tip

- To work with J1939 messages, use the blocks in the J1939 Communication block library instead of this block.

Ports

Input

`input` — CAN FD message input
single | double | int8 | int16 | int32 | int64 | uint32 | uint64 | boolean

The CAN FD Pack block has one input port by default. The number of block inputs is dynamic and depends on the number of signals that you specify for the block. For example, if your block has four signals, it has four block inputs.

Code generation to deploy models to targets. Code generation is not supported if your signal information consists of signed or unsigned integers greater than 32 bits long.

Output

`output` — CAN message output
single | double | int8 | int16 | int32 | int64 | uint32 | uint64 | boolean

This block has one output port, Msg. The CAN FD Pack block takes the specified input parameters and packs the signals into a bus message.

The block outputs CAN FD messages as a Simulink bus signal. For more information on Simulink bus objects, see “Composite Signals”.

Parameters

`Data input as` — Select your data signal

`raw data` (default) | `manually specified signals` | `CANdb specified signals`

- `raw data`: Input data as a uint8 vector array. If you select this option, you only specify the message fields. all other signal parameter fields are unavailable. This option opens only one input port on your block.

  The conversion formula is:

  ```
  raw_value = (physical_value - Offset) / Factor
  ```

  where `physical_value` is the original value of the signal and `raw_value` is the packed signal value.
- `manually specified signals`: Allows you to specify data signal definitions. If you select this option, use the **Signals** table to create your signals. The number of block inputs depends on the number of signals you specify.
- `CANdb specified signals`: Allows you to specify a CAN database file that contains message and signal definitions. If you select this option, select a CANdb file. The number of block inputs depends on the number of signals specified in the CANdb file for the selected message.

Programmatic Use
Block Parameter: `DataFormat`

`CANdb file` — CAN database file

character vector

This option is available if you specify that your data is input through a CANdb file in the **Data is input as** list. Click **Browse** to find the CANdb file on your system. The message list specified in the CANdb file populates the **Message** section of the dialog box. The CANdb file also populates the **Signals** table for the selected message. File names that contain non-alphanumeric characters such as equal signs, ampersands, and so on are not valid CAN database file names. You can use periods in your database name. Before you use the CAN database files, rename them with non-alphanumeric characters.

Programmatic Use
Block Parameter: `CANdbFile`

`Message list` — CAN message list

array of character vectors

This option is available if you specify that your data is input through a CANdb file in the **Data is input as** field and you select a CANdb file in the **CANdb file** field. Select the message to display signal details in the **Signals** table.

Programmatic Use
Block Parameter: `MsgList`

`Name` — CAN FD message name

`CAN Msg` (default) | character vector

Specify a name for your CAN FD message. The default is `CAN Msg`. This option is available if you choose to input raw data or manually specify signals. This option is not available if you choose to use signals from a CANdb file.

Programmatic Use
Block Parameter: `MsgName`

`Protocol mode` — CAN FD message protocol
`CAN FD` (default) | `CAN`

Specify the message protocol mode.

Programmatic Use
Block Parameter: `ProtocolMode`

`Identifier type` — CAN identifier type
`Standard (11-bit identifier)` (default) | `Extended (29-bit identifier)`

Specify whether your CAN message identifier is a `Standard` or an `Extended` type. The default is `Standard`. A standard identifier is an 11-bit identifier and an extended identifier is a 29-bit identifier. This option is available if you choose to input raw data or manually specify signals. For `CANdb specified signals`, the **Identifier type** inherits the type from the database.

Programmatic Use
Block Parameter: `MsgIDType`

`Identifier` — Message identifier
`0` (default) | `0 .. 536870911`

Specify your message ID. This number must be a positive integer from 0 through 2047 for a standard identifier and from 0 through 536870911 for an extended identifier. You can also specify hexadecimal values by using the `hex2dec` function. This option is available if you choose to input raw data or manually specify signals.

Programmatic Use
Block Parameter: `MsgIdentifier`

`Length (bytes)` — CAN FD message length
`8` (default) | `0 .. 8`

Specify the length of your message. For CAN messages the value can be 0-8 bytes; for CAN FD the value can be 0-8, 12, 16, 20, 24, 32, 48, or 64 bytes. If you are using `CANdb specified signals` for your data input, the CANdb file defines the length of your message. This option is available if you choose to input raw data or manually specify signals.

Programmatic Use
Block Parameter: `MsgLength`

`Remote frame` — CAN message as remote frame
`off` (default) | `on`

(Disabled for CAN FD protocol mode.) Specify the CAN message as a remote frame.

Programmatic Use
Block Parameter: `Remote`

`Bit Rate Switch (BRS)` — Enable bit rate switch
`off` (default) | `on`

(Disabled for CAN protocol mode.) Enable bit rate switch.

Programmatic Use
Block Parameter: `BRSSwitch`

`Add signal` — Add CAN FD signal
character vector

Add a signal to the signal table.

Programmatic Use
Block Parameter: `AddSignal`

`Delete signal` — Remove CAN FD signal
character vector

Remove a signal from the signal table.

Programmatic Use
Block Parameter: `DeleteSignal`

`Signals` — Signals table
table

This table appears if you choose to specify signals manually or define signals by using a CANdb file.

If you are using a CANdb file, the data in the file populates this table and you cannot edit the fields. To edit signal information, switch to manually specified signals.

If you have selected to specify signals manually, create your signals in this table. Each signal that you create has these values:

Name

Specify a descriptive name for your signal. The Simulink block in your model displays this name. The default is `Signal [row number]`.

Start bit

Specify the start bit of the data. The start bit is the least significant bit counted from the start of the message data. For CAN the start bit must be an integer from 0 through 63, for CAN FD 0 through 511, within the number of bits in the message. (Note that message length is specified in bytes.)

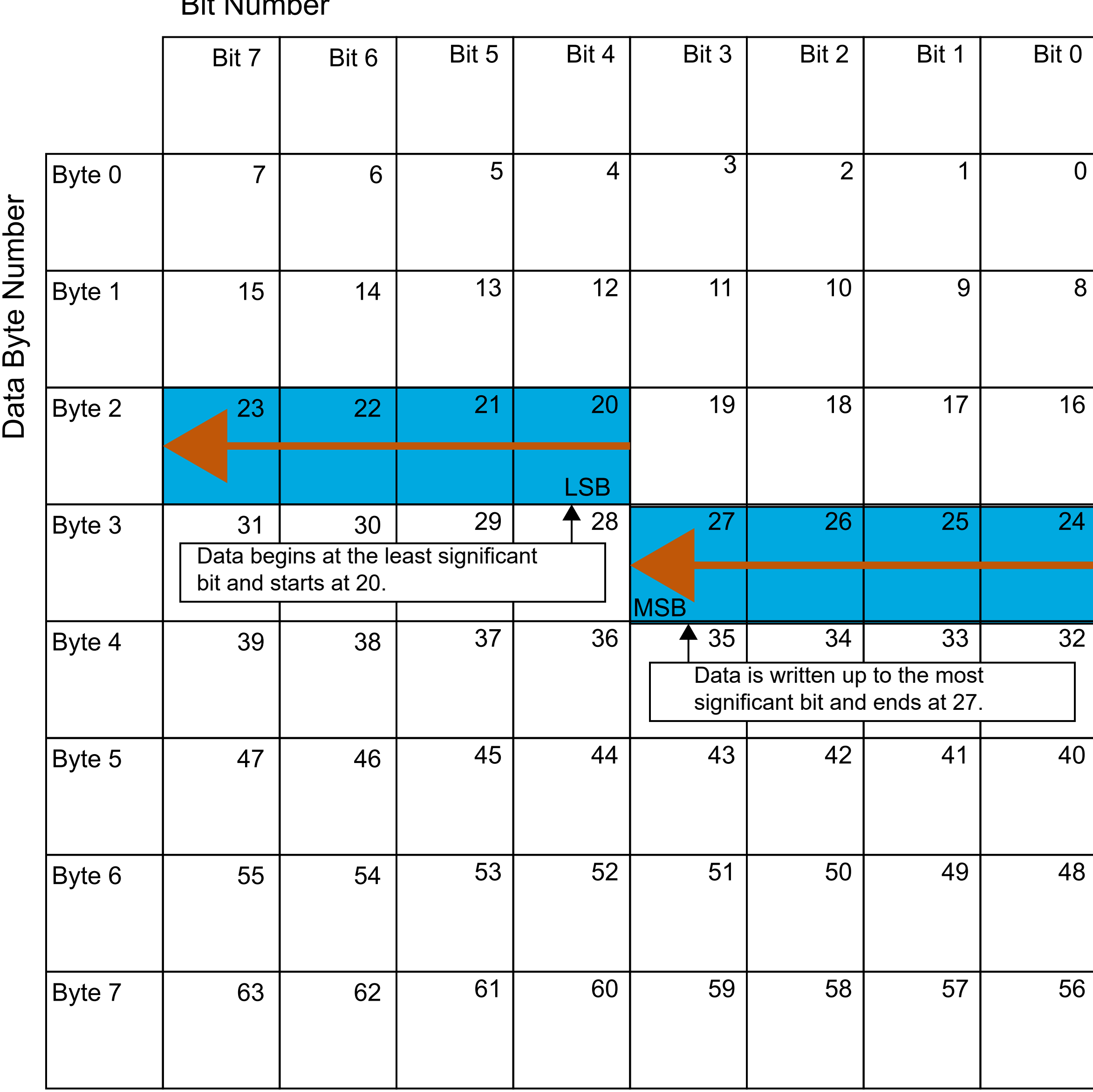
Length (bits)

Specify the number of bits the signal occupies in the message. The length must be an integer from 1 through 64. The sum of all the signal lengths in a message is limited to the number of bits in the message length; that is, all signals must cumulatively fit within the length of the message. (Note that message length is specified in bytes and signal length in bits.)

Byte order

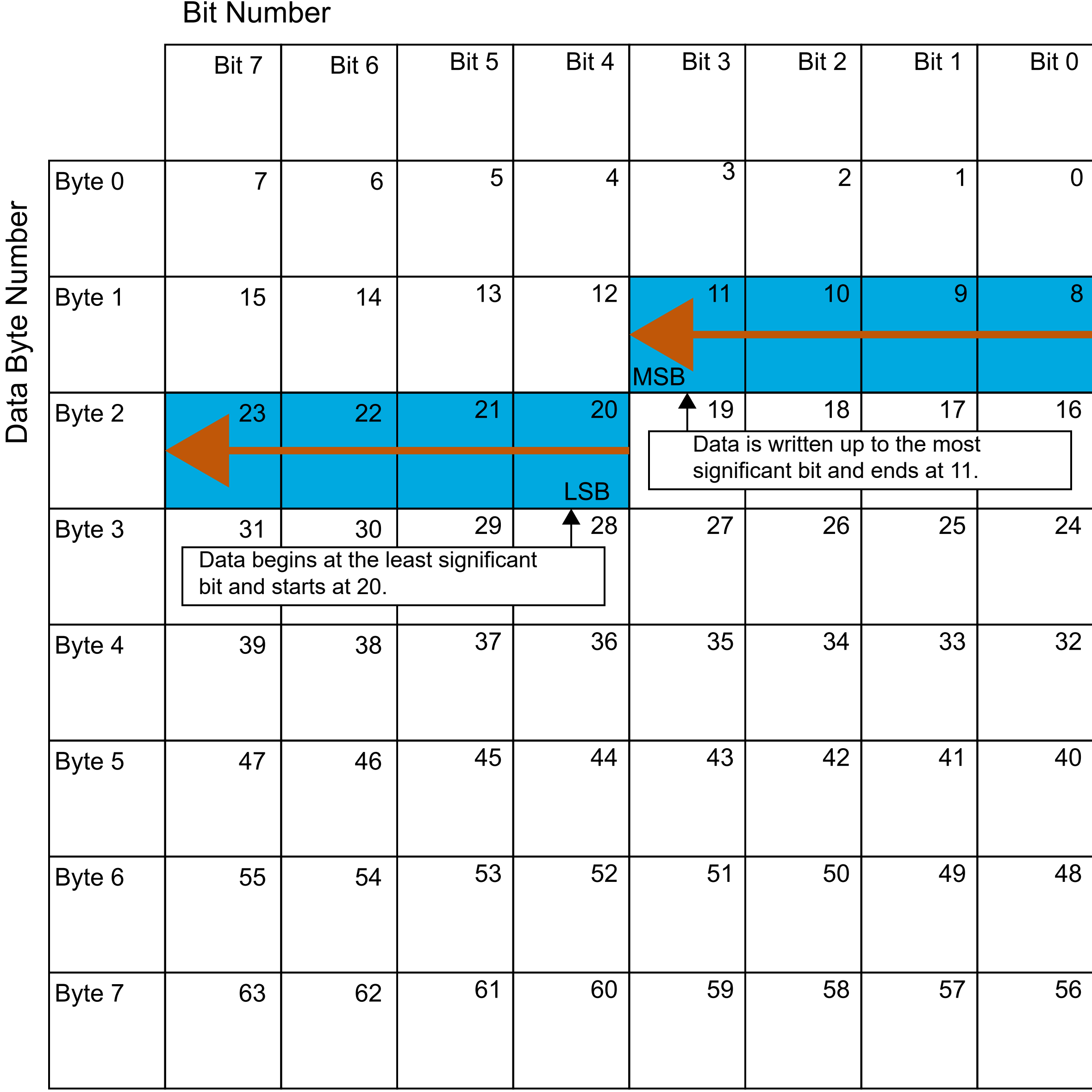
Select either of these options:

- `LE`: Where the byte order is in little-endian format (Intel). In this format you count bits from the least significant bit, to the most significant bit. For example, if you pack one byte of data in little-endian format, with the start bit at 20, the data bit table resembles this figure.



Little-Endian Byte Order Counted from the Least-Significant Bit to the Highest Address

- BE: Where byte order is in big-endian format (Motorola). In this format you count bits from the least-significant bit to the most-significant bit. For example, if you pack one byte of data in big-endian format, with the start bit at 20, the data bit table resembles this figure.



Big-Endian Byte Order Counted from the Least Significant Bit to the Lowest Address

Data type

Specify how the signal interprets the data in the allocated bits. Choose from:

- `signed` (default)
- `unsigned`
- `single`
- `double`

Note: If you have a `double` signal that does not align exactly to the message byte boundaries, to generate code with Embedded Coder you must check **Support long long** under **Device Details** in the **Hardware Implementation** pane of the Configuration Parameters dialog.

Multiplex type

Specify how the block packs the signals into the message at each time step:

- `Standard`: The signal is packed at each time step.
- `Multiplexor`: The `Multiplexor` signal, or the mode signal is packed. You can specify only one `Multiplexor` signal per message.
- `Multiplexed`: The signal is packed if the value of the `Multiplexor` signal (mode signal) at run time matches the configured **Multiplex value** of this signal.

For example, a message has four signals with these types and values.

Signal Name	Multiplex Type	Multiplex Value
Signal-A	Standard	Not applicable
Signal-B	Multiplexed	1
Signal-C	Multiplexed	0
Signal-D	Multiplexor	Not applicable

In this example:

- The block packs Signal-A (Standard signal) and Signal-D (Multiplexor signal) in every time step.
- If the value of Signal-D is 1 at a particular time step, then the block packs Signal-B along with Signal-A and Signal-D in that time step.
- If the value of Signal-D is 0 at a particular time step, then the block packs Signal-C along with Signal-A and Signal-D in that time step.
- If the value of Signal-D is not 1 or 0, the block does not pack either of the Multiplexed signals in that time step.

Multiplex value

This option is available only if you have selected the **Multiplex type** to be `Multiplexed`. The value you provide here must match the `Multiplexor` signal value at run time for the block to pack the `Multiplexed` signal. The **Multiplex value** must be a positive integer or zero.

Factor

Specify the **Factor** value to apply to convert the physical value (signal value) to the raw value packed in the message. See the **Data input as** parameter conversion formula to understand how physical values are converted to raw values packed into a message.

Offset

Specify the **Offset** value to apply to convert the physical value (signal value) to the raw value packed in the message. See the **Data input as** parameter conversion formula to understand how physical values are converted to raw values packed into a message.

Min, **Max**

Define a range of signal values. The default settings are `-Inf` (negative infinity) and `Inf`, respectively. For **CANdb specified signals**, these settings are read from the CAN database. For

manually specified signals, you can specify the minimum and maximum physical value of the signal. By default, these settings do not clip signal values that exceed them.

Programmatic Use
Block Parameter: `SignalInfo`

See Also

Blocks

Topics
"Design Your Model for Effective Acceleration"
"Composite Signals"

Introduced in R2018a

CAN FD Unpack

Unpack individual signals from CAN FD messages

Library: Vehicle Network Toolbox / CAN FD Communication
Simulink Real-Time / CAN / CAN-FD MSG blocks
Embedded Coder Support Package for Texas Instruments C2000 Processors / Target Communication

Description

The CAN FD Unpack block unpacks a CAN FD message into signal data by using the specified output parameters at every time step. Data is output as individual signals.

To use this block, you also need a license for Simulink software.

The CAN FD Unpack block supports:

- Simulink Accelerator mode. You can speed up the execution of Simulink models. For more information, see “Design Your Model for Effective Acceleration”.

Tip

- To process every message coming through a channel, it is recommended that you use the CAN FD Unpack block in a function trigger subsystem. See “Using Triggered Subsystems”.
- To work with J1939 messages, use the blocks in the J1939 Communication block library instead of this block.

Ports

Input

`input` — CAN message input
single | double | int8 | int16 | int32 | int64 | uint32 | uint64 | boolean

This block has one input port, CAN Msg. The CAN Unpack block takes the specified input parameters and unpacks the signals into a message.

The block supports the following input signal data types: single, double, int8, int16, int32, int64, uint8, uint16, uint32, uint64, and boolean. The block does not support fixed-point data types.

Code generation to deploy models to targets. Code generation is not supported if your signal information consists of signed or unsigned integers greater than 32 bits long.

Output

`output` — CAN message output
single | double | int8 | int16 | int32 | int64 | uint32 | uint64 | boolean

The CAN FD Unpack block has one output port by default. The number of output ports is dynamic and depends on the number of signals you specify for the block to output. For example, if your block has four signals, it has four output ports.

If you do not select an **Output ports** option, the number of output ports on your block depends on the number of signals that you specify.

For manually or CANdb specified signals, the default output signal data type is double. To specify other types, use a Signal Specification block. This allows the block to support the following output signal data types: single, double, int8, int16, int32, int64, uint8, uint16, uint32, uint64, and boolean. The block does not support fixed-point types.

Parameters

`Data to output as` — Select your data signal
`raw data` (default) | `manually specify signals` | `CANdb specified signals`

- `raw data`: Output data as a uint8 vector array. If you select this option, you specify only the message fields. The other signal parameter fields are unavailable. This option opens only one output port on your block.

  The conversion formula is:

  ```
  physical_value = raw_value * Factor + Offset
  ```

  where `raw_value` is the unpacked signal value and `physical_value` is the scaled signal value.
- `manually specified signals`: You can specify data signals. If you select this option, use the `Signals` table to create your signals message manually. The number of output ports on your block depends on the number of signals that you specify. For example, if you specify four signals, your block has four output ports.
- `CANdb specified signals`: You can specify a CAN database file that contains data signals. If you select this option, select a CANdb file. The number of output ports on your block depends on the number of signals specified in the CANdb file. For example, if the selected message in the CANdb file has four signals, your block has four output ports.

Programmatic Use
Block Parameter: `DataFormat`

`CANdb file` — CAN database file
character vector

This option is available if you specify that your data is input via a CANdb file in the **Data to be output as** list. Click **Browse** to find the CANdb file on your system. The messages and signal definitions specified in the CANdb file populate the **Message** section of the dialog box. The signals specified in the CANdb file populate **Signals** table. File names that contain non-alphanumeric characters such as equal signs, ampersands, and so forth are not valid CAN database file names. You can use periods in your database name. Rename CAN database files with non-alphanumeric characters before you use them.

Programmatic Use
Block Parameter: `CANdbFile`

`Message list` — Message list
array of character vectors

This option is available if you specify in the **Data to be output as** list that your data is to be output as a CANdb file and you select a CANdb file in the **CANdb file** field. You can select the message that you want to view. The **Signals** table then displays the details of the selected message.

Programmatic Use
Block Parameter: `MsgList`

Name — Message name
`CAN Msg` (default) | character vector

Specify a name for your message. The default is `Msg`. This option is available if you choose to output raw data or manually specify signals.

Programmatic Use
Block Parameter: `MsgName`

`Identifier type` — Identifier type
`Standard (11-bit identifier)` (default) | `Extended (29-bit identifier)`

Specify whether your message identifier is a `Standard` or an `Extended` type. The default is `Standard`. A standard identifier is an 11-bit identifier and an extended identifier is a 29-bit identifier. This option is available if you choose to output raw data or manually specify signals. For CANdb-specified signals, the **Identifier type** inherits the type from the database.

Programmatic Use
Block Parameter: `MsgIDType`

`Identifier` — Message identifier
`0` (default) | `0 .. 536870911`

Specify your message ID. This number must be a integer from 0 through 2047 for a standard identifier and from 0 through 536870911 for an extended identifier. If you specify `–1`, the block unpacks the messages that match the length specified for the message. You can also specify hexadecimal values using the `hex2dec` function. This option is available if you choose to output raw data or manually specify signals.

Programmatic Use
Block Parameter: `MsgIdentifier`

`Length (bytes)` — CAN message length
`8` (default) | `0 .. 8`

Specify the length of your message. For CAN messages the value can be 0-8 bytes; for CAN FD the value can be 0-8, 12, 16, 20, 24, 32, 48, or 64 bytes. If you are using `CANdb specified signals` for your output data, the CANdb file defines the length of your message. This option is available if you choose to output raw data or manually specify signals.

Programmatic Use
Block Parameter: `MsgLength`

`Add signal` — Add CAN signal
character vector

Add a signal to the signal table.

Programmatic Use
Block Parameter: `AddSignal`

`Delete signal` — Remove CAN signal
character vector

Remove a signal from the signal table.

Programmatic Use
Block Parameter: `DeleteSignal`

`Signals` — Signals table
table

If you choose to specify signals manually or define signals by using a CANdb file, this table appears.

If you are using a CANdb file, the data in the file populates this table and you cannot edit the fields. To edit signal information, switch to specified signals.

If you have selected to specify signals manually, create your signals manually in this table. Each signal that you create has these values:

Name

Specify a descriptive name for your signal. The Simulink block in your model displays this name. The default is `Signal [row number]`.

Start bit

Specify the start bit of the data. The start bit is the least significant bit counted from the start of the message data. For CAN the start bit must be an integer from 0 through 63, for CAN FD 0 through 511, within the number of bits in the message. (Note that message length is specified in bytes.)

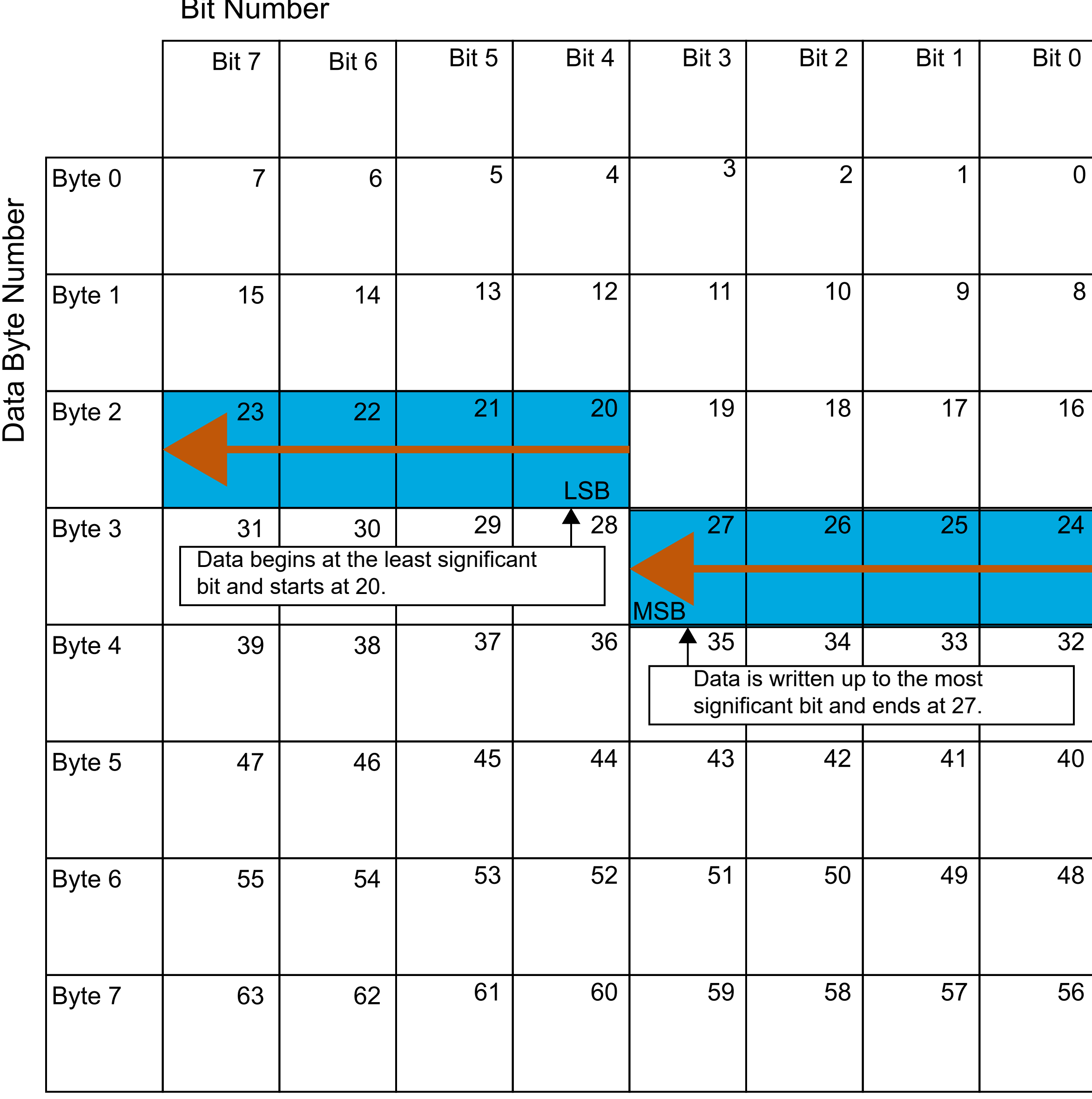
Length (bits)

Specify the number of bits the signal occupies in the message. The length must be an integer from 1 through 64. The sum of all the signal lengths in a message is limited to the number of bits in the message length; that is, all signals must cumulatively fit within the length of the message. (Note that message length is specified in bytes and signal length in bits.)

Byte order

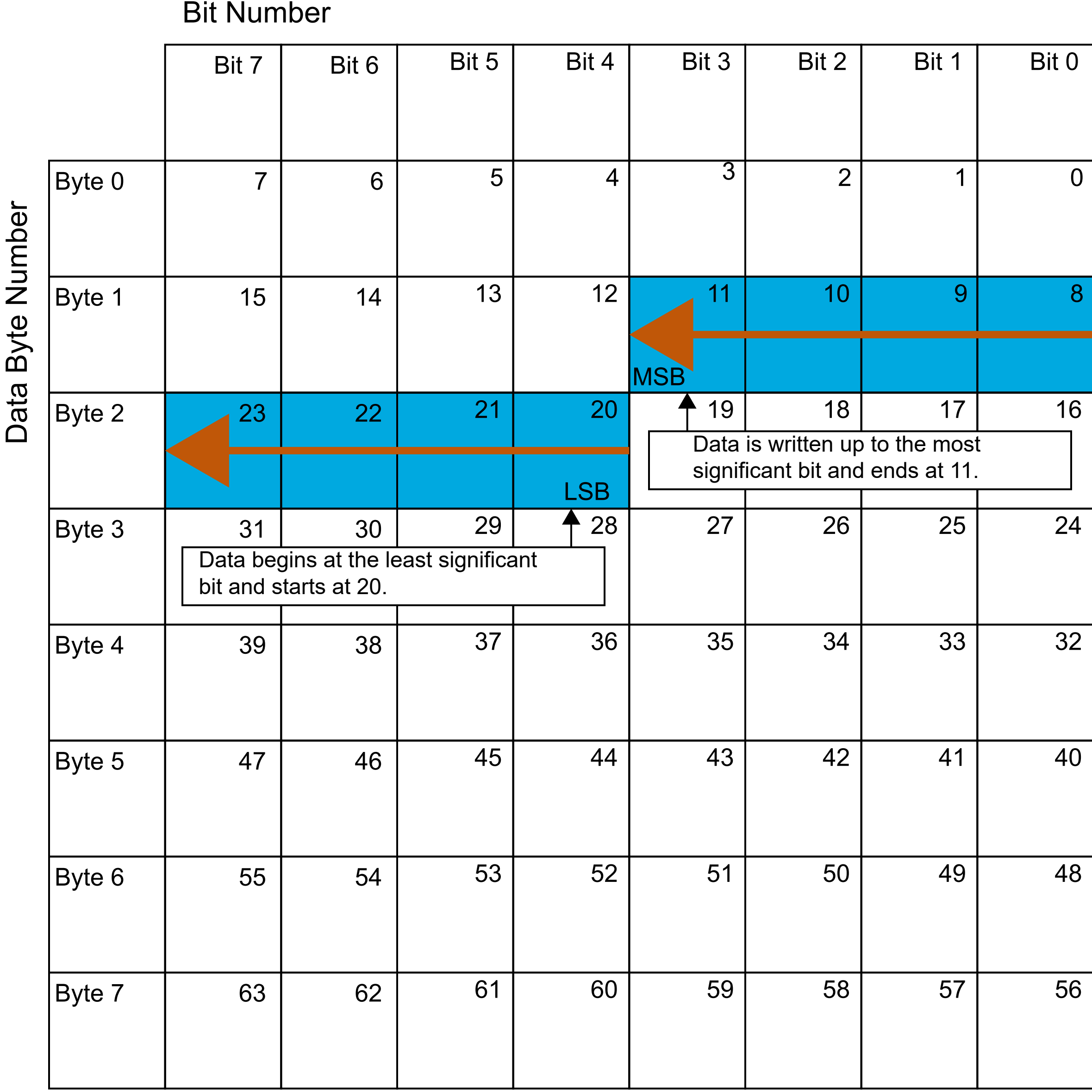
Select either of the following options:

- `LE`: Where the byte order is in little-endian format (Intel). In this format you count bits from the least-significant bit to the most-significant bit and proceeding to the next higher byte as you cross a byte boundary. For example, if you pack one byte of data in little-endian format, with the start bit at 20, the data bit table resembles this figure.



Little-Endian Byte Order Counted from the Least Significant Bit to the Highest Address

- BE: Where the byte order is in big-endian format (Motorola). In this format you count bits from the least-significant bit to the most-significant bit and proceeding to the next lower byte as you cross a byte boundary. For example, if you pack one byte of data in big-endian format, with the start bit at 20, the data bit table resembles this figure.



Big-Endian Byte Order Counted from the Least Significant Bit to the Lowest Address

Data type

Specify how the signal interprets the data in the allocated bits. Choose from:

- `signed` (default)
- `unsigned`
- `single`
- `double`

Note: If you have a `double` signal that does not align exactly to the message byte boundaries, to generate code with Embedded Coder you must check **Support long long** under **Device Details** in the **Hardware Implementation** pane of the Configuration Parameters dialog.

Multiplex type

Specify how the block unpacks the signals from the message at each time step:

- `Standard`: The signal is unpacked at each time step.
- `Multiplexor`: The `Multiplexor` signal, or the mode signal is unpacked. You can specify only one `Multiplexor` signal per message.
- `Multiplexed`: The signal is unpacked if the value of the `Multiplexor` signal (mode signal) at run time matches the configured **Multiplex value** of this signal.

For example, a message has four signals with these values.

Signal Name	Multiplex Type	Multiplex Value
Signal-A	Standard	Not applicable
Signal-B	Multiplexed	1
Signal-C	Multiplexed	0
Signal-D	Multiplexor	Not applicable

In this example:

- The block unpacks Signal-A (Standard signal) and Signal-D (Multiplexor signal) in every time step.
- If the value of Signal-D is 1 at a particular time step, then the block unpacks Signal-B along with Signal-A and Signal-D in that time step.
- If the value of Signal-D is 0 at a particular time step, then the block unpacks Signal-C along with Signal-A and Signal-D in that time step.
- If the value of Signal-D is not 1 or 0, the block does not unpack either of the Multiplexed signals in that time step.

Multiplex value

This option is available only if you have selected the **Multiplex type** to be `Multiplexed`. The value you provide here must match the `Multiplexor` signal value at run time for the block to unpack the `Multiplexed` signal. The **Multiplex value** must be a positive integer or zero.

Factor

Specify the **Factor** value applied to convert the unpacked raw value to the physical value (signal value). For more information, see the **Data input as** parameter conversion formula.

Offset

Specify the **Offset** value applied to convert the physical value (signal value) to the unpacked raw value. For more information, see the **Data input as** parameter conversion formula.

Min, **Max**

Define a range of raw signal values. The default settings are `-Inf` (negative infinity) and `Inf`, respectively. For **CANdb specified signals**, these settings are read from the CAN database. For **manually specified signals**, you can specify the minimum and maximum physical value of the signal. By default, these settings do not clip signal values that exceed them.

Programmatic Use
Block Parameter: `SignalInfo`

`Output identifier` — Enable CAN message identifier
`off` (default) | `on`

Select this option to output a CAN message identifier. The data type of this port is **uint32**.

Programmatic Use
Block Parameter: `IDPort`

`Output timestamp` — Enable CAN message timestamp
`off` (default) | `on`

Select this option to output the message timestamp. This value indicates when the message was received, measured as the number of seconds elapsed since the model simulation began. This option adds a new output port to the block. The data type of this port is **double**.

Programmatic Use
Block Parameter: `TimestampPort`

`Output error` — Enable CAN message error status
`off` (default) | `on`

Select this option to output the message error status. This option adds a new output port to the block. An output value of `1` on this port indicates that the incoming message is an error frame. If the output value is `0`, there is no error. The data type of this port is **uint8**.

Programmatic Use
Block Parameter: `ErrorPort`

`Output remote` — Enable CAN message remote frame status
`off` (default) | `on`

Select this option to output the message remote frame status. This option adds a new output port to the block. The data type of this port is **uint8**.

Programmatic Use
Block Parameter: `RemotePort`

`Output length` — Enable CAN message length
`off` (default) | `on`

Select this option to output the length of the message in bytes. This option adds a new output port to the block. The data type of this port is **uint8**.

Programmatic Use
Block Parameter: `LengthPort`

`Output status` — Enable status output
`off` (default) | `on`

Select this option to output the message received status. The status is `1` if the block receives new message and `0` if it does not. This option adds a new output port to the block. The data type of this port is **uint8**.

Programmatic Use
Block Parameter: `StatusPort`

`Output Bit Rate Switch (BRS)` — Enable BRS output
`off` (default) | `on`

(Disabled for CAN protocol.) Select this option to output the message bit rate switch. This option adds a new output port to the block. The data type of this port is **boolean**.

Programmatic Use
Block Parameter: `BRSPort`

`Output Error Status Indicator (ESI)` — Enable ESI output
`off` (default) | `on`

(Disabled for CAN protocol.) Select this option to output the message error status. This option adds a new output port to the block. The data type of this port is **boolean**.

Programmatic Use
Block Parameter: `ESIPort`

`Output Data Length Code (DLC)` — Enable DLC output
`off` (default) | `on`

(Disabled for CAN protocol.) Select this option to output the message data length. This option adds a new output port to the block. The data type of this port is **double**.

Programmatic Use
Block Parameter: `DLCPort`

See Also

Blocks

Topics
"Design Your Model for Effective Acceleration"
"Composite Signals"

Introduced in R2018a

Custom MATLAB file

Update active configuration parameters of parent model by using file containing custom MATLAB code

Library: Embedded Coder / Configuration Wizards

Description

When you add a Custom MATLAB file block to your Simulink model and double-click it, a custom MATLAB script executes and configures model parameters that are relevant to code generation. Set a block parameter to invoke the build process after configuring the model.

After you double-click the block, open the Configuration Parameters dialog box to see that the model parameter values have changed.

Use the example MATLAB script, `matlabroot/toolbox/coder/simulinkcoder_core/rtwsampleconfig.m` with the Custom MATLAB file block. Adapt the script to your model requirements. The block and the script provide a starting point for customization. For more information, see “Create a Custom Configuration Wizard Block”.

Note To provide a quick way to switch between configurations, you can include more than one Configuration Wizard block in your model.

Parameters

`Configure the model for` — Configuration objective

`Custom` (default) | `ERT (optimized for fixed-point)` | `ERT (optimized for floating-point)` | `GRT (optimized for fixed/floating-point)` | `GRT (debug for fixed/floating-point)`

Objective for updating the active configuration parameters of a parent model by using a predefined MATLAB script.

`Configuration function` — Custom MATLAB script

`rtwsampleconfig` (default)

Predefined or custom MATLAB script that updates the active configuration parameters of the parent model.

Dependencies

This parameter is only used with this Configuration Wizards block. To enable this parameter, set **Configure the model for** to `Custom`.

`Invoke build process after configuration` — Initiate build process

`off` (default) | `on`

If selected, the predefined script initiates the code generation and build process after updating the configuration parameters.

See Also

ERT (optimized for fixed-point) | ERT (optimized for floating-point) | GRT (debug for fixed/floating-point) | GRT (optimized for fixed/floating-point)

Topics

"Configure and Optimize Model with Configuration Wizard Blocks"

Introduced in R2011a

ERT (optimized for fixed-point)

Update active configuration parameters of parent model for ERT fixed-point code generation
Library: Embedded Coder / Configuration Wizards

Description

When you add an ERT (optimized for fixed-point) block to your Simulink model and double-click it, a predefined MATLAB script executes and optimally configures the model parameters for fixed-point code generation with the ERT target. Set a block parameter to invoke the build process after configuring the model.

After you double-click the block, open the Configuration Parameters dialog box to see that the model parameter values have changed.

Note To provide a quick way to switch between configurations, you can include more than one Configuration Wizard block in your model.

Parameters

`Configure the model for` — Configuration objective
`ERT (optimized for fixed-point)` (default) | `ERT (optimized for floating-point)` | `GRT (optimized for fixed/floating-point)` | `GRT (debug for fixed/floating-point)` | `Custom`

Objective for updating the active configuration parameters of a parent model by using a predefined MATLAB script.

`Configuration function` — Custom MATLAB script
`rtwsampleconfig` (default)

Predefined or custom MATLAB script that updates the active configuration parameters of the parent model.

Dependencies

This parameter is only used with the Custom MATLAB file block. To enable this parameter, set **Configure the model for** to `Custom`.

`Invoke build process after configuration` — Initiate build process
`off` (default) | `on`

If selected, the predefined script initiates the code generation and build process after updating the configuration parameters.

See Also

Custom MATLAB file | ERT (optimized for floating-point) | GRT (debug for fixed/floating-point) | GRT (optimized for fixed/floating-point)

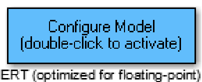
Topics

"Configure and Optimize Model with Configuration Wizard Blocks"

Introduced in R2006b

ERT (optimized for floating-point)

Update active configuration parameters of parent model for ERT floating-point code generation
Library: Embedded Coder / Configuration Wizards



Description

When you add an ERT (optimized for floating-point) block to your Simulink model and double-click it, a predefined MATLAB script executes and optimally configures the model parameters for floating-point code generation with the ERT target. Set a block parameter to invoke the build process after configuring the model.

After you double-click the block, open the Configuration Parameters dialog box to see that the model parameter values have changed.

Note To provide a quick way to switch between configurations, you can include more than one Configuration Wizard block in your model.

Parameters

`Configure the model for` — Configuration objective

`ERT (optimized for floating-point)` (default) | `ERT (optimized for fixed-point)` | `GRT (optimized for fixed/floating-point)` | `GRT (debug for fixed/floating-point)` | `Custom`

Objective for updating the active configuration parameters of a parent model by using a predefined MATLAB script.

`Configuration function` — Custom MATLAB script

`rtwsampleconfig` (default)

Predefined or custom MATLAB script that updates the active configuration parameters of the parent model.

Dependencies

This parameter is only used with the Custom MATLAB file block. To enable this parameter, set **Configure the model for** to `Custom`.

`Invoke build process after configuration` — Initiate build process

`off` (default) | `on`

If selected, the predefined script initiates the code generation and build process after updating the configuration parameters.

See Also

Custom MATLAB file | ERT (optimized for fixed-point) | GRT (debug for fixed/floating-point) | GRT (optimized for fixed/floating-point)

Topics

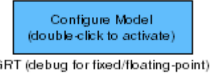
"Configure and Optimize Model with Configuration Wizard Blocks"

Introduced in R2006b

GRT (debug for fixed/floating-point)

Update active configuration parameters of parent model for GRT fixed-point or floating-point code generation

Library: Embedded Coder / Configuration Wizards



Description

When you add a GRT (debug for fixed/floating-point) block to your Simulink model and double-click it, a predefined MATLAB script executes and optimally configures the model parameters for fixed-point or floating-point code generation, with TLC debugging options enabled, with the GRT target. Set a block parameter to invoke the build process after configuring the model.

After you double-click the block, open the Configuration Parameters dialog box to see that the model parameter values have changed.

Note To provide a quick way to switch between configurations, you can include more than one Configuration Wizard block in your model.

Parameters

`Configure the model for` — Configuration objective
`GRT (debug for fixed/floating-point)` (default) | `ERT (optimized for fixed-point)` | `ERT (optimized for floating-point)` | `GRT (optimized for fixed/floating-point)` | `Custom`

Objective for updating the active configuration parameters of a parent model by using a predefined MATLAB script.

`Configuration function` — Custom MATLAB script
`rtwsampleconfig` (default)

Predefined or custom MATLAB script that updates the active configuration parameters of the parent model.

Dependencies

This parameter is only used with the Custom MATLAB file block. To enable this parameter, set **Configure the model for** to `Custom`.

`Invoke build process after configuration` — Initiate build process
`off` (default) | `on`

If selected, the predefined script initiates the code generation and build process after updating the configuration parameters.

See Also

Custom MATLAB file | ERT (optimized for fixed-point) | ERT (optimized for floating-point) | GRT (optimized for fixed/floating-point)

Topics

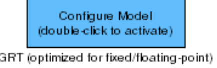
"Configure and Optimize Model with Configuration Wizard Blocks"

Introduced in R2006b

GRT (optimized for fixed/floating-point)

Update active configuration parameters of parent model for GRT fixed-point or floating-point code generation

Library: Embedded Coder / Configuration Wizards



Description

When you add a GRT(optimized for fixed-point) block to your Simulink model and double-click it, a predefined MATLAB script executes and optimally configures the model parameters for fixed-point or floating-point code generation with the GRT target. Set a block parameter to invoke the build process after configuring the model.

After you double-click the block, open the Configuration Parameters dialog box to see that the model parameter values have changed.

Note To provide a quick way to switch between configurations, you can include more than one Configuration Wizard block in your model.

Parameters

`Configure the model for` — Configuration objective

`GRT (optimized for fixed/floating-point)` (default) | `ERT (optimized for fixed-point)` | `ERT (optimized for floating-point)` | `GRT (debug for fixed/floating-point)` | `Custom`

Objective for updating the active configuration parameters of a parent model by using a predefined MATLAB script.

`Configuration function` — Custom MATLAB script

`rtwsampleconfig` (default)

Predefined or custom MATLAB script that updates the active configuration parameters of the parent model.

Dependencies

This parameter is only used with the Custom MATLAB file block. To enable this parameter, set **Configure the model for** to `Custom`.

`Invoke build process after configuration` — Initiate build process

`off` (default) | `on`

If selected, the predefined script initiates the code generation and build process after updating the configuration parameters.

See Also

Custom MATLAB file | ERT (optimized for fixed-point) | ERT (optimized for floating-point) | GRT (debug for fixed/floating-point)

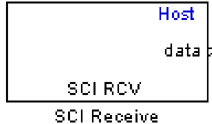
Topics

"Configure and Optimize Model with Configuration Wizard Blocks"

Introduced in R2006b

SCI Receive

Configure host-side serial communications interface to receive data from serial port
Library: Embedded Coder / Embedded Targets / Host Communication



Description

The SCI Receive block specifies the configuration of a data package that is received from a target computer by this block.

Ports

Output

`data` — Data package
scalar (default) | vector

Data package, specified as a scalar or vector, which is received from a target computer. The package can consist of headers, terminators, and data elements. The package size is limited to 16 bytes of ASCII characters, including headers and terminators. Calculate the size of a package by adding the byte sizes of headers, terminators, and the data.

This table lists the number of bytes for supported data types.

Data Type	Byte Count
`single`	4 bytes
`int8` and `uint8`	1 byte
`int16` and `uint16`	2 bytes
`int32` and`uint32`	4 bytes

For example, if your data package includes a 1-byte package header, `'S'`, and a 1-byte package terminator, `'E'`, 14 bytes remain for data. If your data is of type `int8`, the data can consist of up to 14 data elements. If your data is of type `uint16`, the data can consist of up to 7 data elements. If your data is of type `int32`, the data package can consist of up to 3 data elements with 2 bytes left over. Because you cannot mix data types in a package, the remaining 2 bytes are not used.

The number of data elements that can fit into a data package determine the data length (see the **Data length** parameter). In the preceding example, the 14 data elements of type `int8` and the 7 data elements of type `uint16` are the data lengths for each data package, respectively. When the data length exceeds 16 bytes, unexpected behavior, including run-time errors, can result.

Data Types: `single` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

`status` — Error status of data transmission
0 | 1 | 2 | 3 | 4

Optional error status value for data transmissions.

Possible status values are listed in this table.

Status value	Condition
0	No errors.
1	Connection timed out while block was waiting to receive data.
2	Checksum error. Received data contains an error.
3	SCI parity error. Character was received with a mismatch.
4	SCI framing error. Expected stop bit was found.

Port Dependencies

To enable this port, select parameter **Output receiving status**.

Parameters

`Port name` — Name of COM port

COM1 (default) | COM2 | COM3 | COM4

Specify the name of the COM port that you are configuring for this SCI Receive block. You can configure up to four COM ports for a model, one COM port for each SCI Receive block in the model.

`Additional package header` — Header of received data package

'S' (default) | 'ASCII value'

Specify the header of the received data package as an ASCII value in single quotes. The value can be text or a number in the range 0 to 255. The quotes are not received and are not included in the package byte count.

The header is not part of the data being received. Typically, the header marks the start of the data. The header that you specify must match the header specified for the corresponding target computer SCI Transmit block.

`Additional package terminator` — Terminator of received data package

'E' (default) | 'ASCII value'

Specify the terminator of the received data package as an ASCII value in single quotes. The value can be text or a number in the range 0 to 255. The quotes are not received and are not included in the package byte count.

The terminator is not part of the data being received. Typically, the terminator marks the end of the data. The terminator that you specify must match the terminator specified for the corresponding target computer SCI Transmit block.

`Data type` — Data type of data in received data package

uint8 (default) | single | int8 | int16 | uint16 | int32 | uint32

Specify the data type of the input port of the corresponding target computer SCI Transmit block. The data type and corresponding byte count are inherited from the input port.

`Data length` — Number of data elements of specified data type that block receives

1 (default) | vector

Specify the number of data elements of the specified data type that the block receives from the target computer SCI Transmit block. A value other than 1 is treated as a vector. The data length is inherited from the length of the target computer SCI Transmit block input data.

`Initial output` — Default value from SCI Receive block

0 (default) | scalar

Specify a value that the block outputs as the last value received when these conditions exist:

- Parameter **Action taken when connection times out** is set to `Output the last received value`.
- No data has been received.
- A connection times out.

Parameter Dependencies

To enable this parameter, set **Action taken when connection times out** to `Output the last received value`.

`Action Taken when connection times out` — Output value to write when connection times out

`Output the last received value` (default) | `Output custom value`

Specify the output value that the block writes when a connection times out. The block can write the last value received or a custom value.

Parameter Dependencies

If you specify `Output custom value`, use parameter **Output value when connection times out** to set the custom value.

`Output value when connection times out` — Output value to write when connection times out

0 (default) | scalar | vector

Specify the custom value to output when a connection times out.

Parameter Dependencies

To enable this parameter, set **Action taken when connection times out** to `Output custom value`.

`Sample time` — Frequency of calls to block

-1 (default) | scalar

Specify the frequency at which the scheduler calls the SCI Receive block in seconds. To achieve either of these conditions, set this parameter to `-1`:

- The block inherits the sample time setting of the model.
- The block executes asynchronously.

`Output receiving status` — Error status of data transmission

off (default) | on

Select this parameter to create a **status** block output port that provides the status of data transmissions.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

This block does not support code generation. The block is supported for simulations on a host development computer only.

See Also

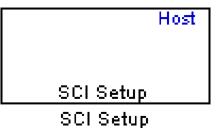
Host SCI Setup | Host SCI Transmit

Introduced in R2011a

SCI Setup

Configure host-side serial communications interface for host-side SCI Transmit and SCI Receive blocks

Library: Embedded Coder / Embedded Targets / Host Communication



Description

The SCI Setup block standardizes the serial communications interface (COM port) settings for use by the host-side SCI Transmit and SCI Receive blocks. This block is a standalone block that sets up one configuration for a COM port that SCI Transmit and SCI Receive blocks share. This block prevents conflicting configurations for the SCI Transmit and SCI Receive blocks. For example, the host-side SCI Transmit block cannot use COM1 with settings that differ from the COM1 settings used by the host-side SCI Receive block.

Parameters

`Communication Mode` — Communication mode to use for data transmissions
raw data (default) | protocol

Specify the communication mode to use for data transmissions. The communication mode can be `raw data` or `protocol`. Use `raw data` if you want the transmitting side to send unformatted data whenever the transmitting side is ready to send, whether the receiving side is ready or not. Without a wait state, deadlocks do not occur. Data transmission is asynchronous. With `raw data` mode, it is possible for the receiving side to miss data. If the data is noncritical, using `raw data` mode can avoid blocking processes.

When you specify `protocol` mode, handshaking between host and target computers occurs. The transmitting side sends `$SND`, indicating that it is ready to transmit. The receiving side sends back `$RDY`, indicating that it is ready to receive. The transmitting side then sends data and, when the transmission is completed, the receiving side sends a checksum.

Advantages to using `protocol` mode include:

- Data is received as expected (checksum).
- Data is received by the target computer.
- Time consistency; each side waits for its turn to send or receive.

Note Deadlocks can occur if an SCI Transmit block tries to communicate with more than one SCI Receive block on different COM ports when both SCI Receive blocks are blocking (use `protocol` mode). Deadlocks cannot occur on the same COM port.

`Baud rate` — Baud rate of COM port
115200 (default) | 57600 | 38400 | 19200 | 9600 | 4800 | 2400 | 1200 | 300 | 110

Specify the baud rate of the COM port.

`Number of stop bits` — Number of stop bits used by COM port
1 (default) | 2

Specify the number of stop bits that the COM port uses.

`Parity mode` — Parity mode used by COM port
none (default) | odd | even

Specify the parity mode that the COM port uses.

`Timeout` — Wait time for `protocol` communication mode
1.0 (default)

When you specify `protocol` for **Communication mode**, specify a value greater than or equal to 0, which indicates the number of seconds the transmitting side waits for an acknowledgement from the receiving side and how long the receiving side waits for data. The system displays a warning message each time a wait period exceeds the timeout value.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

This block does not support code generation. The block is supported for simulations on a host development computer only.

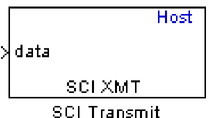
See Also

Host SCI Receive | Host SCI Transmit

Introduced in R2011a

SCI Transmit

Configure host-side serial communications interface to transmit data to serial port
Library: Embedded Coder / Embedded Targets / Host Communication



Description

The SCI Transmit block specifies the configuration of a data package being transmitted to a target computer from this block.

Ports

Input

`data` — Data package
scalar (default) | vector

Data package, specified as a scalar or vector, sent to a target computer. The package can consist of headers, terminators, and data elements. The package size is limited to 16 bytes of ASCII characters, including headers and terminators. Calculate the size of a package by adding the byte sizes of headers, terminators, and the data.

This table lists the number of bytes for supported data types.

Data Type	Byte Count
`single`	4 bytes
`int8` and `uint8`	1 byte
`int16` and `uint16`	2 bytes
`int32` and`uint32`	4 bytes

For example, if your data package includes a 1-byte package header, `'S'`, and a 1-byte package terminator, `'E'`, 14 bytes remain for data. If your data is of type `int8`, the data can consist of up to 14 data elements. If your data is of type `uint16`, the data can consist of up to 7 data elements. If your data is of type `int32`, the data package can consist of up to 3 data elements with 2 bytes left over. Because you cannot mix data types in a package, the remaining 2 bytes are not used.

The number of data elements that can fit into a data package determine the data length (see the **Data length** parameter). In the preceding example, the 14 data elements of type `int8` and the 7 data elements of type `uint16` are the data lengths for each data package, respectively. When the data length exceeds 16 bytes, unexpected behavior, including run-time errors, can result.

Data Types: `single` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

Parameters

`Port name` — Name of COM port
COM1 (default) | COM2 | COM3 | COM4

Specify the name of the COM port that you are configuring for this SCI Transmit block. You can configure up to four COM ports for a model, one COM port for each SCI Transmit block in the model.

`Additional package header` — Header of transmitted data package
'S' (default) | 'ASCII value'

Specify the header of the transmitted data package as an ASCII value in single quotes. The value can be text or a number in the range 0 to 255. The quotes are not transmitted and are not included in the package byte count.

The header is not part of the data being transmitted. Typically, the header marks the start of the data. The header that you specify must match the header specified for the corresponding target computer SCI Receive block.

`Additional package terminator` — Terminator of transmitted data package
'E' (default) | 'ASCII value'

Specify the terminator of the transmitted data package as an ASCII value in single quotes. The value can be text or a number in the range 0 to 255. The quotes are not transmitted and are not included in the package byte count.

The terminator is not part of the data being transmitted. Typically, the terminator marks the end of the data. The terminator that you specify must match the terminator specified for the corresponding target computer SCI Receive block.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

This block does not support code generation. The block is supported for simulations on a host development computer only.

See Also

Host SCI Receive | Host SCI Setup

Introduced in R2011a

Idle Task

Create free-running task

Description

The Idle Task block and the subsystem connected to it specify functions in the downstream subsystem to execute as background tasks. The tasks executed through the Idle Task block are of the lowest priority, lower than that of the base rate task.

This block is not supported on target computers that run an operating system.

Vectorized Output

Ports

Input

`Port_1` — Simulated interrupt
scalar | vector

Optional simulated interrupt for testing asynchronous interrupt behavior during Simulink simulation.

Port Dependencies

To enable this port, select parameter **Enable simulation input**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output

`Port_1` — Task identifiers and preemption flags
vectors

Numbers that identify the background tasks and task preemption flags, represented as vectors. A task identifier vector stores numbers that identify the background tasks.

For each background task, the preemption flag vector stores a flag that indicates whether the task can be preempted. Unless the preemption flag vector contains one element, the number of elements in the preemption flag vector must match the length of the task identifier vector. If the preemption flag vector has the same number of elements as the task identifier vector, each task represented in the task identifier vector has the preemption setting defined by the value of the corresponding element in the preemption flag vector. When the preemption flag vector contains one element, the flag setting applies to the entire task identifier vector.

A higher-priority task cannot preempt a lower-priority task that cannot be preempted.

Data Types: `uint8`

Parameters

`Task numbers` — Identifiers for tasks created
[1 2] (default) | vector of integers in the range 0 to 15

Specify task identifiers for functions that are in the downstream subsystem as a vector of integers in the range 0 to 15. The vector must contain the same number of values as the number of functions in the downstream subsystem. For example, the default vector `[1 2]` indicates that the downstream subsystem contains two functions.

The number of values that you enter corresponds to the number of functions in the downstream subsystem. The values that you specify determine the execution order of the functions. For example, the vector `[2 3 1]` indicates that:

- The subsystem contains three functions.
- The third function executes first.
- The first function executes second.
- The second function executes third.

After the functions execute, the Idle Task block cycles back and repeats the execution of the functions in the same order.

`Preemption flags` — Preemption flags for specified tasks

[0 1] (default) | vector of 0 to 16 integers in the range [0, 1]

Specify preemption flags for the tasks specified by the **Task numbers** parameter as a vector of up to 16 ones and zeros. Higher-priority interrupts can preempt interrupts that have a lower priority. To control preemption, use the preemption flags to specify whether an interrupt can be preempted.

The value `1` indicates that the interrupt for the corresponding task can be preempted. The value `0` indicates that the interrupt cannot be preempted. You have the option of specifying:

- One preemption value that applies to the entire vector that you specify for **Task numbers**.
- A preemption value for each task identified in the vector that you specify for **Task numbers**. Specify the flag values in the order that corresponds to the order of the tasks in **Task numbers**.

For example, if you specify the vector `[2 3 1]` for **Task numbers** and `[0]` for **Preemption flags**, tasks `1`, `2`, and `3` cannot be preempted. If you specify `[1 1 0]` for **Preemption flags**, tasks `2` and `3` can be preempted and task `1` cannot be preempted.

`Enable simulation input` — Input signal for simulation

off (default) | on

Select this parameter to create an input port that receives interrupt input for model simulation. Use the port to connect and test asynchronous interrupt processing behavior for one or more simulated interrupt sources during Simulink model simulation.

See Also

Introduced in R2011a

Memory Allocate

Allocate memory for new variable

Description

The Memory Allocate block, on C2xxx processors, directs the TI compiler to allocate a memory location for a new variable. Block parameters specify the variable name, the alignment of the variable in memory, the data type of the variable, and other features that fully define the memory required.

The block does not verify whether the parameter settings for the variable are valid, such as checking the variable name, data type, or section. You must check that the parameters settings are valid.

You do not connect the Memory Allocation block to other blocks in a model.

Parameters

Memory

Allocate memory for storing the variable. Specify the data type and size.

`Variable name` — Name for variable

myVariable (default) | string | character vector

Specify the name of the variable for which to allocate memory. The variable is allocated in the generated code.

`Specify variable alignment` — Variable alignment flag

off (default) | on

Select this parameter, if required by your target processor, to direct the compiler to align the new variable to a byte alignment boundary.

Parameter Dependencies

If you select this parameter, use parameter **Memory alignment boundary** to set the byte alignment boundary.

`Memory alignment boundary` — Memory alignment for variable

4 (default) | 1 | 2 | 8

Specify the alignment boundary for the variable data type in bytes. Alignment can occur on 1-, 2-, 4-, or 8-byte boundaries. If the variable contains multiple values, such as a vector or an array, the block aligns elements according to rules applied by the compiler.

Parameter Dependencies

To enable this parameter, select **Specify variable alignment**.

`Data type` — Data type for variable

uint32 (default) | double | single | int8 | uint8 | int16 | uint16 | int32 | int64 | uint64 | boolean

Specify the data type for the variable.

`Specify data type qualifier` — Data type qualifier flag
off (default) | on

Select this parameter to specify a data type qualifier to apply to the variable.

Parameter Dependencies

If you select this parameter, use parameter **Data type qualifier** to set the data type qualifier to apply to the variable.

`Data type qualifier` — Data type qualifier for variable
volatile (default) | sting | character vectory

Specify the data type qualifier to apply to the variable in generated code as a string or character vector. Common qualifiers are `volatile`, `const`, `static`, and `register`. The block does not check whether the value that you enter is a valid qualifier.

`Data dimension` — Number of elements of variable data type
64 (default) | positive integer

Specify the number of elements of the specified data type for the variable as a positive integer.

`Initialize memory` — Memory initialization flag
off (default) | on

Select this parameter to specify an initial value for the variable.

Parameter Dependencies

If you select this parameter, use parameter **Initial value** to set the initial value.

`Initial value` — Initial value for variable
0 (default) | scalar | vector | matrix

Specify the initial value for the variable. At run time, the block sets the memory location to this value.

Parameter Dependencies

To enable this parameter, select **Initialize memory**.

Section

Specify the memory section in which to allocate the variable.

`Specify memory section` — Memory section flag
off (default) | on

Select this parameter to specify a memory section to use for allocating space in memory for the variable.

Parameter Dependencies

If you select this parameter, use parameters **Memory section**, **Bind memory section**, **Section start address** to specify memory section details.

`Memory section` — Memory section for variable
mySEC1 (default) | string | character vector

Specify the name of the memory section to use for allocating memory for the variable as a string or character vector. Specify a standard memory section or a custom memory section that you declare elsewhere in your code.

Verify that the memory section has enough space to store the variable.

Parameter Dependencies

- To enable this parameter, select **Specify memory section**.
- To bind the specified memory section to a specific start address in memory, select **Bind memory section** and specify the address by entering a value for **Section start address**.

`Bind memory section` — Bind memory section to start address flag
off (default) | on

Select this parameter to bind a newly created memory section for the variable to a specific start address.

The new memory section specified for **Memory section** is defined when you select this parameter.

Parameter Dependencies

- Select this parameter to enable parameter **Section start address**.
- Do not select this parameter if you are associating the variable with an existing memory section.

`Section start address` — Start address of memory section for variable
hex2dec('8000') (default) | memory address in decimal or hexadecimal form

Specify the start address to which to bind the memory section for the variable in decimal form or in hexadecimal form with a conversion to decimal as shown by the default value `hex2dec('8000')`. The block does not verify the address. Verify that the address that you specify exists and that it can contain the specified memory section.

Parameter Dependencies

- Enable this parameter by selecting parameter **Bind memory section**.
- Do not specify a value for this parameter if you are associating the variable with an existing memory section.

See Also
Memory Copy

Introduced in R2011a

Memory Copy

Copy data from and to memory section

Description

Generated code for the Memory Copy block copies data from and to processor memory as configured by block parameters. When you use this block to copy an individual data element from a source to a destination, the block copies the element from the source, using the source data type, and then casts the data element to the specified destination data type.

Include as many instances of the Memory Copy block in a model as required to manipulate memory on a target processor. Each instance of the block works with one variable, address, or set of addresses provided to the block as input.

Specify the source and destination for a memory copy by using block parameters. You can use block parameters to control other aspects of a memory copy, such as:

- Initialization for memory locations
- Memory stride and offset during run time
- Write operations to memory during program initialization, during program termination, and at every sample time
- Insertion of custom ANSI C source code before and after each memory copy read and write operation (for example, to lock and unlock registers before and after accessing them)
- Quick direct memory access (QDMA) for processors and boards that support QDMA copy operations (C621x, C64xx, and C671x processor families)

The Memory Copy block performs operations at three periods during program execution:

- Initialization
- Real-time operations
- Termination

You can use block parameters to control when and how the block initializes memory, copies data or variables to and from memory, and terminates copy operations. The parameters enable you to turn on and off memory copy operations in the three periods independently.

Use the Memory Copy block and the Memory Allocate block to manipulate and allocate memory for custom device drivers, such as PCI bus drivers or codec-style drivers.

During simulation, the Memory Copy block does not perform an operation. The block output is not defined.

Ports

Input

`src` — Input data for copy operation

scalar | vector

The source data for the memory copy operation, specified as a scalar or vector.

Port Dependencies

To use this port as the source for the memory copy operation, set parameter **Copy from** to `Input port`.

Data Types: `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`&src` — Address of input data for copy operation
scalar | vector

The memory address of source data for the copy operation, specified as a scalar or vector.

Port Dependencies

To use this port as the source for the memory copy operation, set parameter **Copy from** to `Specified address` and **Specify address source** to `Input port`. The Copy Memory block converts input port `src` to `&src`.

Data Types: `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`src ofs` — Offset for data read during copy operation
scalar | vector

Offset to use for data read during the copy operation, specified as a scalar or vector.

Port Dependencies

To create this port, select parameter **Use offset when reading** and set **Specify offset source** to `Input port`.

Data Types: `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`&dst` — Address of output data for copy operation
scalar | vector

The memory address to use as the data destination for the copy operation, specified as a scalar or vector.

Port Dependencies

To use this port as the destination for the memory copy operation, set parameter **Copy to** to `Specified address` and **Specify address source** to `Input port`. The Copy Memory block converts output port `dst` to input port `&dst`.

Data Types: `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`dst ofs` — Offset for data written during copy operation
scalar | vector

Offset to use for data write during the copy operation, specified as a scalar or vector.

Port Dependencies

To create this port, select parameter **Use offset when writing** and set **Specify offset source** to `Input port`.

Data Types: `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output

`dst` — Output data for copy operation

scalar | vector

The data copied, specified as a scalar or vector.

Port Dependencies

To use this port as the destination for the memory copy operation, set parameter **Copy to** to `Output port`.

Data Types: `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Parameters

Source

Specify the source sequential memory location for the copy operation. Specify the data type, size, and other attributes of the source variable.

`Copy from` — Input source for copy operation

Input port (default) | Specified address | Specified source code symbol

Specify the input source for the data read part of the copy operation. Choose from the sources listed in this table.

Source of Data Read	Parameter Value to Specify
`src` input port	`Input port`
Memory address	`Specified address`
Symbol (variable) in source code lookup table	`Specified source code symbol`

Parameter Dependencies

- If you select `Specified address`, use **Specify address source** to specify the source of the memory address and **Address** to specify the address.
- If you select `Specified source code symbol`, use **Source code symbol** to specify the symbol (variable) in the source code symbol table to copy.
- If you select `Specified address` or `Specified source code symbol`, change **Data type** to a value other than `Inherit from source` (the default). If you do not make this change, you receive an error message indicating that the data type cannot be inherited because the input port does not exist.

`Specify address source` — Source of memory address for input data

Specify via dialog (default) | Input port

Specify the source of the memory address of the input source for the copy operation. To specify a memory address for the source variable, select `Specify via dialog`. That selection enables an **Address** parameter that you use to specify the memory address.

To specify that the block get the address from the input port, select `Input port`. When you select `Input port`, the block input port label changes to `&src`.

Parameter Dependencies

- To enable this parameter, set **Copy from** to `Specified address`.
- If you select `Specify via dialog`, this parameter enables the **Address** parameter, which you use to specify the address of the source variable.
- If you select `Specify via dialog`, set **Data type** to a value other than `Inherit from source` (the default). If you do not make this change, you receive an error message indicating that the data type cannot be inherited because the input port does not exist.
- If you select `Inport port` , specify a data type for the **Data type** parameter.

`Address` — Memory address of source data

hex2dec('00001000') (default) | memory address in decimal or hexadecimal form with a conversion to decimal

Specify the memory address of the source data in decimal form or in hexadecimal form with a conversion to decimal as shown by the default value `hex2dec('00001000')`.

This example converts `0x1000` to decimal form.

```
4096 = hex2dec('1000');
```

For this example, you can specify the address as `4096` or `hex2dec('1000')`.

Parameter Dependencies

To enable this parameter, set **Copy from** to `Specified address` and **Specify address source** to `Specify via dialog`.

`Source code symbol` — Symbol in source code symbol table

myVariableSrc (default) | string | character vector

Specify the symbol (variable) in the source code symbol table to copy. The symbol that you specify must exist in the symbol table for your program. The block does not verify whether the symbol exists in the symbol table and whether you specify the symbol with valid syntax. Enter text that specifies the symbol exactly as it appears in your code.

Parameter Dependencies

- To enable this parameter, set **Copy from** to `Specified source code symbol`.
- Set **Data type** to a value other than `Inherit from source` (the default). If you do not make this change, you receive an error message indicating that the data type cannot be inherited because the input port does not exist.

`Data type` — Data type of data being copied

uint8 (default) | double | single | int8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | boolean | Inherit from input port

Specify the data type of the source data being copied. To inherit the data type from the `src` input port, select `Inherit from input port`.

`Data length` — Number of data elements to copy

1 (default) | positive integer

Specify the number of elements to copy from the source location. Each element has the data type specified by the **Data type** parameter.

`Use offset when reading` — Use offset when reading input

off (default) | on

Specify whether the block uses an offset when reading input. The offset value is in elements of the specified data type. Specify the source of the offset by using the **Specify offset source** parameter.

Parameter Dependencies

- If you select this parameter, use **Specify offset source** to specify the source of the offset.
- Use **Offset** to specify the offset value.

`Specify offset source` — Source of input offset

Specify via dialog (default) | Input port

Specify the source of the input offset for the copy operation. To specify an offset value, select `Specify via dialog`. That selection enables an **Offset** parameter that you use to specify the offset value.

To specify for the block to get the offset from an input port, select `Input port`. When you select `Input port`, the block creates an input port labeled `src ofs` and reads the offset value from that port. The `src ofs` port enables your program to change the offset dynamically during program execution.

Parameter Dependencies

To enable this parameter, select **Use offset when reading**.

`Offset` — Number of values to skip before copying first value to destination

0 (default) | positive integer

Before copying the first value to the destination, specify the number of values to skip.

Parameter Dependencies

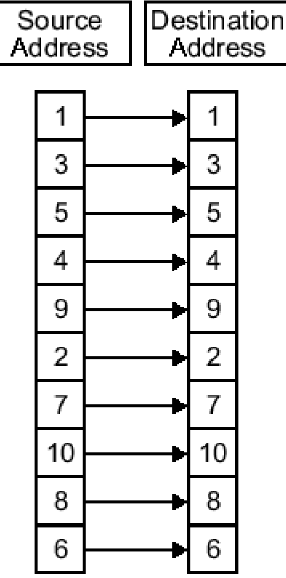
To enable this parameter, select **Use offset when reading** and set **Specify offset source** to `Specify via dialog`.

`Stride` — Spacing for reading input

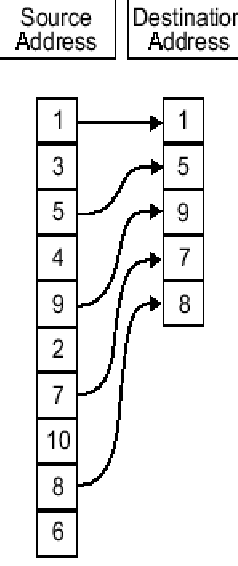
1 (default) | positive integer

Specify the spacing for reading the input. By default, the stride value is one, meaning that the generated code reads the input data sequentially. When you add a stride value that is not equal to one, when reading input data, the generated code skips spaces in the source address equal to the stride.

These figures show the stride concept. In the first figure, data is copied without a stride. The second figure shows the results of a stride value of two. You can specify a stride value for the block output with parameter **Stride** on the **Destination** tab. You can also compare stride with the offset to see the differences.



Input Stride = 1
Output Stride = 1
Number of Elements Copied = 10



Input Stride = 2
Output Stride = 1
Number of Elements Copied = 5

Destination

Specify the destination memory location for the copy operation. Specify the attributes of the destination.

`Copy to` — Type of output destination for copy operation

Output port (default) | Specified address | Specified destination code symbol

Specify the type of output destination for the copy operation. Select one of the values listed in this table.

Parameter Value	Destination of Data Write
`Output port`	Block `dst` output port
`Specified address`	Memory location specified by parameters **Specify address destination** and **Address**
`Specified source code symbol`	Symbol (variable) specified by parameter **Source code symbol**

Parameter Dependencies

- If you select `Specified address`, use **Specify address destination** to specify the destination memory location.
- If you select `Specified source code symbol`, use **Destination code symbol** to specify the symbol (variable) in the source code symbol table to which to copy the variable.

`Specify address source` — Source of memory address for output destination

Specify via dialog (default) | Input port |

Specify the source of the destination memory address of the variable for the copy operation. To specify a memory address for the variable, select `Specify via dialog`. That selection enables an **Address** parameter that you use to specify the memory address. To specify that the block get the address from an input port, select `Input port`. When you select `Input port`, the block creates an input port labeled &`dst`. Changing the address dynamically means that you can use the block to copy different variables by providing the variable address from an upstream block in the model.

Parameter Dependencies

- To enable this parameter, set **Copy to** to `Specified address`.
- If you select `Specify via dialog`, this parameter enables the **Address** parameter, which you use to specify the address of the destination variable.

`Address` — Memory address of destination variable

hex2dec('00002000') (default) | memory address in decimal or hexadecimal form with a conversion to decimal

Specify the memory address of the destination variable in decimal form or in hexadecimal form with a conversion to decimal as shown by the default value `hex2dec('00001000')`.

This example converts `0x2000` to decimal form.

```
8192 = hex2dec('2000');
```

For this example, you can specify the address as `8192` or `hex2dec('2000')`.

Parameter Dependencies

To enable this parameter, set **Copy to** to `Specified address`and **Specify address source** to `Specify via dialog`.

`Source code symbol` — Symbol in source code symbol table
myVariableDst (default) | string | character vector

Specify the symbol (variable) in the source code symbol table to which to copy the variable. The symbol that you specify, must exist in the symbol table for your program. The block does not verify whether the symbol exists in the symbol table and whether you specify the symbol with valid syntax. Enter text that specifies the symbol exactly as it appears in your code.

Parameter Dependencies

To enable this parameter, set **Copy to** to `Specified source code symbol`.

`Data type` — Data type of variable
uint32 (default) | double | single | int8 | uint8 | int16 | uint16 | uint32 | int64 | uint64 | boolean | Inherit from source

Specify the data type of the source variable. To inherit the data type from the source variable, select `Inherit from source`.

`Use offset when writing` — Use offset when writing output
off (default) | on

Specify whether the block uses an offset when writing output. The offset value is in elements of the specified data type. Specify the source of the offset by using the **Specify offset source** parameter.

Parameter Dependencies

If you select this parameter, use **Specify offset source** to specify the source of the offset. Use **Offset** to specify the offset value.

`Specify offset source` — Source of offset for output destination
Specify constant value (default) | Specify source code symbol

Specify the source of the offset for the output destination for the copy operation. To specify an offset value for the destination variable, select `Specify via dialog`. That selection enables an **Offset** parameter that you use to specify the offset value.

To specify that the block get the offset from the input port, select `Input port`. When you select `Input port`, the block adds an input port labeled `dst ofs` and reads the offset value from that port. The `dst ofs` port enables your program to change the offset dynamically during execution.

Parameter Dependencies

To enable this parameter, select **Specify offset source**.

`Offset` — Number of values to skip before writing first value to destination
0 (default) | positive integer

Before writing the first value to the destination, specify the number of values to skip.

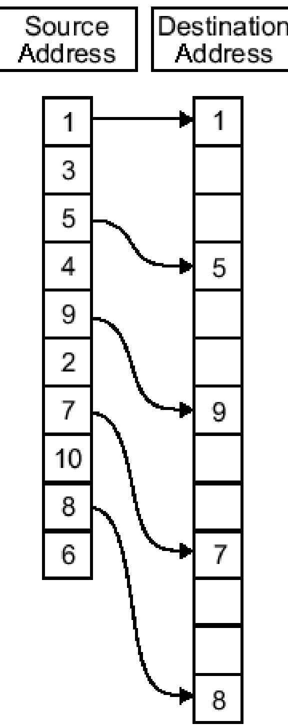
Parameter Dependencies

To enable this parameter, select **Use offset when writing** and set **Specify offset source** to `Specify via dialog`.

`Stride` — Spacing for writing output
1 (default) | positive integer

Specify the spacing for writing the output. By default, the stride value is one meaning that the generated code writes the input data sequentially to the destination in consecutive locations. When you add a stride value that is not equal to one, when writing input data, the generated code skips spaces in the destination address equal to the stride.

This figure shows a stride value of three applied to writing the input to an output location. You can specify a stride value for the input with parameter **Stride** on the **Source** pane. As shown in the figure, you can use an input stride and an output stride at the same time to enable manipulating memory more fully.



Input Stride = 2
Output Stride = 3
Number of Elements Copied = 5

`Sample time` — Rate of memory copy
inf (default) | scalar

Specify the rate at which the memory copy operation occurs in seconds. To use a constant sample time, specify `Inf`. To inherit the sample time from the input port or, when the block does not have an input port, from the Simulink model, specify `-1`.

Options

Configure parameters that control the copy process.

`Set memory value at initialization` — Set memory address during initialization
off (default) | on

Specify whether to initialize the memory address to a specific value during program initialization.

Parameter Dependencies

If you select this parameter, use a combination of parameters to configure the initialization value.

What to Configure	Parameter
Source of the initialization value	**Specify initialization value source**
Initialization value as a constant	**Initialization value (constant)**
Initialization value as a variable	**Initialization value (source code symbol)**
Initialization value as a mask to manipulate register contents at the bit level	**Apply initialization value as mask**
Apply a mask value	**Bitwise operator**

`Specify initialization value source` — Source of initialization value

Specify constant value (default) | Specify source code symbol

Specify the source of the initial value. To configure the source for initializing memory as a specific value, select `Specify constant value`. To configure the source as a variable (a symbol), select `Specify source code symbol`.

Parameter Dependencies

- To enable this parameter, select **Set memory value at initialization**.
- Use **Initialization value (constant)** or **Initialization value (source code symbol)** to specify the initial value.

`Initialization value (constant)` — Constant initialization value

1 (default) | scalar

Specify a constant value.

Parameter Dependencies

To enable this parameter, select **Set memory value at initialization** and set **Set initialization value source** to `Specify constant value`.

`Initialization value (source code symbol)` — Symbol in source code symbol table

myInitValueVariable (default) | string | character vector

Specify the symbol (variable) in the source code symbol table to use for the initialization value. The symbol that you specify must exist in the symbol table for your program. The block does not verify whether the symbol exists in the symbol table and whether you specify the symbol with valid syntax. Enter text that specifies the symbol exactly as it appears in your code.

Parameter Dependencies

To enable this parameter, select **Set memory value at initialization** and set **Set initialization value source** to `Specify source code symbol`.

`Apply initialization value as mask` — Apply initialization value as mask

off (default) | on

Specify whether to use the initialization value as a mask to manipulate register content at the bit level. Your initialization value is treated as a string of bits for the mask.

To define how to apply the mask value, specify a value for the **Bitwise operator** parameter.

To use your initialization value as a mask, the output from the copy must be a specific address. The output:

- Can be a symbol
- Cannot be an output port

Parameter Dependencies

If you select this parameter, use **Bitwise operator** to define how to apply the mask value.

`Bitwise operator` — Type of bitwise operation

bitwise AND (default) | bitwise OR | bitwise exclusive OR | left shift | right shift

Specify the type of bitwise operation to use as a mask to manipulate the memory value. Applying a mask to the copy process means that you can select individual bits in the result. For example, by applying a mask, you can read the value of the fifth bit.

Select one of the bitwise operations in this table.

Bitwise Operation	Description
`bitwise AND`	Apply the mask value as a bitwise AND to the value in the register.
`bitwise OR`	Apply the mask value as a bitwise OR to the value in the register.
`bitwise exclusive OR`	Apply the mask value as a bitwise exclusive OR to the value in the register.
`left shift`	Shift the bits in the register left by the number of bits represented by the initialization value. For example, if your initialization value is 3, the block shifts the register value to the left 3 bits. In this case, the value must be a positive integer.
`right shift`	Shift the bits in the register to the right by the number of bits represented by the initialization value. For example, if your initialization value is 6, the block shifts the register value to the right 6 bits. In this case, the value must be a positive integer.

Parameter Dependencies

To enable this parameter, select **Apply initialization value as mask**.

`Set memory value at termination` — Copy memory during program termination

off (default) | on

Specify that your program copy memory during program termination. Copying a value in memory during termination occurs in addition to a copy during program initialization.

Parameter Dependencies

If you select this parameter, you can use **Set memory value only at initialization/termination** to limit copy operations to occur during program initialization and termination only.

`Termination value` — Termination value

1 (default) | scalar | vector | matrix

Specify a value to write to memory during program termination.

Parameter Dependencies

To enable this parameter, select **Set memory value at termination**.

`Set memory value only at initialization/termination` — Copy memory value during program initialization and termination only
off (default) | on

Specify whether to perform copies during program initialization and termination only. When this parameter is cleared, the block performs copies during initialization, real-time operations, and termination. If you select this parameter, the block performs copies during initialization and termination only.

`Insert custom code before memory write` — Insert custom code before memory write
off (default) | on

Specify whether the code generator inserts custom ANSI C code immediately before the program writes to the specified memory location. You can use this parameter and **Insert custom code after memory write** to lock and unlock registers before and after accessing them. For example, some processors have registers that you might need to unlock and lock with EALLOW and EDIS macros before and after your program accesses them.

Parameter Dependencies

If you select this parameter, use **Custom code** to specify the custom ANSI C code to insert into the generated code immediately before the memory write operation.

`Insert custom code after memory write` — Custom code after memory write flag
off (default) | on

Specify whether the code generator inserts custom ANSI C code immediately after the program writes to the specified memory location. You can the **Insert custom code before memory write** and this parameter to lock and unlock registers before and after accessing them. For example, some processors have registers that you might need to unlock and lock with EALLOW and EDIS macros before and after your program accesses them.

Parameter Dependencies

If you select this parameter, use **Custom code** to specify the custom ANSI C code to insert into the generated code immediately after the memory write operation.

`Custom code` —
/* Custom Code Before Write*/ or /* Custom Code After Write*/ (default) | string | character vector

Specify custom ANSI C code to insert into the generated code immediately before or immediately after the memory write operation. Code that you specify appears in the generated code exactly as you enter it.

Parameter Dependencies

To enable this parameter, select **Insert custom code before memory write** or **Insert custom code after memory write**.

`Use QDMA for copy (if available)` — Use quick direct memory access (QDMA) for copy
off (default) | on

Specify whether to enable quick direct memory access (QDMA) for processors that support QDMA.

Parameter Dependencies

If you select this parameter:

- Source and destination data types must match. If the data types do not match, the copy operation returns an error.
- Input and output stride values must be set to `1`.
- If you select this parameter, use **Enable blocking model** to specify whether memory copy operations are blocking processes.

`Enable blocking mode` — Enable blocking mode

on (default) | off

Specify whether memory copy operations that use QDMA are blocking processes. Select this parameter to enable the memory copy operations blocking processes. When you enable blocking mode, other program processing waits until the memory copy operation is complete.

Parameter Dependencies

To enable this parameter, select **Use QDMA for copy (if available)**.

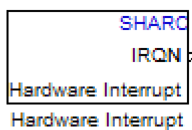
See Also

Memory Allocate

Introduced in R2011a

SHARC Hardware Interrupt

Generate Interrupt Service Routine



Library

Embedded Coder/ Embedded Targets/ Processors/ Analog Devices® SHARC®/ Scheduling

Description

Create interrupt service routines (ISR) in the software generated by the build process. When you incorporate this block in your model, code generation results in ISRs on the processor that either run the processes that are downstream from this block or trigger an Idle Task block connected to this block.

Parameters

Interrupt numbers

Specify an array of interrupt numbers for the interrupts to install. The valid ranges are 8-36 and 38-40.

The width of the block output signal corresponds to the number of interrupt numbers specified in this field. The values in this field and the preemption flag entries in **Preemption flags: preemptible-1, non-preemptible-0** define how the code and processor handle interrupts during asynchronous scheduler operations.

Simulink task priorities

Each output of the Hardware Interrupt block drives a downstream block (for example, a function call subsystem). Simulink model task priority specifies the priority of the downstream blocks. Specify an array of priorities corresponding to the interrupt numbers entered in **Interrupt numbers**.

Code generation requires rate transition code. Refer to Rate Transitions and Asynchronous Blocks. The task priority values facilitate absolute time integrity when the asynchronous task must obtain real time from its base rate or its caller. Typically, assign priorities for these asynchronous tasks that are higher than the priorities assigned to periodic tasks.

Preemption flags preemptible – 1, non-preemptible – 0

Higher-priority interrupts can preempt interrupts that have lower priority. To allow you to control preemption, use the preemption flags to specify whether an interrupt can be preempted.

- Entering `1` indicates that the interrupt can be preempted.
- Entering `0` indicates the interrupt cannot be preempted.

When **Interrupt numbers** contains more than one interrupt value, you can assign different preemption flags to each interrupt by entering a vector of flag values to correspond to the order of the interrupts in **Interrupt numbers**. If **Interrupt numbers** contains more than one interrupt, and you enter only one flag value in this field, that status applies to all interrupts.

In the default settings `[0 1]`, the interrupt with priority `18` in **Interrupt numbers** is not preemptible and the priority `39` interrupt can be preempted.

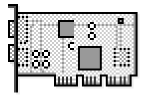
Enable simulation input

When you select this option, Simulink software adds an input port to the Hardware Interrupt block. This port is used in simulation only. Connect one or more simulated interrupt sources to the simulation input.

Introduced in R2011a

Target Preferences (Removed)

Configure model for specific IDE, tool chain, board, and processor



Library

Simulink Coder / Desktop Targets

Embedded Coder/ Embedded Targets

Description

The Target Preferences block has been removed from the Simulink block libraries. The contents of the Target Preferences block have been moved to the **Hardware Implementation** pane, located in the Configuration Parameters dialog. For more information, see:

- “Hardware configuration relocation from Target Preferences block to Configuration Parameters dialog box”
- “Configure Target Hardware Resources”
- “Hardware Implementation Pane”

Introduced in R2013a

UDP Receive

Receive UDP packet

Library: Embedded Coder / Embedded Targets / Host Communication

Description

The UDP Receive block receives UDP packets from an IP network port and saves them to its buffer. With each sample, the block outputs the contents of a single UDP packet as a data vector. The local IP port number on which the block receives the UDP packets is tunable in the generated code.

The generated code for this block relies on prebuilt `.dll` files. You can run this code outside the MATLAB environment or redeploy it, but you must account for the extra `.dll` files. The `packNGo` function creates a ZIP file that contains the pieces required to run or rebuild this code. For more details, see "How To Run a Generated Executable Outside MATLAB" (DSP System Toolbox).

Ports

Output

`Port_1` — UDP packet
vector

UDP packet, specified as a data vector, which is received from an IP network port.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `enumerated` | `bus`

Parameters

`Local IP port` — Number of IP port
25000 (default) | scalar in the range [1, 65535]

Specify the IP port number on which to receive UDP packets. This parameter is tunable in the generated code but is not tunable during simulation.

On Linux®, to set the IP port number below 1024, run MATLAB with root privileges. For example, at the Linux command line, enter:

```
sudo matlab
```

`Remote IP address ('0.0.0.0' to accept all)` — IP address from which to accept packets
'0.0.0.0' (default) | IP address

Specify the IP address from which to accept UDP packets. Specify a specific IP address to block UDP packets from other addresses. To accept packets from any IP address, specify `'0.0.0.0'`.

`Receive buffer size (bytes)` — Size of buffer that receives UDP packets

8192 (default) | scalar in the range [1, 67108864]

Specify the size of the buffer, in bytes, that receives the UDP packets. Make the buffer large enough to avoid data loss caused by buffer overflows.

`Maximum length for Message` — Maximum length of output data

255 (default) | scalar

Specify the maximum length, in vector elements, of the data output vector. Set this parameter to a value equal or greater than the data size of a UDP packet. The system truncates data that exceeds this length.

Parameter Dependencies

If you disable **Output variable-size signal**, the block output is the length specified by this parameter.

`Data type for Message` — Data type of message

uint8 (default) | single | double | int8 | int16 | int32 | int64 | uint16 | uint32 | uint64 | boolean | fixed point | enumerated | bus

Specify the data type of the vector elements in the message output. Match the data type to the data input used to create the UDP packets.

`Message is complex` — Message data complexity

off (default) | on

Specify whether the block receives a message as complex data. Select this parameter to receive a message as complex data. Clear this parameter if a received message is real data.

`Output variable-size signal` — Message output that varies in length

on (default) | off

Specify whether your model supports signals of varying length. If your model supports signals of varying length, select this parameter. In that case:

- The output vector varies in length, depending on the amount of data in the UDP packet.
- The block emits the data vector from a single unlabeled output.

If your model does not support signals of varying length, clear this parameter. In that case:

- The block emits a fixed-length output that is the same length as specified by **Maximum length for Message**.
- If the UDP packet contains less data than the fixed-length output, the difference contains invalid data.
- The block emits the data vector from the **Message** output.
- The block emits the length of the valid data from the **Length** output.
- The **Data type for Length** parameter is enabled.

Parameter Dependencies

If you disable this parameter, the block emits output that is the length specified by **Maximum length for Message**.

`Blocking time (seconds)` — Number of seconds to wait for UDP packet
inf (default) | scalar

For each sample, specify the number of seconds to wait for a UDP packet before returning control to the scheduler. To wait indefinitely, specify `inf`.

Note This parameter applies to the Embedded Coder UDP Receive block only.

`Sample time (seconds)` — Frequency of calls to block
0.01 (default) | scalar

Specify the frequency at which the scheduler calls the UDP Receive block, in seconds. Enter a value greater than zero. In real-time operation, setting this parameter to a smaller value reduces the likelihood of dropped UDP messages.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Byte Pack | Byte Reversal | Byte Unpack | UDP Send

Introduced in R2011a

UDP Send

Send UDP packet

Library: Embedded Coder / Embedded Targets / Host Communication

Description

The UDP Send block transmits an input data vector as a UDP packet to a remote IP network port. The remote IP port number to which the block sends the UDP packets is tunable in the generated code.

Some Simulink blocks and `.exe` files built from models that contain those blocks require shared libraries, such as `.dll` files on Windows. The UDP Send block requires the `networkdevice.dll` library file. To meet this requirement, follow the example on the `packNGo` function page to package the code files for your model. The resulting compressed folder contains the `.dll` files that the model requires, including `networkdevice.dll`. To run this type of `.exe` file outside of a MATLAB environment, place the required `.dll` files in the same folder as the `.exe` file or place them in a folder on the Windows system path. For more details, see "How To Run a Generated Executable Outside MATLAB" (DSP System Toolbox).

Ports

Input

`Port_1` — UDP packet
scalar (default) | vector

UDP packet, specified as a data vector, which is transmitted to an IP network port.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `enumerated` | `bus`

Parameters

`Remote IP address ('255.255.255.255' for broadcast)` — IP address from which to accept UDP packets
'255.255.255.255' (default) | IP address | string

Specify the IP address or hostname to which to send UDP packets. If you specify a hostname, specify it as a string. To broadcast a UDP packet, specify `'255.255.255.255'`.

`Remote IP port` — Number of remote IP port
25000 (default) | scalar in the range [1, 65535]

Specify the IP port number to which to send UDP packets. This parameter is tunable in the generated code but is not tunable during simulation.

On Linux, to set the IP port number below 1024, run MATLAB with root privileges. For example, at the Linux command line, enter:

```
sudo matlab
```

`Local IP port source` — Source of local IP port
`Automatically determine` (default) | `Specify via dialog`

Specify whether the block uses a local port number that the system assigns or that you specify by using the **Local IP port** parameter. To let the system assign the port number, select `Automatically determine`. If the receiving address expects UDP packets from a specific port number, select `Specify via dialog` and specify the port number by using the **Local IP port** parameter.

Parameter Dependencies

To enable the **Local IP port** parameter, select `Specify via dialog`.

`Local IP port` — Number of local IP port
-1 (default) | scalar in the range [1, 65535]

Specify the IP port number from which the block sends UDP packets. Use this parameter when the receiving address expects messages from a specific port number.

Parameter Dependencies

To enable this parameter, set **Local IP port source** to `Specify via dialog`.

`Send buffer size (bytes)` — Size of buffer that sends UDP packets
8192 (default) | scalar in the range [1, 67108864]

Specify the size of the buffer, in bytes, that sends the UDP packets. Make the buffer large enough to avoid data loss caused by buffer overflows.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
Byte Pack | Byte Reversal | Byte Unpack | UDP Receive

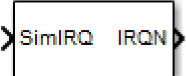
Introduced in R2011a

Simulink Coder Blocks

Async Interrupt

Generate Versa Module Eurocard (VME) interrupt service routines (ISRs) that execute downstream subsystems or Task Sync blocks

Library: Simulink Coder / Asynchronous / Interrupt Templates



Description

For each specified VME interrupt level in the example RTOS (VxWorks®), the Async Interrupt block generates an interrupt service routine (ISR) that calls one of the following:

- A function call subsystem
- A Task Sync block
- A Stateflow chart configured for a function call input event

---

Note You can use the blocks in the `vxlib1` library (Async Interrupt and Task Sync) for simulation and code generation. These blocks provide starting point examples to help you develop custom blocks for your target environment.

Assumptions and Limitations

- The block supports VME interrupts 1 through 7.
- The block uses these RTOS (VxWorks) system calls:

```
sysIntEnable
sysIntDisable
intConnect
intLock
intUnlock
tickGet
```

Performance Considerations

Execution of large subsystems at interrupt level can have a significant impact on interrupt response time for interrupts of equal and lower priority in the system. Usually, it is best to keep ISRs as short as possible. Connect only function-call subsystems that contain a few blocks to an Async Interrupt block.

A better solution for large subsystems is using the Task Sync block to synchronize the execution of the function-call subsystem to an RTOS task. Place the Task Sync block between the Async Interrupt block and the function-call subsystem. The Async Interrupt block then uses the Task Sync block as the ISR. The ISR releases a synchronization semaphore (performs a `semGive`) to the task, and returns immediately from interrupt level. The example RTOS (VxWorks) then schedules and runs the task. See the description of the Task Sync block.

Ports

Input

`Input` — Simulated interrupt source
scalar

A simulated interrupt source.

Output Arguments

`Output` — Control signal
scalar

Control signal for a:

- Function-call subsystem
- Task Sync block
- Stateflow chart configured for a function call input event

Parameters

`VME interrupt number(s)` — VME interrupt numbers for the interrupts to be installed
[1 2] (default) | integer array

An array of VME interrupt numbers for the interrupts to be installed. The valid range is `1..7`.

The width of the Async Interrupt block output signal corresponds to the number of VME interrupt numbers specified.

Note A model can contain more than one Async Interrupt block. However, if you use more than one Async Interrupt block, do not duplicate the VME interrupt numbers specified in each block.

`VME interrupt vector offset(s)` — Interrupt vector offset numbers corresponding to the VME interrupt numbers
[192 193] (default) | integer array

An array of unique interrupt vector offset numbers corresponding to the VME interrupt numbers entered for parameter **VME interrupt number(s)**. The Stateflow software passes the offsets to the RTOS (VxWorks) call `intConnect(INUM_TO_IVEC(offset),...)`.

`Simulink task priority(s)` — Priority of downstream blocks
[10 11] (default) | integer array

The Simulink priority of downstream blocks. Each output of the Async Interrupt block drives a downstream block (for example, a function-call subsystem). Specify an array of priorities corresponding to the VME interrupt numbers that you specify for parameter **VME interrupt number(s)**.

Parameter **Simulink task priority** values are required to generate a rate transition code (see “Rate Transitions and Asynchronous Blocks”). Simulink task priority values are also required to maintain

absolute time integrity when the asynchronous task must obtain real time from its base rate or its caller. The assigned priorities typically are higher than the priorities assigned to periodic tasks.

Note The Simulink software does not simulate asynchronous task behavior. The task priority of an asynchronous task is for code generation purposes only and is not honored during simulation.

`Preemption flag(s); preemptable-1; non-preemptable-0` — Selects preemption
[0 1] (default) | integer array

Set this option to 1 if an output signal of the Async Interrupt block drives a Task Sync block.

Higher priority interrupts can preempt lower priority interrupts in the example RTOS (VxWorks). To lock out interrupts during the execution of an ISR, set the pre-emption flag to 0. This setting causes generation of `intLock()` and `intUnlock()` calls at the beginning and end of the ISR code. Use interrupt locking carefully, as it increases the interrupt response time of the system for interrupts at the `intLockLevelSet()` level and below. Specify an array of flags corresponding to the VME interrupt numbers entered for parameter **VME interrupt number(s)**.

Note The number of elements in the arrays specifying parameters **VME interrupt vector offset(s)** and **Simulink task priority** must match the number of elements in the array specified for parameter **VME interrupt number(s)**.

`Manage own timer` — Select timer manager
on (default) | off

If selected, the ISR generated by the Async Interrupt block manages its own timer by reading absolute time from the hardware timer. Specify the size of the hardware timer with parameter **Timer size**.

`Timer resolution (seconds)` — Resolution of ISR timer
1/60 (default)

The resolution of the ISRs timer. ISRs generated by the Async Interrupt block maintain their own absolute time counters. By default, these timers obtain their values from the RTOS (VxWorks) kernel by using the `tickGet` call. Parameter **Timer resolution** determines the resolution of these counters. The default resolution is `1/60` second. The `tickGet` resolution for your board support package (BSP) can be different. Determine the `tickGet` resolution for your BSP and enter it for parameter **Timer resolution**.

If you are targeting an RTOS other than the example RTOS (VxWorks), replace the `tickGet` call with an equivalent call to the target RTOS. Or, generate code to read the timer register on the target hardware. For more information, see “Timers in Asynchronous Tasks” and “Async Interrupt Block Implementation”.

`Timer size` — Number of bits to store the clock tick
`32bits` (default) | `16bits` | `8bits` | `auto`

The number of bits to store the clock tick for a hardware timer. The ISR generated by the Async Interrupt block uses the timer size when you select parameter **Manage own timer**. The size can be `32bits` (the default), `16bits`, `8bits`, or `auto`. If you select `auto`, the code generator determines

the timer size based on the settings of parameteres **Application lifespan (days)** and **Timer resolution**.

By default, timer values are stored as 32-bit integers. When parameter **Timer size** is set to `auto`, you can indirectly control the word size of the counters by setting parameter **Application lifespan (days)**. If you set **Application lifespan (days)** to a value that is too large for the code generator to handle as a 32-bit integer of the specified resolution, the code generator uses a second 32-bit integer to address overflows.

For more information, see “Control Memory Allocation for Time Counters”. See also “Timers in Asynchronous Tasks”.

`Enable simulation input` — Select add simulation input port
on (default) | off

If selected, the Simulink software adds an input port to the Async Interrupt block. This port is for simulation only. Connect one or more simulated interrupt sources to the simulation input.

Note Before generating code, consider removing blocks that drive the simulation input to prevent the blocks from contributing to the generated code. Alternatively, you can use the Environment Controller block, as explained in “Dual-Model Approach: Code Generation”. If you use the Environment Controller block, the sample times of driving blocks contribute to the sample times supported in the generated code.

See Also
Task Sync

Topics
“Asynchronous Events”
“Asynchronous Events”

Introduced in R2006a

Asynchronous Task Specification

Specify priority of asynchronous task represented by referenced model triggered by asynchronous interrupt

Library: Simulink Coder / Asynchronous

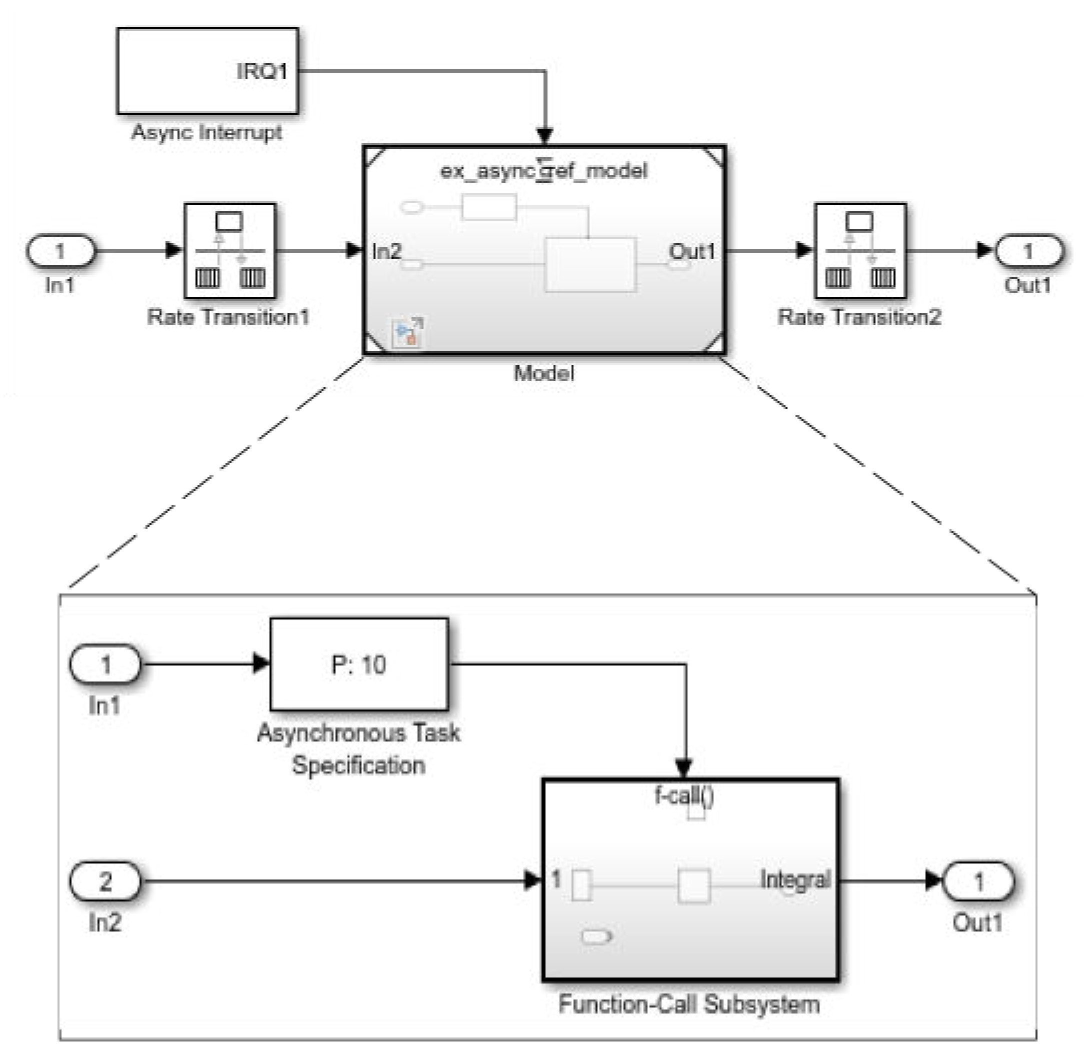
Description

The Asynchronous Task Specification block specifies parameters, such as the task priority, of an asynchronous task represented by a function-call subsystem with a trigger from an asynchronous interrupt. Use this block to control scheduling of function-call subsystems with triggers from asynchronous events. You control the scheduling by assigning a priority to each function-call subsystem within a referenced model.

To use this block, follow the procedure in “Convert an Asynchronous Subsystem into a Model Reference”.

Observe in the figure:

- The block must reside in a referenced model between a root-level Inport block and a function-call subsystem. The Asynchronous Task Specification block must immediately follow and connect directly to the Inport block.
- The Inport block must receive an interrupt signal from an Async Interrupt block that is in the parent model.
- The Inport block must be configured to receive and send function-call trigger signals.



Ports

Input

`Port_1` — Interrupt input signal
scalar

Interrupt input signal received from a root-level Inport block.

Output

`Port_1` — Interrupt signal with priority
scalar

Interrupt signal with specified task priority that triggers a function-call subsystem.

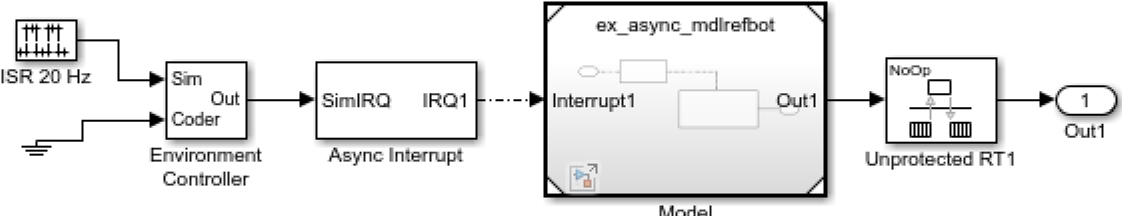
Parameters

`Task priority` — Priority of asynchronous task that calls function-call subsystem
`10` (default)

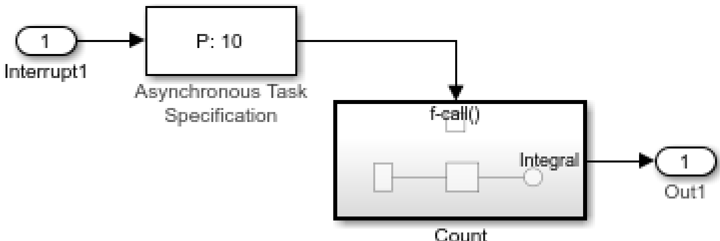
Specify an integer or [] as the priority of the asynchronous task that calls the connected function-call subsystem. The priority must be a value that generates relevant rate transition behaviors.

- If you specify an integer, it must match the priority value of the interrupt signal initiator in the parent model.
- If you specify [], the priority does not have to match the priority of the interrupt signal initiator in the top model. The rate transition algorithm is conservative (not optimized). The priority is unknown but static.

Consider the following model.



The referenced model has the following content.



If the **Task priority** parameter is set to 10, the Async Interrupt block in the parent model must also have a priority of 10. If the parameter is set to [], the priority of the Async Interrupt block can be a value other than 10.

See Also

Blocks
Function-Call Subsystem | Inport

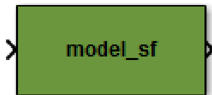
Topics
"Asynchronous Events"
"Spawn and Synchronize Execution of RTOS Task"
"Pass Asynchronous Events in RTOS as Input To a Referenced Model"
"Convert an Asynchronous Subsystem into a Model Reference"
"Rate Transitions and Asynchronous Blocks"
"Asynchronous Support"

“Asynchronous Events”
“Model References”

Introduced in R2011a

Generated S-Function

Represent model or subsystem as generated S-function code
Library: Simulink Coder / S-Function Target



Description

An instance of the Generated S-Function block represents code that the code generator produces from its S-function system target file for a model or subsystem. For example, you extract a subsystem from a model and build a Generated S-Function block from it by using the S-function target. This mechanism can be useful for:

- Converting models and subsystems to application components
- Reusing models and subsystems
- Optimizing simulation—often, an S-function simulates more efficiently than the original model

For details on how to create a Generated S-Function block from a subsystem, see “Create S-Function Blocks from a Subsystem”.

Requirements

- The S-Function block must perform identically to the model or subsystem from which it was generated.
- Before creating the block, explicitly specify Inport block signal attributes, such as signal widths or sample times. The sole exception to this rule concerns sample times, as described in “Sample Time Propagation in Generated S-Functions”.
- Set the solver parameters of the Generated S-Function block to be the same as the parameters of the original model or subsystem. The generated S-function code operates identically to the original subsystem (for an exception to this rule, see “Choose a Solver Type”).

Ports

Input

`Input` — S-function input
varies

See requirements.

Output Arguments

`Output` — S-function output
varies

See requirements.

Parameters

`Generated S-function name (model_sf)` — Name of S-function
model_sf (default) | character vector

The name of the generated S-function. The code generator derives the name by appending `_sf` to the name of the model or subsystem from which the block is generated.

`Show module list` — Select display module list
off (default) | on

If selected, displays modules generated for the S-function.

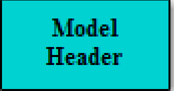
See Also

Topics
"Generate S-Function from Subsystem"
"Create S-Function Blocks from a Subsystem"

Introduced in R2011b

Model Header

Specify external header code



Description

For a model that includes the Model Header block, the code generator adds external code that you specify to the header file (`model.h`) that it generates. You can specify code for the code generator to add near the top and bottom of the header file.

If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

Parameters

`Top of Model Header` — Code to add near top of generated header file

Specify code that you want the code generator to add near the top of the header file for the model. The code generator places the code in the section labeled `user code (top of header file)`.

`Bottom of Model Header` — Code to add at bottom of generated header file

Specify code that you want the code generator to add at the bottom of the header file for the model. The code generator places the code in the section labeled `user code (bottom of header file)`.

See Also

Model Source | System Disable | System Outputs | System Update | System Derivatives | System Enable | System Initialize | System Start | System Terminate

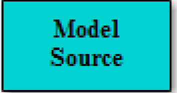
Topics

"Place External C/C++ Code in Generated Code"

Introduced in R2006a

Model Source

Specify external source code



Description

For a model that includes the Model Source block, the code generator adds external code that you specify to the source file (`model.c` or `model.cpp`) that it generates. You can specify code for the code generator to add near the top and bottom of the source file.

If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

Parameters

`Top of Model Source` — Code to add near top of generated source file

Specify code that you want the code generator to add near the top of the source file for the model. The code generator places the code in the section labeled `user code (top of source file)`.

`Bottom of Model Source` — Code to add at bottom of generated source file

Specify code that you want the code generator to add at the bottom of the source file for the model. The code generator places the code in the section labeled `user code (bottom of source file)`.

Example

See "Add External Code to Generated Start Function".

See Also

Model Header | System Disable | System Outputs | System Update | System Derivatives | System Enable | System Initialize | System Start | System Terminate

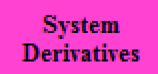
Topics

"Place External C/C++ Code in Generated Code"

Introduced in R2006a

System Derivatives

Specify external system derivative code



Description

For a model or nonvirtual subsystem that includes the System Derivatives block and a block that computes continuous states, the code generator adds external code, which you specify, to the `SystemDerivatives` function that it generates. You can specify code for the code generator to add to the declaration, execution, and exit sections of the function code.

Note If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

Parameters

`System Derivatives Function Declaration Code` — Code to add to the declaration section of the generated function

Specify code that you want the code generator to add to the declaration section of the `SystemDerivatives` function for the model or subsystem.

`System Derivatives Function Execution Code` — Code to add to the execution section of the generated function

Specify code that you want the code generator to add to the execution section of the `SystemDerivatives` function for the model or subsystem.

`System Derivatives Function Exit Code` — Code to add to the exit section of the generated function

Specify code that you want the code generator to add to the exit section of the `SystemDerivatives` function for the model or subsystem.

See Also

Model Header | Model Source | System Initialize | System Disable | System Enable | System Outputs | System Start | System Terminate | System Update

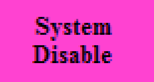
Topics

"Place External C/C++ Code in Generated Code"

Introduced in R2006a

System Disable

Specify external system disable code



Description

For a model or nonvirtual subsystem that includes the System Disable block and a block that uses a `SystemDisable` function, the code generator adds external code, which you specify, to the `SystemDisable` function that it generates. You can specify code for the code generator to add to the declaration, execution, and exit sections of the function code.

Note If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

Parameters

`System Disable Function Declaration Code` — Code to add to the declaration section of the generated function

Specify code that you want the code generator to add to the declaration section of the `SystemDisable` function for the model or subsystem.

`System Disable Function Execution Code` — Code to add to the execution section of the generated function

Specify code that you want the code generator to add to the execution section of the `SystemDisable` function for the model or subsystem.

`System Disable Function Exit Code` — Code to add to the exit section of the generated function

Specify code that you want the code generator to add to the exit section of the `SystemDisable` function for the model or subsystem.

See Also

Model Header | Model Source | System Initialize | System Derivatives | System Enable | System Outputs | System Start | System Terminate | System Update

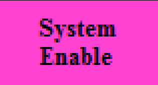
Topics

"Place External C/C++ Code in Generated Code"

Introduced in R2006a

System Enable

Specify external system enable code



Description

For a model or nonvirtual subsystem that includes the System Enable block and a block that uses a `SystemEnable` function, the code generator adds external code, which you specify, to the `SystemEnable` function that it generates. You can specify code for the code generator to add to the declaration, execution, and exit sections of the function code.

Note If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

Parameters

`System Enable Function Declaration Code` — Code to add to the declaration section of the generated function

Specify code that you want the code generator to add to the declaration section of the `SystemEnable` function for the model or subsystem.

`System Enable Function Execution Code` — Code to add to the execution section of the generated function

Specify code that you want the code generator to add to the execution section of the `SystemEnable` function for the model or subsystem.

`System Enable Function Exit Code` — Code to add to the exit section of the generated function

Specify code that you want the code generator to add to the exit section of the `SystemEnable` function for the model or subsystem.

See Also

Model Header | Model Source | System Initialize | System Derivatives | System Disable | System Outputs | System Start | System Terminate | System Update

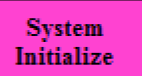
Topics

"Place External C/C++ Code in Generated Code"

Introduced in R2006a

System Initialize

Specify external system initialization code



Description

For a model or nonvirtual subsystem that includes the System Initialize block and a block that uses a `SystemInitialize` function, the code generator adds external code, which you specify, to the `SystemInitialize` function that it generates. You can specify code for the code generator to add to the declaration, execution, and exit sections of the function code.

Note If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

Parameters

`System Initialize Function Declaration Code` — Code to add to the declaration section of the generated function

Specify code that you want the code generator to add to the declaration section of the `SystemInitialize` function for the model or subsystem.

`System Initialize Function Execution Code` — Code to add to the execution section of the generated function

Specify code that you want the code generator to add to the execution section of the `SystemInitialize` function for the model or subsystem.

`System Initialize Function Exit Code` — Code to add to the exit section of the generated function

Specify code that you want the code generator to add to the exit section of the `SystemInitialize` function for the model or subsystem.

See Also

Model Header | Model Source | System Enable | System Derivatives | System Disable | System Outputs | System Start | System Terminate | System Update

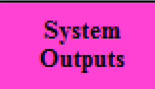
Topics

"Place External C/C++ Code in Generated Code"

Introduced in R2006a

System Outputs

Specify external system outputs code



Description

For a model or nonvirtual subsystem that includes the System Outputs block and a block that uses a `SystemOutputs` function, the code generator adds external code, which you specify, to the `SystemOutputs` function that it generates. You can specify code for the code generator to add to the declaration, execution, and exit sections of the function code.

Note If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

Parameters

`System Outputs Function Declaration Code` — Code to add to the declaration section of the generated function

Specify code that you want the code generator to add to the declaration section of the `SystemOutputs` function for the model or subsystem.

`System Outputs Function Execution Code` — Code to add to the execution section of the generated function

Specify code that you want the code generator to add to the execution section of the `SystemOutputs` function for the model or subsystem.

`System Outputs Function Exit Code` — Code to add to the exit section of the generated function

Specify code that you want the code generator to add to the exit section of the `SystemOutputs` function for the model or subsystem.

See Also

Model Header | Model Source | System Enable | System Derivatives | System Disable | System Initialize | System Start | System Terminate | System Update

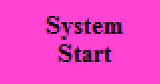
Topics

"Place External C/C++ Code in Generated Code"

Introduced in R2006a

System Start

Specify external system startup code



Description

For a model or nonvirtual subsystem that includes the System Start block and a block that uses a `SystemStart` function, the code generator adds external code, which you specify, to the `SystemStart` function that it generates. You can specify code for the code generator to add to the declaration, execution, and exit sections of the function code.

Note If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

Parameters

`System Start Function Declaration Code` — Code to add to the declaration section of the generated function

Specify code that you want the code generator to add to the declaration section of the `SystemStart` function for the model or subsystem.

`System Start Function Execution Code` — Code to add to the execution section of the generated function

Specify code that you want the code generator to add to the execution section of the `SystemStart` function for the model or subsystem.

`System Start Function Exit Code` — Code to add to the exit section of the generated function

Specify code that you want the code generator to add to the exit section of the `SystemStart` function for the model or subsystem.

See Also

Model Header | Model Source | System Enable | System Terminate | System Derivatives | System Disable | System Initialize | System Outputs | System Update

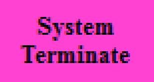
Topics

"Place External C/C++ Code in Generated Code"

Introduced in R2006a

System Terminate

Specify external system termination code



Description

For a model or nonvirtual subsystem that includes the System Terminate block and a block that uses a `SystemTerminate` function, the code generator adds external code, which you specify, to the `SystemTerminate` function that it generates. You can specify code for the code generator to add to the declaration, execution, and exit sections of the function code.

Note If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

Parameters

`System Terminate Function Declaration Code` — Code to add to the declaration section of the generated function

Specify code that you want the code generator to add to the declaration section of the `SystemTerminate` function for the model or subsystem.

`System Disable Terminate Execution Code` — Code to add to the execution section of the generated function

Specify code that you want the code generator to add to the execution section of the `SystemTerminate` function for the model or subsystem.

`System Disable Terminate Exit Code` — Code to add to the exit section of the generated function

Specify code that you want the code generator to add to the exit section of the `SystemTerminate` function for the model or subsystem.

See Also

Model Header | Model Source | System Enable | System Start | System Derivatives | System Disable | System Initialize | System Outputs | System Update

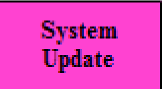
Topics

"Place External C/C++ Code in Generated Code"

Introduced in R2006a

System Update

Specify external system update code



Description

For a model or nonvirtual subsystem that includes the System Update block and a block that uses a `SystemUpdate` function, the code generator adds external code, which you specify, to the `SystemUpdate` function that it generates. You can specify code for the code generator to add to the declaration, execution, and exit sections of the function code.

Note If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

Parameters

`System Update Function Declaration Code` — Code to add to the declaration section of the generated function

Specify code that you want the code generator to add to the declaration section of the `SystemUpdate` function for the model or subsystem.

`System Update Function Execution Code` — Code to add to the execution section of the generated function

Specify code that you want the code generator to add to the execution section of the `SystemUpdate` function for the model or subsystem.

`System Update Function Exit Code` — Code to add to the exit section of the generated function

Specify code that you want the code generator to add to the exit section of the `SystemUpdate` function for the model or subsystem.

See Also

Model Header | Model Source | System Enable | System Start | System Derivatives | System Disable | System Initialize | System Outputs | System Terminate

Topics

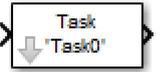
"Place External C/C++ Code in Generated Code"

Introduced in R2006a

Task Sync

Run code of downstream function-call subsystem or Stateflow chart by spawning an example RTOS (VxWorks) task

Library: Simulink Coder / Asynchronous / Interrupt Templates



Description

The Task Sync block spawns an example RTOS (VxWorks) task that calls a function-call subsystem or Stateflow chart. Typically, you place the Task Sync block between an Async Interrupt block and a function-call subsystem block or Stateflow chart. Alternatively, you could connect the Task Sync block to the output port of a Stateflow diagram that has an event, `Output to Simulink`, configured as a function call.

The Task Sync block:

- Uses the RTOS (VxWorks) system call `taskSpawn` to spawn an independent task. When the task is activated, it calls the downstream function-call subsystem code or Stateflow chart. The block calls `taskDelete` to delete the task during model termination.
- Creates a semaphore to synchronize the connected subsystem with execution of the block.
- Wraps the spawned task in an infinite `for` loop. In the loop, the spawned task listens for the semaphore by using `semTake`. The first call to `semTake` specifies `NO_WAIT`. This setting lets the task determine whether a second `semGive` has occurred before the completion of the function-call subsystem or chart. This sequence indicates that the interrupt rate is too fast or the task priority is too low.
- Generates synchronization code (for example, `semGive()`). This code lets the spawned task run. The task in turn calls the connected function-call subsystem code. The synchronization code can run at interrupt level. The connection between the Async Interrupt and Task Sync blocks accomplishes this operation and triggers execution of the Task Sync block within an ISR.
- Supplies absolute time if blocks in the downstream algorithmic code require it. The time comes from the timer maintained by the Async Interrupt block or comes from an independent timer maintained by the task associated with the Task Sync block.

When you design your application, consider when timer and signal input values could be taken for the downstream function-call subsystem that is connected to the Task Sync block. By default, the time and input data are read when the RTOS (VxWorks) activates the task. For this case, the data (input and time) are synchronized to the task itself. If you select the **Synchronize the data transfer of this task with the caller task** option and the Task Sync block driver is an Async Interrupt block, the time and input data are read when the interrupt occurs (that is, within the ISR). For this case, data is synchronized with the caller of the Task Sync block.

Note You can use the blocks in the `vxlib1` library (Async Interrupt and Task Sync) for simulation and code generation. These blocks provide starting point examples to help you develop custom blocks for your target environment.

Ports

Input

`Input` — Call from interrupt block
call

A call from an Async Interrupt block.

Output Arguments

`Output` — Call to function-call subsystem
call

A call to a function-call subsystem.

Parameters

`Task name (10 characters or less)` — Task function name
`Task0` (default) | character vector

The first argument passed to the `taskSpawn` system call in the RTOS. The RTOS (VxWorks) uses this name as the task function name. This name also serves as a debugging aid. Routines use the task name to identify the task from which they are called.

`Simulink task priority (0–255)` — RTOS task priority
50 (default) | integer

The RTOS task priority assigned to the function-call subsystem task when spawned. RTOS (VxWorks) priorities range from 0 to 255, with 0 representing the highest priority.

Note The Simulink software does not simulate asynchronous task behavior. The task priority of an asynchronous task is for code generation purposes only and is not honored during simulation.

`Stack size (bytes)` — Maximum size for stack of the task
1024 (default) | integer

Maximum size to which the stack of the task can grow. The stack size is allocated when the RTOS (VxWorks) spawns the task. Choose a stack size based on the number of local variables in the task. Determine the size by examining the generated code for the task (and functions that are called from the generated code).

`Synchronize the data transfer of this task with the caller task` — Select synchronization
off (default) | on

If not selected (the default),

- The block maintains a timer that provides absolute time values required by the computations of downstream blocks. The timer is independent of the timer maintained by the Async Interrupt block that calls the Task Sync block.

- A **Timer resolution** option appears.
- The **Timer size** option specifies the word size of the time counter.

If selected,

- The block does not maintain an independent timer and does not display the **Timer resolution** field.
- Downstream blocks that require timers use the timer maintained by the Async Interrupt block that calls the Task Sync block (see “Timers in Asynchronous Tasks”). The timer value is read at the time the asynchronous interrupt is serviced. Data transfers to blocks called by the Task Sync block execute within the task associated with the Async Interrupt block. Therefore, data transfers are synchronized with the caller.

`Timer resolution (seconds)` — Resolution for timer of the block
1/60 (default)

The resolution of the timer of the block in seconds. This option appears only if **Synchronize the data transfer of this task with the caller task** is not selected. By default, the block gets the timer value by calling the `tickGet` function in the RTOS (VxWorks). The default resolution is 1/60 second.

`Timer size` — Number of bits to store clock tick
`32bits` (default) | `16bits` | `8bits` | `auto`

The number of bits to store the clock tick for a hardware timer. The size can be `32bits` (the default), `16bits`, `8bits`, or `auto`. If you select `auto`, the code generator determines the timer size based on the settings of **Application lifespan (days)** and **Timer resolution**.

By default, timer values are stored as 32-bit integers. When **Timer size** is `auto`, you can indirectly control the word size of the counters by setting the **Application lifespan (days)** option. If you set **Application lifespan (days)** to a value that is too large for the code generator to handle as a 32-bit integer of the specified resolution, it uses a second 32-bit integer to address overflows.

For more information, see “Control Memory Allocation for Time Counters”. See also “Timers in Asynchronous Tasks”.

See Also
Async Interrupt

Topics
“Asynchronous Events”

Introduced in R2006a

Embedded Coder Parameters: Advanced Parameters

Use only existing shared code

Description

Check whether build process requires shared code that is not present in the existing shared code folder.

Category: Diagnostics

Settings

Default: `none`

`none`

Simulink software takes no action.

`warning`

Simulink software displays a warning.

`error`

Simulink software terminates the simulation and displays an error message.

Dependency

You can specify this parameter only if you specify a folder in the **Existing shared code** field. Otherwise the field appears dimmed.

Command-Line Information

Parameter: `UseOnlyExistingSharedCode`
Type: character vector
Value: `'none'` | `'warning'` | `'error'`
Default: `'none'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Cross-Release Shared Utility Code Reuse”
- “Cross-Release Code Integration”

Use Embedded Coder Features

Description

Enable "Embedded Coder" features for models deployed to "Simulink Supported Hardware".

Note If you enable this parameter in a model where Embedded Coder is not installed or available in the environment, a question dialog box prompts you to update the model to build without Embedded Coder features.

Category: Hardware Implementation

Settings

Default: `On`

On

Enable advanced Embedded Coder configuration parameters.

Note Enabling `Use Embedded Coder Features` also enables the "Use Simulink Coder Features" parameter.

Off

Disable advanced Embedded Coder configuration parameters.

Dependencies

This parameter requires an Embedded Coder license.

Command-Line Information

Parameter: `UseEmbeddedCoderFeatures`
Value: `'on'` or `'off'`
Default: `'on'`

See Also

Related Examples

- "Hardware Implementation Pane"

Feature set for selected hardware board

Select to use features from either the SoC Blockset™ support package or a Simulink Coder or Embedded Coder support package. Features, including properties and blocks, for the selected hardware board in SoC Blockset cannot be used in the Simulink Coder or Embedded Coder hardware support package.

Note This selection only appears when the SoC Blockset and a Simulink Coder or Embedded Coder support package for the same hardware board are both installed.

Settings

Default: `SoC Blockset`, `Simulink or Embedded Coder Hardware Support Package`

Remove reset function

Description

Remove unreachable (dead-code) instances of the `reset` functions from the generated code for ERT-based systems that include model referencing hierarchies. If you enable this parameter, Simulink checks that live code will be removed and errors if it finds such code.

Category: Code Generation > Interface

Settings

Default: `On`

On

Remove unreachable instances of the `reset` functions from the generated code for ERT-based systems that include model referencing hierarchies.

Off

Generate code without removing unreachable instances of the `reset` function.

Dependencies

This parameter requires an Embedded Coder license.

To set the **Remove reset function** parameter, set **Configuration Parameters** > **Code Generation** > **System target file** parameter to an ERT-based system target file, such as `ert.tlc`.

Command-Line Information

Parameter: `RemoveResetFunc`
Value: `'on'` or `'off'`
Default: `'on'`

See Also

Related Examples

- “Remove disable function” on page 5-6
- “Model Configuration Parameters: Code Generation Interface”

Remove disable function

Description

Remove unreachable (dead-code) instances of the `disable` functions from the generated code for ERT-based systems that include model referencing hierarchies. If you enable this parameter, Simulink checks that live code will be removed and errors if it finds such code.

Category: Code Generation > Interface

Settings

Default: `Off`

On

Remove unreachable instances of the `disable` functions from the generated code for ERT-based systems that include model referencing hierarchies.

Off

Generate code without removing unreachable instances of the `disable` function.

Dependencies

This parameter requires an Embedded Coder license.

To set the **Remove disable function** parameter, set **Configuration Parameters > Code Generation > System target file** parameter to an ERT-based system target file, such as `ert.tlc`.

Command-Line Information

Parameter: `RemoveDisableFunc`
Value: `'on'` or `'off'`
Default: `'off'`

See Also

Related Examples

- “Remove reset function” on page 5-5
- “Model Configuration Parameters: Code Generation Interface”

Code Generation Parameters: AUTOSAR

Model Configuration Parameters: Code Generation AUTOSAR

The **Code Generation > AUTOSAR Code Generation Options** category includes parameters for controlling AUTOSAR code generation. Based on your system target file selection, `autosar.tlc` or `autosar_adaptive.tlc`, you see code generation parameters for the AUTOSAR Classic Platform or the AUTOSAR Adaptive Platform.

Classic Platform Parameter	Description
"Generate XML file for schema version" on page 6-4	Select the AUTOSAR Classic Platform schema version to use when generating XML files.
"Maximum SHORT-NAME length" on page 6-6	Specify maximum length for SHORT-NAME XML elements for the AUTOSAR Classic Platform.
"Use AUTOSAR compiler abstraction macros" on page 6-7	Specify use of AUTOSAR macros to abstract compiler directives.
"Support root-level matrix I/O using one-dimensional arrays" on page 6-8	Allow root-level matrix I/O for column-major array layout.

Adaptive Platform Parameter	Description
"Generate XML file for schema version" on page 6-9	Select the AUTOSAR Adaptive Platform schema version to use when generating XML files.
"Maximum SHORT-NAME length" on page 6-10	Specify maximum length for SHORT-NAME XML elements for the AUTOSAR Adaptive Platform.
"Transport layer" on page 6-11	Select transport layer for XCP Slave communication.
"IP address" on page 6-12	Specify IP address for XCP communication.
"Port" on page 6-13	Specify network port for XCP communication.
"Verbose" on page 6-14	Enable verbose XCP Slave messages.
"Use custom XCP Slave" on page 6-15	Specify whether to use custom or default XCP Slave.

See Also

More About

- "Code Generation" (AUTOSAR Blockset)
- "Code Generation" (AUTOSAR Blockset)
- "Model Configuration"

Code Generation: AUTOSAR Code Generation Options Tab Overview

Parameters for controlling AUTOSAR code generation.

Configuration

This pane appears only if you specify the `autosar.tlc` or `autosar_adaptive.tlc` system target file.

To get help on an option

1. Right-click the option text label.
2. From the context menu, select **What's This**.

Tip

To configure other AUTOSAR options, use the AUTOSAR code perspective. From the **Apps** tab, open the AUTOSAR Component Designer app.

See Also

Related Examples

- "Model Configuration Parameters: Code Generation AUTOSAR" on page 6-2

Generate XML file for schema version

Description

Select the AUTOSAR Classic Platform schema version to use when generating XML files

Category: Code Generation > AUTOSAR Code Generation Options

Settings

Default: `4.3`

`4.4`
: Use schema version 4.4 (revision 4.4.0)

`4.3`
: Use schema version 4.3 (revision 4.3.1)

`4.2`
: Use schema version 4.2 (revision 4.2.2)

`4.1`
: Use schema version 4.1 (revision 4.1.3)

`4.0`
: Use schema version 4.0 (revision 4.0.3)

Tips

- Selecting the AUTOSAR target for your model for the first time sets the schema version parameter to the default value, `4.3`.
- When you import ARXML code into Simulink, the ARXML importer detects the schema version and sets the schema version parameter in the model. For a list of AUTOSAR schema revisions supported for ARXML import, see "Select AUTOSAR Schema" (AUTOSAR Blockset).
- Set the same value for top and referenced models.
- To configure other AUTOSAR options, use the AUTOSAR code perspective. From the **Apps** tab, open the AUTOSAR Component Designer app.

Command-Line Information

Parameter: `AutosarSchemaVersion`
Type: character vector
Value: `'4.4'` | `'4.3'` | `'4.2'` | `'4.1'` | `'4.0'`
Default: `'4.3'`

See Also

Related Examples

- "Model Configuration Parameters: Code Generation AUTOSAR" on page 6-2

- “Select AUTOSAR Schema” (AUTOSAR Blockset)
- “Code Generation” (AUTOSAR Blockset)

Maximum SHORT-NAME length

Description

Specify maximum length for SHORT-NAME XML elements for the AUTOSAR Classic Platform

Category: Code Generation > AUTOSAR Code Generation Options

Settings

Default: 128

The AUTOSAR standard specifies that the maximum length of `SHORT-NAME` XML elements is 128 characters. To configure a maximum length for `SHORT-NAME` elements exported by the code generator, specify an integer value between 32 and 128, inclusive.

Tip

Set the same value for top and referenced models.

Command-Line Information

Parameter: `AutosarMaxShortNameLength`
Type: integer
Value: integer value between 32 and 128, inclusive
Default: 128

See Also

Related Examples

- “Model Configuration Parameters: Code Generation AUTOSAR” on page 6-2
- “Specify Maximum SHORT-NAME Length” (AUTOSAR Blockset)

Use AUTOSAR compiler abstraction macros

Description

Specify use of AUTOSAR macros to abstract compiler directives

Category: Code Generation > AUTOSAR Code Generation Options

Settings

Default: Off

On

Software generates code with C macros that are abstracted compiler directives (near/far memory calls)

Off

Software generates code that does *not* contain AUTOSAR compiler abstraction macros.

Tip

Set the same value for top and referenced models.

Command-Line Information

Parameter: `AutosarCompilerAbstraction`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

Related Examples

- “Model Configuration Parameters: Code Generation AUTOSAR” on page 6-2
- “Configure AUTOSAR Compiler Abstraction Macros” (AUTOSAR Blockset)

Support root-level matrix I/O using one-dimensional arrays

Description

Allow root-level matrix I/O for column-major array layout

Category: Code Generation > AUTOSAR Code Generation Options

Settings

Default: Off

On

For AUTOSAR component models that use column-major array layout, software supports matrix I/O at the root level by generating code that implements matrices as one-dimensional arrays.

Off

For column-major array layout, software does not allow matrix I/O at the root level. If you try to build a model that has matrix I/O at the root level, the software produces an error.

Tips

- Set the same value for top and referenced models.
- For an AUTOSAR component model with multidimensional arrays, if you set the model configuration parameter **Array layout** to `Row-major`, you can preserve dimensions of multidimensional arrays in the generated C code. Preserving array dimensions in the generated code can enhance code integration.

  When **Array layout** is set to `Row-major`, **Support root-level matrix I/O using one-dimensional arrays** does not apply.

Command-Line Information

Parameter: `AutosarMatrixIOAsArray`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

Related Examples

- “Model Configuration Parameters: Code Generation AUTOSAR” on page 6-2
- “Root-Level Matrix I/O” (AUTOSAR Blockset)

Generate XML file for schema version

Description

Select the AUTOSAR Adaptive Platform schema version to use when generating XML files

Category: Code Generation > AUTOSAR Code Generation Options

Settings

Default: `00048 (R19-11)`

`00048 (R19-11)`

Use schema version 00048 (Release 19-11)

`00047 (R19-03)`

Use schema version 00047 (Release 19-03)

`00046 (R18-10)`

Use schema version 00046 (Release 18-10)

Tips

- Selecting the AUTOSAR adaptive target for your model for the first time sets the schema version parameter to the default value `00048 (R19-11)`.
- When you import ARXML code into Simulink, the ARXML importer detects the schema version and sets the schema version parameter in the model.
- Set the same value for top and referenced models.
- To configure other AUTOSAR options, use the AUTOSAR code perspective. From the **Apps** tab, open the AUTOSAR Component Designer app.

Command-Line Information

Parameter: `AutosarSchemaVersion`
Type: character vector
Value: `'00048'` | `'00047'` | `'00046'`
Default: `'00048'`

See Also

Related Examples

- “Model Configuration Parameters: Code Generation AUTOSAR” on page 6-2
- “Select AUTOSAR Schema” (AUTOSAR Blockset)
- “Code Generation” (AUTOSAR Blockset)

Maximum SHORT-NAME length

Description

Specify maximum length for SHORT-NAME XML elements for the AUTOSAR Adaptive Platform

Category: Code Generation > AUTOSAR Code Generation Options

Settings

Default: 128

The AUTOSAR standard specifies that the maximum length of `SHORT-NAME` XML elements is 128 characters. To configure a maximum length for `SHORT-NAME` elements exported by the code generator, specify an integer value between 32 and 128, inclusive.

Tip

Set the same value for top and referenced models.

Command-Line Information

Parameter: `AutosarMaxShortNameLength`
Type: integer
Value: integer value between 32 and 128, inclusive
Default: 128

See Also

Related Examples

- “Model Configuration Parameters: Code Generation AUTOSAR” on page 6-2
- “Specify Maximum SHORT-NAME Length” (AUTOSAR Blockset)

Transport layer

Description

Select transport layer for XCP Slave communication

Category: Code Generation > AUTOSAR Code Generation Options

Settings

Default: `None`

`None`

XCP transport layer information is not provided

`XCP On TCP/IP`

Use the TCP/IP transport layer for XCP communication and enable additional parameters for specifying interface details

Tip

Set the same value for top and referenced models.

Command-Line Information

Parameter: `AdaptiveAutosarXCPSlaveTransportLayer`
Type: character vector
Value: `'None'` | `'XCP On TCP/IP'`
Default: `'None'`

See Also

Related Examples

- “Model Configuration Parameters: Code Generation AUTOSAR” on page 6-2
- “Specify XCP Slave Transport Layer” (AUTOSAR Blockset)
- “Configure AUTOSAR Adaptive Data for Run-Time Measurement and Calibration” (AUTOSAR Blockset)

IP address

Description

Specify IP address for XCP communication

Category: Code Generation > AUTOSAR Code Generation Options

Settings

Default: 127.0.0.1

To specify the IP address of the machine on which the AUTOSAR adaptive application (XCP Slave) executes, use this parameter.

Tip

Set the same value for top and referenced models.

Command-Line Information

Parameter: `AdaptiveAutosarXCPSlaveTCPIPAddress`
Type: character vector
Value: valid IP address
Default: 127.0.0.1

See Also

Related Examples

- “Model Configuration Parameters: Code Generation AUTOSAR” on page 6-2
- “Specify XCP Slave IP Address” (AUTOSAR Blockset)
- “Configure AUTOSAR Adaptive Data for Run-Time Measurement and Calibration” (AUTOSAR Blockset)

Port

Description

Specify network port for XCP communication

Category: Code Generation > AUTOSAR Code Generation Options

Settings

Default: 17725

To specify the network port on which the AUTOSAR adaptive application (XCP Slave) serves XCP Master commands, use this parameter.

Tip

Set the same value for top and referenced models.

Command-Line Information

Parameter: `AdaptiveAutosarXCPSlavePort`
Type: character vector
Value: valid port number
Default: 17725

See Also

Related Examples

- “Model Configuration Parameters: Code Generation AUTOSAR” on page 6-2
- “Specify XCP Slave Port” (AUTOSAR Blockset)
- “Configure AUTOSAR Adaptive Data for Run-Time Measurement and Calibration” (AUTOSAR Blockset)

Verbose

Description

Enable verbose XCP Slave messages

Category: Code Generation > AUTOSAR Code Generation Options

Settings

Default: Off

On

Software generates code with verbose messages

Off

Software generates code that does *not* contain verbose messages

Tip

Set the same value for top and referenced models.

Command-Line Information

Parameter: `AdaptiveAutosarXCPSlaveVerbosity`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

Related Examples

- “Model Configuration Parameters: Code Generation AUTOSAR” on page 6-2
- “Enable XCP Slave Message Verbosity” (AUTOSAR Blockset)
- “Configure AUTOSAR Adaptive Data for Run-Time Measurement and Calibration” (AUTOSAR Blockset)

Use custom XCP Slave

Description

Specify whether to use custom or default XCP Slave

Category: Code Generation > AUTOSAR Code Generation Options

Settings

Default: Off

On

Use a custom XCP Slave, with an implementation for XCP functions provided in the XCP header file `xcp_slave.h`

Off

Use the default MathWorks XCP Slave

Tip

Set the same value for top and referenced models.

Command-Line Information

Parameter: `AdaptiveAutosarUseCustomXCPSlave`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

Related Examples

- “Model Configuration Parameters: Code Generation AUTOSAR” on page 6-2
- “Use Custom XCP Slave” (AUTOSAR Blockset)
- “Configure AUTOSAR Adaptive Data for Run-Time Measurement and Calibration” (AUTOSAR Blockset)

Code Generation Parameters: Code Placement

Model Configuration Parameters: Code Generation Code Placement

The **Code Generation > Code Placement** category includes parameters for configuring the appearance of the generated code. On the Configuration Parameters dialog box, the following configuration parameters are on the **Commonly Used** tab on the **Code Generation > Code Placement** pane.

Parameter	Description
“Data definition” on page 7-4	Specify where to place definitions of global variables.
“Data definition filename” on page 7-6	Specify the name of the file that is to contain data definitions.
“Data declaration” on page 7-8	Specify where `extern`, `typedef`, and `#define` statements are to be declared.
“Data declaration filename” on page 7-10	Specify the name of the file that is to contain data declarations.
“#include file delimiter” on page 7-13	Specify the type of `#include` file delimiter to use in generated code.
“Use owner from data object for data definition placement” on page 7-12	Specify whether the model uses or ignores the ownership setting of a data object for data definition in code generation.
“Signal display level” on page 7-14	Specify the persistence level for MPT signal data objects.
“Parameter tune level” on page 7-15	Specify the persistence level for MPT parameter data objects.
“File packaging format” on page 7-16	Specify whether code generation modularizes the code components into many files or compacts the generated code into a few files.
“Header files” on page 7-18	Specify customized name for generated header files.
“Source files” on page 7-20	Specify customized name for generated source files.
“Data files” on page 7-22	Specify customized name for generated data files.
“Rate Transition block code” on page 7-24	Specify the format for Rate Transition block code and data.

See Also

More About

- “Model Configuration”

Code Generation: Code Placement Tab Overview

Specify the data placement in the generated code.

Configuration

This tab appears only if you specify an ERT based system target file.

To get help on an option

1. Right-click the option text label.
2. From the context menu, select **What's This**.

See Also

Related Examples

- "Model Configuration Parameters: Code Generation Code Placement" on page 7-2

Data definition

Description

Specify where to place definitions of global variables.

Category: Code Generation > Code Placement

Settings

Default: `Auto`

`Auto`

Lets the code generator determine where the definitions should be located.

`Data defined in source file`

Places definitions in `.c` source files where functions are located. The code generator places the definitions in one or more function `.c` files, depending on the number of function source files and the file partitioning previously selected in the Simulink model.

`Data defined in a single separate source file`

Places definitions in the source file specified in the **Data definition filename** field. The code generator organizes and formats the definitions based on the data source template specified by the **Source file (*.c) template** parameter in the data section of the **Templates** pane.

Limitation

This parameter applies to data with storage classes except these:

- `ExportedGlobal`
- `ImportedExtern`
- `ImportedExternPointer`
- `BitField`
- `FileScope`
- `Localizable`
- `Struct`
- `CompilerFlag`

In the Embedded Coder Dictionary, the **Header File** value for unsupported storage classes is empty. You cannot specify default file placement for unsupported storage classes.

Dependency

This parameter enables **Data definition filename**.

Command-Line Information

Parameter: `GlobalDataDefinition`
Type: character vector

Value: `'Auto'` | `'InSourceFile'` | `'InSeparateSourceFile'`
Default: `'Auto'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid value
Efficiency	No impact
Safety precaution	No impact

See Also

“Data definition filename” on page 7-6

Related Examples

- “Model Configuration Parameters: Code Generation Code Placement” on page 7-2
- “Control Placement of Global Data Definitions and Declarations in Generated Files”

Data definition filename

Description

Specify the name of the file that is to contain data definitions.

Category: Code Generation > Code Placement

Settings

Default: `global.c`

The code generator organizes and formats the data definitions in the specified file based on the data source template specified by the **Source file (*.c) template** parameter in the data section of the **Code Generation** pane: **Templates** tab.

You do not need to specify an extension for the file name. If you want to specify an extension, you must use a `.c` extension. In either case:

- If you select C as the target language, the code generator creates a file with a `.c` extension.
- If you select C++ as the target language, the code generator creates a file with a `.cpp` extension.

Limitation

The code generator does not check for unique filenames. Specify filenames that do not collide with default filenames from code generation.

Dependency

This parameter is enabled by **Data definition**.

Command-Line Information

Parameter: `DataDefinitionFile`
Type: character vector
Value: a valid file
Default: `'global.c'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid file
Efficiency	No impact
Safety precaution	No impact

See Also

"Data definition" on page 7-4

Related Examples

- “Model Configuration Parameters: Code Generation Code Placement” on page 7-2
- Selecting and Defining Templates
- Custom File Processing

Data declaration

Description

Specify where `extern`, `typedef`, and `#define` statements are to be declared.

Category: Code Generation > Code Placement

Settings

Default: `Auto`

`Auto`

Lets the code generator determine where the declarations should be located.

`Data declared in source file`

Places declarations in `.c` source files where functions are located. The data header template file is not used. The code generator places the declarations in one or more function `.c` files, depending on the number of function source files and the file partitioning previously selected in the Simulink model.

`Data defined in a single separate source file`

Places declarations in the data header file specified in the **Data declaration filename** field. The code generator organizes and formats the declarations based on the data header template specified by the **header file (*.h) template** parameter in the data section of the **Code Generation** pane: **Templates** tab.

Limitation

This parameter applies to data with storage classes except these:

- `ExportedGlobal`
- `ImportedExtern`
- `ImportedExternPointer`
- `BitField`
- `FileScope`
- `Localizable`
- `Struct`
- `CompilerFlag`

Dependency

This parameter enables **Data declaration filename**.

Command-Line Information

Parameter: `GlobalDataReference`
Type: character vector
Value: `'Auto'` | `'InSourceFile'` | `'InSeparateHeaderFile'`

Default: `'Auto'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid value
Efficiency	No impact
Safety precaution	No impact

See Also

"Data declaration filename" on page 7-10

Related Examples

- "Model Configuration Parameters: Code Generation Code Placement" on page 7-2
- "Control Placement of Global Data Definitions and Declarations in Generated Files"

Data declaration filename

Description

Specify the name of the file that is to contain data declarations.

Category: Code Generation > Code Placement

Settings

Default: `global.h`

The code generator organizes and formats the data declarations in the specified file based on the data header template specified by the **Header file (*.h) template** parameter in the data section of the **Code Generation** pane: **Templates** tab.

Limitation

The code generator does not check for unique filenames. Specify filenames that do not collide with default filenames from code generation.

Dependency

This parameter is enabled by **Data declaration**.

Command-Line Information

Parameter: `DataReferenceFile`
Type: character vector
Value: a valid file
Default: `'global.h'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid file
Efficiency	No impact
Safety precaution	No impact

See Also

“Data declaration” on page 7-8

Related Examples

- “Model Configuration Parameters: Code Generation Code Placement” on page 7-2
- Selecting and Defining Templates

- Custom File Processing

Use owner from data object for data definition placement

Description

Specify whether the model uses or ignores the ownership setting of a data object for data definition in code generation.

Category: Code Generation > Code Placement

Settings

Default: off

On

Uses the ownership setting of the data object for data definition.

Off

Ignores the ownership setting of the data object for data definition.

Command-Line Information

Parameter: `EnableDataOwnership`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid value
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Placement” on page 7-2

#include file delimiter

Description

Specify the type of `#include` file delimiter to use in generated code.

Category: Code Generation > Code Placement

Settings

Default: `Auto`

`Auto`

Lets the code generator choose the `#include` file delimiter

`#include “header.h”`

Uses double quote (" ") characters to delimit file names in `#include` statements.

`#include <header.h>`

Uses angle brackets (< >) to delimit file names in `#include` statements.

Dependency

The delimiter format that you use when specifying parameter and signal object property values overrides what you set for this parameter.

Command-Line Information

Parameter: `IncludeFileDelimiter`
Type: character vector
Value: `'Auto'` | `'UseQuote'` | `'UseBracket'`
Default: `'Auto'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid value
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Placement” on page 7-2

Signal display level

Description

Specify the persistence level for MPT signal data objects.

Category: Code Generation > Code Placement

Settings

Default: 10

Specify an integer value indicating the persistence level for MPT signal data objects. This value indicates the level at which to declare signal data objects as global data in the generated code. The persistence level allows you to make intermediate variables global during initial development so you can remove them during later stages of development to gain efficiency.

This parameter is related to the **Persistence level** value that you can specify for a specific MPT signal data object in the mpt.Signal properties dialog.

Dependency

This parameter must be the same for top-level and referenced models.

Command-Line Information

Parameter: `SignalDisplayLevel`
Type: integer
Value: a valid integer
Default: `10`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid integer
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Placement” on page 7-2
- Selecting Persistence Level for Signals and Parameters

Parameter tune level

Description

Specify the persistence level for MPT parameter data objects.

Category: Code Generation > Code Placement

Settings

Default: 10

Specify an integer value indicating the persistence level for MPT parameter data objects. This value indicates the level at which to declare parameter data objects as tunable global data in the generated code. The persistence level allows you to make intermediate variables global and tunable during initial development so you can remove them during later stages of development to gain efficiency.

This parameter is related to the **Persistence level** value you that can specify for a specific MPT parameter data object in the mpt.Parameter properties dialog.

Dependency

This parameter must be the same for top-level and referenced models.

Command-Line Information

Parameter: `ParamTuneLevel`
Type: integer
Value: a valid integer
Default: `10`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid integer
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Placement” on page 7-2
- Selecting Persistence Level for Signals and Parameters

File packaging format

Description

Specify whether code generation modularizes the code components into many files or compacts the generated code into a few files. You can specify a different file packaging format for each referenced model.

Category: Code Generation > Code Placement

Settings

Default: `Modular`

`Modular`

- Outputs *model*`_data.c`, *model*`_private.h`, and *model*`_types.h`, in addition to generating *model*`.c` and *model*`.h`. For the contents of these files, see the table in “Generated Code Modules”.
- Supports generating separate source files for subsystems. For more information on generating code for subsystems, see “Control Generation of Functions for Subsystems”.
- If you specify **Shared code placement** as `Auto` on the **Code Generation** > **Interface** pane of the Configuration Parameter dialog box, some utility files are in the build directory. If you specify **Shared code placement** as `Shared location`, separate files are generated for utility code in a shared location.

`Compact (with separate data file)`

- Conditionally outputs *model*`_data.c`, in addition to generating *model*`.c` and *model*`.h`.
- If you specify **Shared code placement** as `Auto` on the **Code Generation** > **Interface** pane of the Configuration Parameter dialog box, utility algorithms are defined in *model*`.c`. If you specify **Shared code placement** as `Shared location`, separate files are generated for utility code in a shared location.
- Does not support separate source files for subsystems.
- Does not support models with noninlined S-functions.

`Compact`

- The contents of *model*`_data.c` are in *model*`.c`.
- The contents of *model*`_private.h` and *model*`_types.h` are in *model*`.h` or *model*`.c`.
- If you specify **Shared code placement** as `Auto` on the **Code Generation** > **Interface** pane of the Configuration Parameter dialog box, utility algorithms are defined in *model*`.c`. If you specify **Shared code placement** as `Shared location`, separate files are generated for utility code in a shared location.
- Does not support separate source files for subsystems.
- Does not support models with noninlined S-functions.

Command-Line Information

Parameter: `ERTFilePackagingFormat`
Type: character vector
Value: `'Modular'` | `'CompactWithDataFile'` | `'Compact'`
Default: `'Modular'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Placement” on page 7-2
- “Customize Generated Code Modules”
- “Manage File Packaging of Generated Code Modules”
- “Customize Post-Code-Generation Build Processing”
- “Generate Shared Utility Code”

Header files

Description

Specify customized name for generated header files.

Category: Code Generation > Code Placement

Settings

Default: `$R$E`

Enter a macro that specifies whether, and in what order, certain text is to be included in the generated identifier. The macro can include a combination of custom user text and these format tokens.

Token	Description
`$E`	Insert the file type. `$E` represents these instances of file types: • `capi` • `capi_host` • `dt` • `testinterface` • `private` • `types` Required.
`$R`	Insert root model name into identifier, replacing unsupported characters with the underscore (_) character. Required for model referencing.
`$U`	Insert text that you specify for the `$U` token. To specify this text, use the **Custom token text** on page 15-44 parameter.

Custom naming is supported only for `.h` and `.hpp` files. When you have model hierarchy, custom naming is applicable to only the root model.

Dependency

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Command-Line Information

Parameter: `ERTHeaderFileRootName`
Type: character vector

Value: Valid combination of tokens and custom text
Default: `$R$E`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Placement” on page 7-2
- “Customize Generated File Names”
- “Identifier Format Control”

Source files

Description

Specify customized name for generated source files.

Category: Code Generation > Code Placement

Settings

Default: `$R$E`

Enter a macro that specifies whether, and in what order, certain text is to be included in the generated identifier. The macro can include a combination of custom user text and these format tokens:

Token	Description
`$E`	Insert the file type. `$E` represents these instances of file types: • `capi` • `capi_host` • `dt` • `testinterface` • `private` • `types` Required.
`$R`	Insert root model name into identifier, replacing unsupported characters with the underscore (_) character. Required for model referencing.
`$U`	Insert text that you specify for the `$U` token. To specify this text, use the **Custom token text** on page 15-44 parameter.

Custom naming is supported only for `.c` and `.cpp` files. When you have model hierarchy, custom naming is applicable to only the root model.

Dependency

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Command-Line Information

Parameter: `ERTSourceFileRootName`
Type: character vector

Value: Valid combination of tokens and custom text
Default: RE

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Placement” on page 7-2
- “Customize Generated File Names”
- “Identifier Format Control”

Data files

Description

Specify customized name for generated data files.

Category: Code Generation > Code Placement

Settings

Default: `$R_data`

Enter a macro that specifies whether, and in what order, certain text is to be included in the generated identifier. The macro can include a combination of custom user text and these format tokens:

Token	Description
`$R`	Insert root model name into identifier, replacing unsupported characters with the underscore (_) character. Required for model referencing.
`$U`	Insert text that you specify for the `$U` token. To specify this text, use the **Custom token text** on page 15-44 parameter.

Custom naming is supported only for `.c` and `.cpp` files. When you have model hierarchy, custom naming is applicable to only the root model.

Dependency

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.
- `Compact (with separate data file)` option for **File packaging format** on page 7-16 enables this parameter.

Command-Line Information

Parameter: `ERTDataFileRootName`
Type: character vector
Value: Valid combination of tokens and custom text
Default: `$R_data`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Use default

Application	Setting
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Placement” on page 7-2
- “Customize Generated File Names”
- “Identifier Format Control”

Rate Transition block code

Description

Specify the format for Rate Transition block code and data. Inline the code with the model code or create separate functions that the model code calls with state data in a dedicated structure.

Category: Code Generation > Code Placement

Settings

Default: `Inline`

`Inline`

Inline Rate Transition block code with model code. Declare Rate Transition block state data in global block state structure.

`Function`

Separate Rate Transition block code and data from the model code and data. The generated code contains separate `get` and `set` functions that the *`model`*`_step` functions call and a dedicated structure for state data. The generated code also contains separate start and initialize functions that the *`model`*`_initialize` function calls.

Dependencies

- This parameter requires an Embedded Coder license.
- Appears only for ERT-based targets.

Command-Line Information

Parameter: `RateTransitionBlockCode`
Value: `'Inline'` | `'Function'` |
Default: `'Inline'`

Recommended Settings

Application	Setting
Debugging	Function
Traceability	Function
Efficiency	Inline
Safety precaution	No impact

Note

- The code generator does not separate code and data for Rate Transition blocks that have variable-size signals or are inside a For Each Subsystem block.

- In the Rate Transition block parameters dialog box, you must select the **Ensure data integrity during data transfer** parameter. If you do not select this parameter, the model produces an error during code generation.
- In Configuration Parameters dialog box, the **Multitask rate transition** parameter must be set to `error`. If this parameter is not set to `error`, Embedded Coder disables the **Rate Transition block code** parameter and the code generator inlines Rate Transition block code.

See Also

Related Examples

- "Time-Based Scheduling"

8

Code Generation Parameters: Code Style

Model Configuration Parameters: Code Style

The **Code Generation > Code Style** category includes parameters for configuring the appearance of the generated code. These parameters require a Simulink Coder license. Additional parameters for an ERT-based target require an Embedded Coder license.

You can change the code style, cast expressions, and indentation of your generated code to conform to certain coding standards.

Code style modifications have the following uses:

- Enhance the readability and traceability of code
- Convey information across files
- Enhance the efficiency of the generated code
- Allow memory manipulation through type casting

On the Configuration Parameters dialog box, the following configuration parameters are on the **Commonly Used** tab on the **Code Generation > Code Style** pane.

Parameter	Description
"Parentheses level" on page 8-5	Specify parenthesization style for generated code.
"Preserve operand order in expression" on page 8-7	Specify whether to preserve order of operands in expressions.
"Preserve condition expression in if statement" on page 8-8	Specify whether to preserve empty primary condition expressions in `if` statements.
"Convert if-elseif-else patterns to switch-case statements" on page 8-10	Specify whether to generate code for `if-elseif-else` decision logic as `switch-case` statements.
"Preserve extern keyword in function declarations" on page 8-12	Specify whether to include the `extern` keyword in function declarations in the generated code.
"Preserve static keyword in function declarations" on page 8-14	Specify whether to include the `static` keyword in function declarations in the generated code.
"Suppress generation of default cases for Stateflow switch statements if unreachable" on page 8-16	Specify whether to generate default cases for switch-case statements in the code for Stateflow charts.
"Replace multiplications by powers of two with signed bitwise shifts" on page 8-18	Specify whether to replace multiplications by powers of two with signed bitwise shifts.
"Allow right shifts on signed integers" on page 8-20	Specify whether to allow signed right bitwise shifts in the generated C/C++ code.
"Casting modes" on page 8-22	Specify how the code generator casts data types for variables.
"Array container type" on page 8-24	Specify the container type for arrays in the generated code. Choose either C-style array or `std::array`.

Parameter	Description
“Indent style” on page 8-25	Specify style for the placement of braces in generated code.
“Indent size” on page 8-27	Specify indent size for generated code.
“Newline style” on page 8-28	Specify the newline style for generated code.
“Maximum line width” on page 8-29	Specify the maximum line width for wrapping the generated code.

See Also

More About

- “Code Appearance”
- “Model Configuration”

Code Generation: Code Style Tab Overview

Control optimizations for readability in generated code.

Configuration

This tab appears only if you specify an ERT based system target file.

To get help on an option

1. Right-click the option text label.
2. From the context menu, select **What's This**.

See Also

Related Examples

- "Model Configuration Parameters: Code Style" on page 8-2

Parentheses level

Description

Specify parenthesization style for generated code.

Category: Code Generation > Code Style

Settings

Default: `Nominal (Optimize for readability)`

Minimum (Rely on C/C++ operators for precedence)

Inserts parentheses only where required by ANSI[1] C or C++, or to override default precedence. For example:

```
Out = In2 - In1 > 1.0 && In2 > 2.0;
```

If you generate C/C++ code using the minimum level, for certain settings in some compilers, you can receive compiler warnings. To eliminate these warnings, try the nominal level.

Nominal (Optimize for readability)

Inserts parentheses in a way that compromises between readability and visual complexity. For example:

```
Out = ((In2 - In1 > 1.0) && (In2 > 2.0));
```

Maximum (Specify precedence with parentheses)

Includes parentheses to specify meaning without relying on operator precedence. Code generated with this setting conforms to MISRA[2] requirements. For example:

```
Out = (((In2 - In1) > 1.0) && (In2 > 2.0));
```

Command-Line Information

Parameter: `ParenthesesLevel`
Type: character vector
Value: `'Minimum'` | `'Nominal'` | `'Maximum'`
Default: `'Nominal'`

Recommended Settings

Application	Setting
Debugging	`Nominal (Optimized for readability)`
Traceability	`Nominal (Optimized for readability)`
Efficiency	`Minimum (Rely on C/C++ operators for precedence)`

1. ANSI is a registered trademark of the American National Standards Institute, Inc.
2. MISRA is a registered trademarks of MIRA Ltd, held on behalf of the MISRA Consortium.

Application	Setting
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Style” on page 8-2
- “Control Parentheses in Generated Code”

Preserve operand order in expression

Description

Specify whether to preserve order of operands in expressions.

Category: Code Generation > Code Style

Settings

Default: off

On

Preserves the expression order specified in the model. Select this option to increase readability of the code or for code traceability purposes.

```
A*(B+C)
```

Off

Optimizes efficiency of code for nonoptimized compilers by reordering commutable operands to make expressions left-recursive. For example:

```
(B+C)*A
```

Command-Line Information

Parameter: `PreserveExpressionOrder`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Off
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Style” on page 8-2
- “Optimize Code by Reordering Commutable Operands”

Preserve condition expression in if statement

Description

Specify whether to preserve empty primary condition expressions in `if` statements.

Category: Code Generation > Code Style

Settings

Default: off

On

Preserves empty primary condition expressions in `if` statements, such as the following, to increase the readability of the code or for code traceability purposes.

```
if expression1
else
       statements2;
end
```

Off

Optimizes empty primary condition expressions in `if` statements by negating them. For example, consider the following `if` statement:

```
if expression1
else
       statements2;
end
```

By default, the code generator negates this statement as follows:

```
if ~expression1
        statements2;
end
```

Command-Line Information

Parameter: `PreserveIfCondition`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Off (execution, ROM), No impact (RAM)
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Style” on page 8-2
- “Optimize Generated Code by Consolidating Redundant If-Else Statements”

Convert if-elseif-else patterns to switch-case statements

Description

Specify whether to generate code for `if-elseif-else` decision logic as `switch-case` statements.

This readability optimization works on a per-model basis and applies only to:

- Flow charts in Stateflow charts
- MATLAB functions in Stateflow charts
- MATLAB Function blocks in that model

Category: Code Generation > Code Style

Settings

Default: on

On

Generate code for `if-elseif-else` decision logic as `switch-case` statements.

For example, assume that you have the following logic pattern:

```
if (x == 1) {
       y = 1;
} else if (x == 2) {
       y = 2;
} else if (x == 3) {
       y = 3;
} else {
       y = 4;
}
```

Selecting this check box converts the `if-elseif-else` pattern to the following `switch-case` statements:

```
switch (x) {
     case 1:
       y = 1; break;
     case 2:
       y = 2; break;
     case 3:
       y = 3; break;
     default:
       y = 4; break;
}
```

Off

Preserve `if-elseif-else` decision logic in generated code.

Command-Line Information

Parameter: `ConvertIfToSwitch`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Off
Efficiency	On (execution, ROM), No impact (RAM)
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Style” on page 8-2
- “Enhance Readability of Code for Flow Charts”
- “Enhance Code Readability for MATLAB Function Blocks”

Preserve extern keyword in function declarations

Description

Specify whether to include the `extern` keyword in function declarations in the generated code.

Note The `extern` keyword is optional for functions with external linkage. It is considered good programming practice to include the `extern` keyword in function declarations for code readability.

Category: Code Generation > Code Style

Settings

Default: on

On

Include the `extern` keyword in function declarations in the generated code. For example, the generated code for the model `rtwdemo_hyperlinks` contains the following function declarations in `rtwdemo_hyperlinks.h`:

```
/* Model entry point functions */
extern void rtwdemo_hyperlinks_initialize(void);
extern void rtwdemo_hyperlinks_step(void);
```

The `extern` keyword explicitly indicates that the function has external linkage. The function definitions in this example are in the generated file `rtwdemo_hyperlinks.c`.

Off

Remove the `extern` keyword from function declarations in the generated code.

Command-Line Information

Parameter: `PreserveExternInFcnDecls`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Style” on page 8-2

Preserve static keyword in function declarations

Description

Specify whether to include the `static` keyword in function declarations in the generated code.

Category: Code Generation > Code Style

Settings

Default: on

On

Include the `static` keyword in function declarations in the generated code. You can link different executables generated from different models that refer to locally scoped subsystem and utility functions with the same name. This parameter also impacts these functions:

- Stateflow graphical function
- Variant subsystem
- MATLAB subfunction
- Privately scoped Simulink function

When you select this parameter, the generated code is compliant with MISRA C:2012 Rule 8.10.

Off

Remove the `static` keyword in function declarations in the generated code.

Dependency

- This parameter requires Embedded Coder license when you generate code.
- This parameter appears only for ERT-based targets.
- This parameter is enabled when you select `Compact/Compact(with separate data file)` file packaging.

Command-Line Information

Parameter: `PreserveStaticInFcnDecls`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On (execution, ROM)

Application	Setting
Safety precaution	No impact

See Also

Related Examples

- "Model Configuration Parameters: Code Style" on page 8-2
- `MISRA C:2012 Rule 8.10` (Polyspace Code Prover)

Suppress generation of default cases for Stateflow switch statements if unreachable

Description

Specify whether to generate default cases for switch-case statements in the code for Stateflow charts. This optimization works on a per-model basis. It applies to the code generated for a state that has multiple substates. For a list of the state functions in the generated code, see “Inline State Functions in Generated Code”.

Category: Code Generation > Code Style

Settings

Default: on

On

Do not generate the default case when it is unreachable. This setting enables better code coverage because every branch in the generated code is falsifiable.

Off

Generate a default case whether or not it is reachable. This setting supports MISRA C® compliance and provides a backup in case of RAM corruption.

For example, when the state has a nontrivial `entry` function, the following default case appears in the generated code for the `during` function:

```
default:
 entry_internal();
 break;
```

In this case, the code marks the corresponding substate as active.

Command-Line Information

Parameter: `SuppressUnreachableDefaultCases`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	Noimpact
Traceability	On
Efficiency	On (execution, ROM), Noimpact (RAM)
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Style” on page 8-2
- “Suppress Generation of Default Cases for Unreachable Stateflow Switch Statements”

Replace multiplications by powers of two with signed bitwise shifts

Description

Specify whether to replace multiplications by powers of two with signed bitwise shifts. Some coding standards, such as MISRA, do not allow bitwise operations on signed integers. Clearing this option increases the likelihood of generating MISRA C compliant code.

Category: Code Generation > Code Style

Settings

Default: on

On

Generate code that replaces multiplications by powers of two with signed bitwise shifts.

For example, when you select this option, multiplications by 8 are left-shifted in the generated code:

```
Y.Out1 = (U.In1 << ((int8_T)3));
```

Similarly, multiplications by 16 are left-shifted in the generated code:

```
Y.Out4 = (U.In2 << ((int8_T)4));
```

Off

Do not allow replacement of multiplications by powers of two with signed shifts. Clearing this option supports MISRA C compliance.

For example, when you clear this option, multiplications by 8 are not replaced by bitwise shifts:

```
Y.Out1 = U.In1 * ((int64_T)8);
```

Similarly, multiplications by 16 are not replaced by bitwise shifts:

```
Y.Out4 = U.In2 * ((int32_T)16);
```

Command-Line Information

Parameter: `EnableSignedLeftShifts`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On

Application	Setting
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Style” on page 8-2
- “Replace Multiplication by Powers of Two with Signed Bitwise Shifts”

Allow right shifts on signed integers

Description

Specify whether to allow signed right bitwise shifts in the generated C/C++ code. Some coding standards, such as MISRA, do not allow bitwise operations on signed integers. Clearing this option increases the likelihood of generating MISRA-C:2004 compliant code.

Category: Code Generation > Code Style

Settings

Default: on

On

Generate code that uses right bitwise shifts on signed integers.

For example, when you select this option, right shifts appear in the generated code.

```
i >>= 3
```

Off

Do not allow right shifts on signed integers. Clearing this option supports MISRA C compliance.

For example, when you clear this option, right shifts are replaced with a function call.

```
i = asr_s32(i, 3U);
```

Command-Line Information

Parameter: `EnableSignedRightShifts`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On
Safety precaution	No impact

See Also

Related Examples

- "Model Configuration Parameters: Code Style" on page 8-2

- “Generate Code Containing Right Shifts on Signed Integers”

Casting modes

Description

Specify how the code generator casts data types for variables.

Category: Code Generation > Code Style

Settings

Default: `Nominal`

Nominal

Generate code that uses default C compiler data type casting.

For example:

```
void rtwdemo_rtwecintro_step(void)
  {
    boolean_T rtb_equal_to_count;
    rtDWork.X++;
    rtb_equal_to_count = (rtDWork.X != 16);
    if (rtb_equal_to_count && (rtPrevZCSigState.Amplifier_Trig_ZCE != POS_ZCSIG))
     { rtY.Output = rtU.Input << 1;
     }
```

Standards Compliant

Generate code where data type casting complies with MISRA standards.

For example:

```
void rtwdemo_rtwecintro_step(void)
    {
      boolean_T rtb_equal_to_count;
      rtDWork.X = (uint8_T)((uint32_T)rtDWork.X + 1U);
      rtb_equal_to_count = ((int32_T)rtDWork.X != 16);
      if (rtb_equal_to_count && ((uint32_T)rtPrevZCSigState.Amplifier_Trig_ZCE !=
           POS_ZCSIG)) {
        rtY.Output = rtU.Input << 1U;
      }
```

Note The expression `rtY.Output = rtU.Input << 1U` is not compliant with MISRA C:12 Rule 10.1 because the model configuration parameter **Replace multiplications by powers of two with signed bitwise shift** is selected. For more information, see “Replace multiplications by powers of two with signed bitwise shifts” on page 8-18.

Depending on the setting, the configuration parameter **Casting modes** can replace bitwise XOR operations with relational operations in the generated code to satisfy the MISRA C:12 Rule 10.1 when the operands are signed types. For example, generate code from the following model with the **Casting modes** set to `Nominal` and `Standard compliant` respectively.

```
// Model step function (casting mode set to Nominal)
void step(void)
{rtY.Out3 = (boolean_T)((int32_T)(rtU.In1 != 0.0F) ^ (int32_T)(rtU.Inport1 !=
```

```
    0.0F));
}
// Model step function (Casting modes set to Standard Compliant)
void step(void)
{  rtY.Out3 = ((rtU.In1 != 0.0F) != (rtU.Inport1 != 0.0F));
}
```

Here, the parameters `rtU.In1` and `rtU.Inport1` are single singed types. Performing a bitwise XOR(^) operation on these operands violates the MISRA C:12 Rule 10.1. To prevent this violation, the code generator replaces the bitwise XOR(^) operation with an inequality(!=) in the generated code when **Casting modes** is set to `Standard compliant`.

Explicit

Generate code that casts data type values explicitly.

For example:

```
void rtwdemo_rtwecintro_step(void)
   {
    boolean_T rtb_equal_to_count;
    rtDWork.X = (uint8_T)((uint32_T)(int32_T)rtDWork.X + 1U);
    rtb_equal_to_count = (boolean_T)((int32_T)rtDWork.X != 16);
    if (((int32_T)rtb_equal_to_count) && ((int32_T)
    rtPrevZCSigState.Amplifier_Trig_ZCE != (int32_T)POS_ZCSIG)) {
    rtY.Output = rtU.Input << 1;
      }
```

Command-Line Information

Parameter: `CastingMode`
Type: character vector
Value: `'Nominal'` | `'Standards'` | `'Explicit'`
Default: `'Nominal'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Style” on page 8-2
- “MISRA C Guidelines”

Array container type

Description

Specify the container type for arrays in the generated code. Choose either C-style array or `std::array`.

Category: Code Generation > Code Style

Settings

Default: `C-style array`

`C-style array`

The code generator generates array containers by using C-style arrays. For example:

```
real_T const_val[4] = { 1.0, 2.0, 3.0, 4.0 } ;
```

`std::array`

`std::array` is a template class that represents fixed-size arrays. If you choose this option, the code generator generates array containers by using `std::array`. For example:

```
std::array<real_T, 4> const_val = { { 1.0, 2.0, 3.0, 4.0 } };
```

Command-Line Information

Parameter: `ArrayContainerType`
Type: character vector
Value: `'C-style array'` | `'std::array'`
Default: `'C-style array'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Style” on page 8-2

Indent style

Description

Specify style for the placement of braces in generated code.

Category: Code Generation > Code Style

Settings

Default: K&R

K&R

For blocks within a function, an opening brace is on the same line as its control statement. For example:

```
void rt_OneStep(void)
{
  static boolean_T OverrunFlag = 0;
  if (OverrunFlag) {
    rtmSetErrorStatus(rtwdemo_counter_M, "Overrun");
    return;
  }

  OverrunFlag = TRUE;
  rtwdemo_counter_step();
  OverrunFlag = FALSE;
}
```

Allman

For blocks within a function, an opening brace is on its own line at the same level of indentation as its control statement. For example:

```
void rt_OneStep(void)
{
  static boolean_T OverrunFlag = 0;
  if (OverrunFlag)
  {
    rtmSetErrorStatus(rtwdemo_counter_M, "Overrun");
    return;
  }

  OverrunFlag = TRUE;
  rtwdemo_counter_step();
  OverrunFlag = FALSE;
}
```

Command-Line Information

Parameter: `IndentStyle`
Type: character vector
Value: `'K&R'` | `'Allman'`
Default: `'K&R'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Style” on page 8-2
- “Replace and Rename Data Types to Conform to Coding Standards”

Indent size

Description

Specify indent size for generated code.

Category: Code Generation > Code Style

Settings

Default: 2

Specify an integer value that indicates the number of characters per indent level. Possible values range from 2–8 characters.

Command-Line Information

Parameter: `IndentSize`
Type: integer
Value: integer from 2–8
Default: 2

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Style” on page 8-2
- “Replace and Rename Data Types to Conform to Coding Standards”

Newline style

Description

Specify the newline character in the generated code.

Category: Code Generation > Code Style

Settings

Default: `Default`

Default

Generates the newline character based on the operating system that the code is generated on.

LF (Line Feed)

Generates the Line Feed character as the newline character in the generated code. `"\n"` is inserted as the newline character.

CR+LF (Carriage Return + Line Feed)

Generates the Carriage Return + Line Feed character as the newline character in the generated code. `"\r\n"` is inserted as the newline character.

Command-Line Information

Parameter: `NewlineStyle`
Type: character vector
Value: `'Default'|'LF'|'CRLF'`
Default: `'Default'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Style” on page 8-2
- “Replace and Rename Data Types to Conform to Coding Standards”

Maximum line width

Description

Specify the maximum line width for wrapping generated code.

Category: Code Generation > Code Style

Settings

Default: 80

Specify an integer value that indicates the maximum number of columns in a single line of generated code. Possible values range from 50–1000 columns.

If the comments exceed the maximum line width specified, the tail comments are generated on a new line with right justification. Other types of comments are not wrapped:

- #define tail comments
- Simulink block comments
- Stateflow object comments
- Banner comments

Example

Here is generated code that is wrapped using the default **Maximum line width** value 80:

```
/* Definition for custom storage class: Default */
real_T const_val[4] = { 1.0, 2.0, 3.0, 4.0 } ;
                  /* This parameter defines the vector of output index values */
```

The tail comments are generated on a new line with right justification.

Here is the same code wrapped with **Maximum line width** set to 120:

```
/* Definition for custom storage class: Default */
real_T const_val[4] = { 1.0, 2.0, 3.0, 4.0 } ;/* This parameter defines the vector of output index values */
```

Command-Line Information

Parameter: MaxLineWidth
Type: integer
Value: integer from 50–1000
Default: 80

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact

Application	Setting
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Style” on page 8-2
- “Replace and Rename Data Types to Conform to Coding Standards”

Code Generation Parameters: Code Generation

- “Model Configuration Parameters: Code Generation” on page 9-2
- “Set Objectives — Code Generation Advisor Dialog Box” on page 9-6

Model Configuration Parameters: Code Generation

The **Code Generation** category includes parameters for defining the code generation process including target selection. It also includes parameters for inserting comments and pragmas into the generated code for data and functions. These parameters require a Simulink Coder license. Additional parameters available with an ERT-based target require an Embedded Coder license. Generating CUDA® C++ code for NVIDIA® GPUs requires a GPU Coder™ license.

These configuration parameters appear in the **Configuration Parameters > Code Generation** general category.

Parameter	Description
“System target file”	Specify which target file configuration will be used.
“Browse”	Browse file configuration options.
“Language”	Specify C or C++ code generation.
“Generate GPU code”	Use GPU Coder for CUDA code generation. This parameter requires a GPU Coder license.
“Description”	A description of the target file.
“Generate code only”	Specify code generation versus an executable build.
“Package code and artifacts”	Specify whether to automatically package generated code and artifacts for relocation.
“Zip file name”	Specify the name of the `.zip` file in which to package generated code and artifacts for relocation.
“Compiler optimization level”	Control compiler optimizations for building generated code.
“Custom compiler optimization flags”	Specify custom compiler optimization flags.
“Toolchain”	Specify the toolchain to use when building an executable or library.
“Build configuration”	Specify compiler optimization or debug settings for toolchain.
“Tool/Options”	Display or customize build configuration settings.
“Generate makefile”	Enable generation of a makefile based on a template makefile.
“Make command”	Specify a make command and optionally append makefile options.
“Template makefile”	Specify the template makefile from which to generate the makefile.
“Select objective”	Select a code generation objective to use with the Code Generation Advisor.
“Prioritized objectives”	List of prioritized code generation objectives.

Parameter	Description
“Set Objectives”	Open Configuration Set Objectives dialog box.
“Set Objectives — Code Generation Advisor Dialog Box” on page 9-6	Select and prioritize code generation objectives.
“Check model before generating code”	Choose whether to run Code Generation Advisor checks before generating code.
“Check Model”	Check whether the model meets code generation objectives.

These configuration parameters are under the **Advanced parameters**.

Parameter	Description
“Custom FFT library callback”	Specify a callback class for FFTW library calls in code generated for FFT functions in MATLAB code.
“Custom BLAS library callback”	Specify BLAS library callback class for BLAS calls in code generated from MATLAB code.
“Custom LAPACK library callback”	Specify LAPACK library callback class for LAPACK calls in code generated from MATLAB code.
“Verbose build”	Display code generation progress.
“Retain .rtw file”	Specify `model.rtw` file retention.
“Profile TLC”	Profile the execution time of TLC files.
“Enable TLC assertion”	Produce the TLC stack trace.
“Start TLC coverage when generating code”	Generate the TLC execution report.
“Start TLC debugger when generating code”	Specify use of the TLC debugger
“Show Custom Hardware App in Simulink Toolstrip”	Read-only internal parameter for Simulink toolstrip.
“Show Embedded Hardware App in Simulink Toolstrip”	Read-only internal parameter for Simulink toolstrip.
“Package” on page 17-3	Specify a package that contains memory sections you want to apply to model-level functions and internal data.
“Refresh package list” on page 17-5	Add user-defined packages that are on the search path to list of packages.
“Initialize/Terminate” on page 17-6	Specify whether to apply a memory section to Initialize/Start and Terminate functions.
“Execution” on page 17-7	Specify whether to apply a memory section to execution functions.
“Shared utility” on page 17-8	Specify whether to apply memory sections to shared utility functions.
“Constants” on page 17-9	Specify whether to apply a memory section to constants.

Parameter	Description
“Inputs/Outputs” on page 17-11	Specify whether to apply a memory section to root input and output.
“Internal data” on page 17-13	Specify whether to apply a memory section to internal data.
“Parameters” on page 17-15	Specify whether to apply a memory section to parameters.
“Validation results” on page 17-16	Display the results of memory section validation.

The following parameters under **Advanced parameters** are infrequently used and have no other documentation.

Parameter	Description
`PostCodeGenCommand` *`character vector`* - `''`	Add the specified post code generation command to the model build process.
`TLCOptions` *`character vector`* - `''`	Specify additional TLC command-line options.

The following parameters are for MathWorks use only.

Parameter	Description
`Comment`	For MathWorks use only.
`PreserveName`	For MathWorks use only.
`PreserveNameWithParent`	For MathWorks use only.
`SignalNamingFcn`	For MathWorks use only.
`TargetTypeEmulationWarnSuppressLevel` int - **`0`**	For MathWorks use only. When greater than or equal to 2, suppress warning messages that the code generator displays when emulating integer sizes in rapid prototyping environments.

The Configuration Parameters dialog box also includes other code generation parameters:

- “Model Configuration Parameters: Code Generation Optimization”
- “Model Configuration Parameters: Code Generation Report”
- “Model Configuration Parameters: Comments”
- “Model Configuration Parameters: Code Generation Identifiers”
- “Model Configuration Parameters: Code Generation Custom Code”
- “Model Configuration Parameters: Code Generation Interface”

See Also

More About

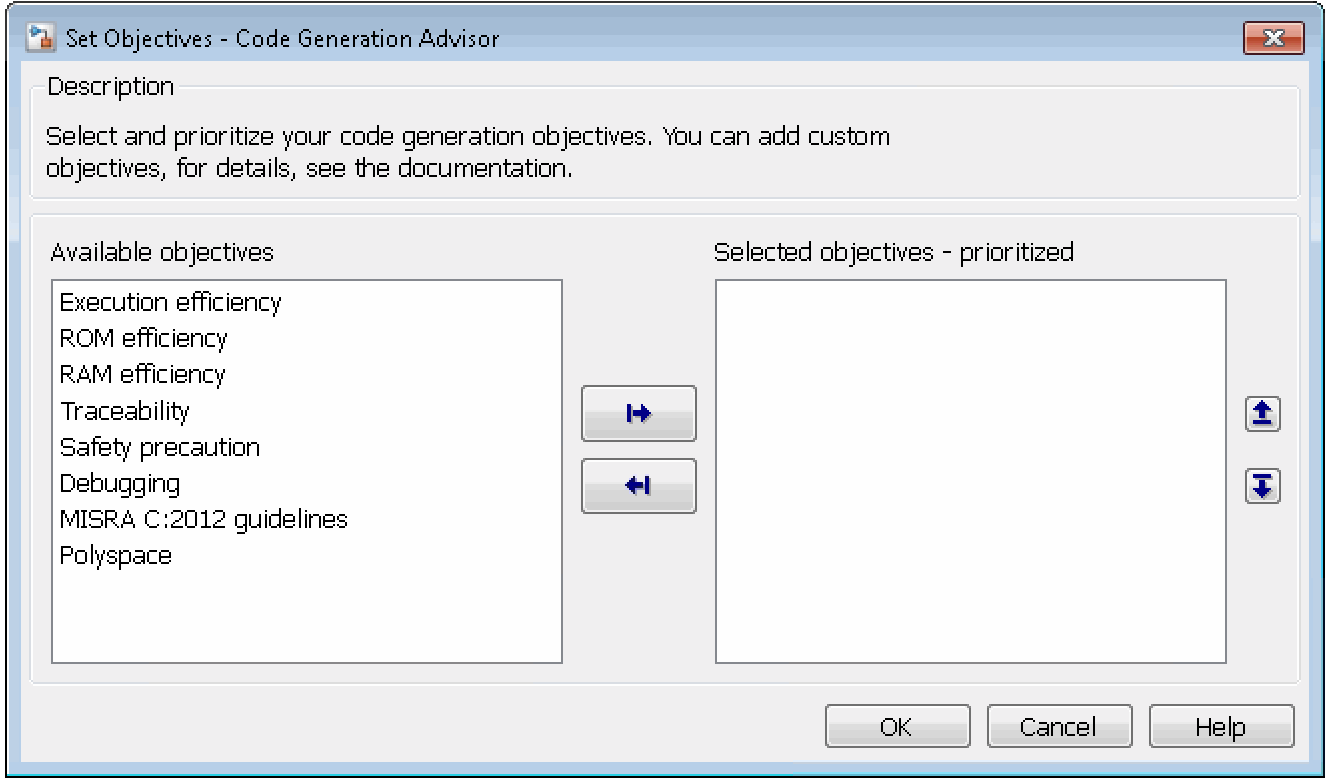
- “Model Configuration”

- “Control Data and Function Placement in Memory by Inserting Pragmas”

Set Objectives — Code Generation Advisor Dialog Box

Description

Select and prioritize code generation objectives to use with the Code Generation Advisor.



Category: Code Generation

Settings

1 From the **Available objectives** list, select objectives.

2 Click the select button (arrow pointing right) to move the objectives that you selected into the **Selected objectives - prioritized** list.

3 Click the higher priority (up arrow) and lower priority (down arrow) buttons to prioritize the objectives.

Objectives

List of available objectives.
`Execution efficiency` — Configure code generation settings to achieve fast execution time.
`ROM efficiency` — Configure code generation settings to reduce ROM usage.
`RAM efficiency` — Configure code generation settings to reduce RAM usage.
`Traceability` — Configure code generation settings to provide mapping between model elements and code.

`Safety precaution` — Configure code generation settings to increase clarity, determinism, robustness, and verifiability of the code.
`Debugging` — Configure code generation settings to debug the code generation build process.
`MISRA C:2012 guidelines` — Configure code generation settings to increase compliance with MISRA C:2012 guidelines.
`Polyspace` — Configure code generation settings to prepare the code for Polyspace® analysis.

Note If you select the MISRA C:2012 guidelines code generation objective, the Code Generation Advisor checks:

- The model configuration settings for compliance with the MISRA C:2012 configuration setting recommendations.
- For blocks that are not supported or recommended for MISRA C:2012 compliant code generation.

Priorities

After you select objectives from the **Available objectives** parameter, organize the objectives in the **Selected objectives - prioritized** parameter with the highest priority objective at the top.

Dependency

This dialog box appears only for ERT-based targets.

Command-Line Information

Parameter: `'ObjectivePriorities'`
Type: cell array of character vectors or string array; combination of the available values
Value: `{''}` | `{'Execution efficiency'}` | `{'ROM efficiency'}` | `{'RAM efficiency'}` | `{'Traceability'}` | `{'Safety precaution'}` | `{'Debugging'}` | `{'MISRA C:2012 guidelines'}` | `{'Polyspace'}`
Default: `{''}`

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 9-2
- “Configure Model for Code Generation Objectives by Using Code Generation Advisor”
- “Application Objectives Using Code Generation Advisor”

Code Generation Parameters: Optimization

- “Model Configuration Parameters: Code Generation Optimization” on page 10-2
- “Pass reusable subsystem outputs as” on page 10-5
- “Remove root level I/O zero initialization” on page 10-7
- “Remove internal data zero initialization” on page 10-9
- “Level” on page 10-11
- “Priority” on page 10-16
- “Specify custom optimizations” on page 10-21
- “Reuse global block outputs” on page 10-23
- “Perform in-place updates for Assignment and Bus Assignment blocks” on page 10-25
- “Reuse buffers for Data Store Read and Data Store Write blocks” on page 10-27
- “Simplify array indexing” on page 10-29
- “Pack Boolean data into bitfields” on page 10-31
- “Bitfield declarator type specifier” on page 10-33
- “Reuse buffers of different sizes and dimensions” on page 10-35
- “Generate parallel for-loops” on page 10-37
- “Optimize global data access” on page 10-38
- “Optimize block operation order in the generated code” on page 10-40
- “Optimize using the specified minimum and maximum values” on page 10-42
- “Remove Code from Tunable Parameter Expressions That Saturate Out-of-Range Values” on page 10-45
- “Remove code that protects against division arithmetic exceptions” on page 10-47
- “Use signal labels to guide buffer reuse” on page 10-49
- “Operator to represent Bitwise and Logical Operator blocks” on page 10-51

Model Configuration Parameters: Code Generation Optimization

The **Code Generation** > **Optimization** category includes parameters for improving the simulation speed of your models and improving the performance of the generated code. Model configuration parameters to improve the generated code require Simulink Coder or Embedded Coder.

Parameter	Description
“Default parameter behavior”	Transform numeric block parameters into constant inlined values in the generated code.
“Pass reusable subsystem outputs as” on page 10-5	Specify how a reusable subsystem passes outputs.
“Remove root level I/O zero initialization” on page 10-7	Specify whether to generate initialization code for root-level inports and outports set to zero.
“Remove internal data zero initialization” on page 10-9	Specify whether to generate initialization code for internal work structures, such as block states and block outputs, to zero.
“Level” on page 10-11	Choose the optimization level that you want to apply to the generated code.
“Priority” on page 10-16	Optimize the generated code for increased execution efficiency, decreased RAM consumption, or a balance between the two.
“Specify custom optimizations” on page 10-21	Instead of applying an optimization level, select this parameter to select the optimization parameters in the **Details** section.
“Use memcpy for vector assignment”	Optimize code generated for vector assignment by replacing `for` loops with `memcpy`.
“Memcpy threshold (bytes)”	Specify the minimum array size in bytes for which `memcpy` and `memset` function calls should replace `for` loops for vector assignments in the generated code.
“Enable local block outputs”	Specify whether block signals are declared locally or globally.
“Reuse local block outputs”	Specify whether Simulink Coder software reuses signal memory.
“Eliminate superfluous local variables (Expression folding)”	Collapse block computations into single expressions.
“Reuse global block outputs” on page 10-23	Reuse global memory for block outputs.
“Perform in-place updates for Assignment and Bus Assignment blocks” on page 10-25	Reuse the input and output variables of Bus Assignment and Assignment blocks if possible.
“Reuse buffers for Data Store Read and Data Store Write blocks” on page 10-27	Remove temporary buffers for Data Store Read and Data Store Write blocks. Use the Data Store Memory block directly if possible.

Parameter	Description
“Simplify array indexing” on page 10-29	Replace multiply operations in array indices when accessing arrays in a loop.
“Pack Boolean data into bitfields” on page 10-31	Specify whether Boolean signals are stored as one–bit bitfields or as a Boolean data type.
“Bitfield declarator type specifier” on page 10-33	Specify the bitfield type when selecting configuration parameter “Pack Boolean data into bitfields” on page 10-31.
“Reuse buffers of different sizes and dimensions” on page 10-35	Reduce memory consumption by reusing buffers to store data of different sizes and dimensions.
“Optimize global data access” on page 10-38	Select global variable optimization.
“Optimize block operation order in the generated code” on page 10-40	Reorder block operations in the generated code for improved code execution speed.
“Use bitsets for storing state configuration”	Use bitsets to reduce the amount of memory required to store state configuration variables.
“Use bitsets for storing Boolean data”	Use bitsets to reduce the amount of memory required to store Boolean data.
“Maximum stack size (bytes)”	Specify the maximum stack size in bytes for your model.
“Loop unrolling threshold”	Specify the minimum signal or parameter width for which a `for` loop is generated.
“Optimize using the specified minimum and maximum values” on page 10-42	Optimize generated code using the specified minimum and maximum values for signals and parameters in the model.
Maximum number of arguments for subsystem outputs	Set maximum number of subsystem outputs to pass individually.
“Inline invariant signals”	Transform symbolic names of invariant signals into constant values.
“Remove code from floating-point to integer conversions with saturation that maps NaN to zero”	Remove code that handles floating-point to integer conversion results for `NaN` values.
“Use memset to initialize floats and doubles to 0.0”	Specify whether to generate code that explicitly initializes floating-point data to 0.0.
“Remove code from floating-point to integer conversions that wraps out-of-range values”	Remove wrapping code that handles out-of-range floating-point to integer conversion results.
“Remove Code from Tunable Parameter Expressions That Saturate Out-of-Range Values” on page 10-45	Remove wrapping code of tunable parameters.
“Remove code that protects against division arithmetic exceptions” on page 10-47	Specify whether to generate code that guards against division by zero and `INT_MIN/-1` operations for integers and fixed-point data.
“Buffer for reusable subsystems”	Improve reuse by inserting buffers at reusable subsystem boundaries.

Parameter	Description
Disable incompatible optimizations	Specify whether to disable optimizations that are incompatible with Simulink Code Inspector.
“Base storage type for automatically created enumerations”	Set the storage type and size for enumerations created with active state output.
“Use signal labels to guide buffer reuse” on page 10-49	For signals with the same label, the code generator attempts to use the same signal memory.
“Generate parallel for-loops” on page 10-37	Specify whether for-loops in the generated code shall be implement in parallel for Matlab Function, Matlab System or a For Each block.
“Signal storage reuse”	Specify reuse of memory buffers allocated to store block input and output signals thereby reducing the memory requirement of real-time program
“Operator to represent Bitwise and Logical Operator blocks” on page 10-51	Specify whether the generated code contains bitwise or logical operators or both.

See Also

Related Examples

- “Performance”

Pass reusable subsystem outputs as

Description

Specify how a reusable subsystem passes outputs.

Category: Optimization

Settings

Default: `Individual arguments`

`Individual arguments`

Passes each reusable subsystem output argument as an address of a local, instead of as a pointer to an area of global memory containing the output arguments. This option reduces global memory usage and eliminates copying local variables back to global block I/O structures. When the signals are allocated as local variables, there may be an increase in stack size. If the stack size increases beyond a level that you want, use the default setting. By default, the maximum number of output arguments passed individually is 12. To increase the number of arguments, increase the value of the **Maximum number of arguments for subsystem outputs** parameter.

`Structure reference`

Passes reusable subsystem outputs as a pointer to a structure stored in global memory.

Note The default option is used for reusable subsystems that have signals with variable dimensions.

Dependencies

This parameter:

- Requires a Embedded Coder license.
- Appears only for ERT-based targets.

Command-Line Information

Parameter: `PassReuseOutputArgsAs`
Value: `'Structure reference'` | `'Individual arguments'`
Default: `'Individual arguments'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	`Individual arguments` (execution, RAM),`Structure reference` (ROM)
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Optimization” on page 10-2
- “Generate Reusable Code from Library Subsystems Shared Across Models”
- “Optimize Generated Code By Passing Reusable Subsystem Outputs as Individual Arguments”
- “Performance”

Remove root level I/O zero initialization

Description

Specify whether to generate initialization code for root-level inports and outports set to zero.

Category: Optimization

Settings

Default: When the **Code interface packaging** model configuration parameter is set to `Nonreusable function`, the **Remove root level I/O zero initialization** check box is selected and at the command line, `ZeroExternalMemoryAtStartup` is set to `'off'` . When the **Code interface packaging** parameter is set to `Reusable function` or `C++ Class`, the **Remove root level I/O zero initialization** check box is cleared and at the command-line `ZeroExternalMemoryAtStartup` is set to `'on'` .

On

Does not generate initialization code for root-level inports and outports set to zero.

During startup, standards-compliant C and C++ compilers initialize global data to zero eliminating the need to include zero initialization code for this data in the generated code. Standards-compliant compilers do not necessarily initialize dynamically allocated data and local variables to zero. Before leaving the **Remove root level I/O zero initialization** parameter selected, confirm that your model meets the following conditions:

- If your compiler is not standards compliant, confirm that it initializes global data to zero.
- If you set the **Code Interface packaging** parameter to `Reusable function` or `C++ class`, confirm that data is either statically allocated or that dynamically allocated data is initialized to zero.

Off

Generates initialization code for root-level inports and outports.

If you set the **Code interface packaging** parameter to `Reusable function` and select the "Use dynamic memory allocation for model initialization" on page 14-28 parameter, the **Remove root level I/O zero initialization** check box is cleared and `ZeroExternalMemoryAtStartup` is set to `'on'`.

Note Generated code does not initialize data whose storage class has imported scope.

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: `ZeroExternalMemoryAtStartup`
Value: `'off'` | `'on'`
Default: `'off'`

Note The command-line values are the reverse of the settings values. Therefore, `'on'` in the command line corresponds to the description of “Off” in the settings section. `'off'` in the command line corresponds to the description of “On” in the settings section.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On (GUI), `off` (command line) (execution, ROM), No impact (RAM)
Safety precaution	No recommendation

See Also

Related Examples

- “Remove Initialization Code from Root-Level Inports and Outports Set to Zero”
- “Model Configuration Parameters: Code Generation Optimization” on page 10-2
- “Performance”

Remove internal data zero initialization

Description

Specify whether to generate initialization code for internal work structures, such as block states and block outputs, to zero.

Category: Optimization

Settings

Default: When the **Code interface packaging** parameter is set to `Nonreusable function`, the **Remove internal data zero initialization** check box is selected and at the command line, `ZeroInternalMemoryAtStartup` is set to `'off'` . When the **Code interface packaging** parameter is set to `Reusable function` or `C++ Class`, the **Remove internal data zero initialization** check box is cleared and `ZeroInternalMemoryAtStartup` is set to `'on'` .

On

Does not generate code that initializes internal work structures to zero.

During startup, standards-compliant C and C++ compilers initialize global data to zero eliminating the need to include zero initialization code for this data in the generated code. Standards compliant compilers do not necessarily initialize dynamically allocated data and local variables to zero. Before leaving the **Remove internal data zero initialization** parameter selected, confirm that your model meets the following conditions:

- If your compiler is not standards-compliant, confirm that it initializes global data to zero.
- If you set the **Code Interface packaging** to `Reusable function` or `C++ class`, confirm that data is either statically allocated or that dynamically allocated data is initialized to zero.

Off

Generates code that initializes internal work structures to zero.

The **Remove internal data zero initialization** check box is cleared and `ZeroInternalMemoryAtStartup` is set to `'on'` and is read-only for a model in which the **Code interface packaging** parameter is set to `C++ class` and the “Use dynamic memory allocation for model initialization” on page 14-28 parameter is selected.

If you set the **Code interface packaging** parameter to `Reusable function` and select the “Use dynamic memory allocation for model block instantiation” on page 14-30 parameter, the **Remove internal data zero initialization** check box is cleared and `ZeroInternalMemoryAtStartup` is set to `'on'`.

Note Generated code does not initialize data whose storage class has imported scope.

Dependencies

- This parameter appears only for ERT-based targets.

- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: `ZeroInternalMemoryAtStartup`
Value: `'off'` | `'on'`
Default: `'off'`

Note The command-line values are the reverse of the settings values. Therefore, `'on'` in the command line corresponds to the description of “Off” in the settings section. `'off'` in the command line corresponds to the description of “On” in the settings section.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On (GUI), `off` (command line), (execution, ROM), No impact (RAM)
Safety precaution	No recommendation

See Also

Related Examples

- “Remove Zero Initialization Code for Internal Data”
- “Model Configuration Parameters: Code Generation Optimization” on page 10-2
- “Performance”

Level

Description

Choose the optimization level that you want to apply to the generated code.

Settings

Default: `Maximum`

`Minimum (Debugging)`

Configure code generation settings for debugging.

`Balanced with Readability`

Apply code generation optimizations that balance RAM efficiency and execution speed with the readability of the generated code. For example, selecting this value disables optimizations that cross atomic subsystem boundaries.

`Maximum`

Configure code generation settings based on your code efficiency objectives. Choosing this setting enables the **Priority** parameter. Set the **Priority** parameter to one of these values:

- `Balance RAM and speed (default setting)`
- `Maximum execution speed`
- `Minimize RAM`

Dependencies

- This parameter appears only for ERT-based targets.
- When generating code, this parameter requires an Embedded Coder license.

Tips

For each **Priority** and **Level** parameter value, there are corresponding values for the parameters in the **Details** section. These are some important differences among these various settings:

- If you set the **Level** parameter to `Minimum (debugging)`, the parameters in the **Details** section are set to off. The code generator does not implement optimizations that remove variables or code making it easier to debug the generated code.
- The parameter settings for `Balanced with Readability` and `Balance RAM and speed` are the same except for these three parameters:
  - **Reuse buffers of different sizes and dimensions**
  - **Optimize global data access**
  - **Optimize block operation order in the generated code**

  The above optimizations can potentially hurt readability because they cross atomic subsystem boundaries and **Optimize block operation order in the generated code** might change the block execution order in the generated code so that it is different than in simulation.

- If you have limited RAM, choose the `Minimize RAM` setting. This setting enables these optimizations that reduce RAM at the expense of a potential slow-down in execution speed:
  - **Pack Boolean data into bitfields**
  - **Reuse buffers of different sizes and dimensions**
  - **Use bitsets for storing state configuration**
  - **Use bitsets for storing Boolean data**

  This setting also changes the **Optimize block operation order in the generated code** from `Improved Code Execution Speed` to `off`.

For each **Priority** and **Level** parameter value, this table lists the corresponding values for the parameters in the **Details** section.

Parameters	Settings					Example
Level	`Minimum (debugging)`	`Balanced with readability`	`Maximum`			
Priority	Not Applicable (N/A)	N/A	`Balance RAM and speed`	`Maximize execution speed`	`Minimize RAM`	
Details						
Use memcpy for vector assignment	`Off`	`On`	`On`	`On`	`On`	“Use memcpy Function to Optimize Generated Code for Vector Assignments”
Memcpy threshold (bytes)	`Off`	`64`	`64`	`64`	`64`	“Use memcpy Function to Optimize Generated Code for Vector Assignments”
Enable local block outputs	`Off`	`On`	`On`	`On`	`On`	“Enable and Reuse Local Block Outputs in Generated Code”

Parameters	Settings					Example
Reuse local block outputs	`Off`	`On`	`On`	`On`	`On`	"Enable and Reuse Local Block Outputs in Generated Code"
Eliminate superfluous local variables (expression folding)	`Off`	`On`	`On`	`On`	`On`	"Minimize Computations and Storage for Intermediate Results at Block Outputs"
Reuse global block outputs	`Off`	`On`	`On`	`On`	`On`	"Reuse Global Block Outputs in the Generated Code"
Perform in-place updates for Assignment and Bus Assignment blocks	`Off`	`On`	`on`	`On`	`On`	"Data copy reduction for Bus Assignment block"
Reuse buffers for Data Store Read and Data Store Write blocks	`Off`	`On`	`On`	`On`	`On`	"Data Copy Reduction for Data Store Read and Data Store Write Blocks"
Simplify array indexing	`Off`	`Off`	`Off`	`On`	`Off`	"Simplify Multiply Operations in Array Indexing"
Pack Boolean data into bitfields	`Off`	`Off`	`Off`	`Off`	`On`	"Optimize Generated Code By Packing Boolean Data Into Bitfields"

Parameters	Settings					Example
Reuse buffers of different sizes and dimensions	`Off`	`Off`	`On`	`Off`	`On`	"Reuse Buffers of Different Sizes and Dimensions"
Optimize global data access	`None`	`None`	`Use global to hold temporary results`	`None`	`Use global to hold temporary results`	"Optimize Global Variable Usage"
Optimize block operation order in the generated code	`Off`	`Off`	`Improved Code Execution Speed`	`Improved Code Execution Speed`	`Off`	"Remove Data Copies by Reordering Block Operations in the Generated Code"
Use bitsets for storing state configuration	`Off`	`Off`	`Off`	`Off`	`On`	"Reduce Memory Usage for Boolean and State Configuration Variables"
Use bitsets for storing Boolean data	`Off`	`Off`	`Off`	`Off`	`On`	"Reduce Memory Usage for Boolean and State Configuration Variables"

If you plan on upgrading your software, be aware that:

- Setting the **Level** and **Priority** parameters enables the latest optimizations corresponding with the above parameter settings for each subsequent release.
- Selecting the **Specify custom optimizations** parameter enables you to select individual parameters in the **Details** section. When you load a model in a future release, optimization parameters that were introduced in releases after you adopted the software to when you upgrade are set to `off`. If you want to reduce the number of changes in the generated code when you upgrade your software, this option can be a good choice.

Command-Line Information

Parameter: `OptimizationLevel`
Value: `'level0'` | `'level1'`| `'level2'`
Default: `'level2'`

Mapping Between Command-Line Parameter Value and UI Parameter Setting

Command-Line Parameter Value	UI Parameter Setting
level0	`Minimum (debugging)`
level1	`Balanced with Readability`
level2	`Maximum`

Recommended Settings

Application	Setting
Debugging	`Minimum (debugging)`
Traceability	`Minimum (debugging)`
Efficiency	Based on your goals, choose `Balanced with Readability` or `Maximum`. If you choose `Maximum`, set the **Priority** parameter.
Safety precaution	No impact

See Also

"Priority" on page 10-16 | "Specify custom optimizations" on page 10-21

Related Examples

- "Model Configuration Parameters: Code Generation Optimization" on page 10-2

Priority

Description

Optimize the generated code for increased execution efficiency, decreased RAM consumption, or a balance between execution efficiency and RAM consumption.

Settings

Default: `Balance RAM and speed`

`Balance RAM and speed`

Configure code generation settings to balance RAM and execution speed.

`Maximize execution speed`

Apply code generation settings to maximize execution speed.

`Minimize RAM`

Configure code generation settings to minimize RAM consumption.

Dependencies

- Enable this parameter by setting the **Level** parameter to `Maximum`.
- This parameter requires an Embedded Coder license.
- This parameter appears only for ERT-based targets.

Tips

For each **Priority** and **Level** parameter value, there are corresponding values for the parameters in the **Details** section. These are some important differences among these various settings:

- If you set the **Level** parameter to `Minimum (debugging)`, the parameters in the **Details** section are set to off. The code generator does not implement optimizations that remove variables or code making it easier to debug the generated code.
- The parameter settings for `Balanced with Readability` and `Balance RAM and speed` are the same except for these three parameters:

  - **Reuse buffers of different sizes and dimensions**
  - **Optimize global data access**
  - **Optimize block operation order in the generated code**

  The above optimizations can potentially hurt readability because they cross atomic subsystem boundaries and **Optimize block operation order in the generated code** might change the block execution order in the generated code so that it is different than in simulation.
- If you have limited RAM, choose the `Minimize RAM` setting. This setting enables these optimizations that reduce RAM at the expense of a potential slow-down in execution speed:

  - **Pack Boolean data into bitfields**
  - **Reuse buffers of different sizes and dimensions**

- **Use bitsets for storing state configuration**
- **Use bitsets for storing Boolean data**

This setting also changes the **Optimize block operation order in the generated code** from `Improved Code Execution Speed` to `off`.

For each **Priority** and **Level** parameter value, this table lists the corresponding values for the parameters in the **Details** section.

Parameters	Settings					Example
Level	`Minimum (debugging)`	`Balanced with readability`	`Maximum`			
Priority	Not Applicable (N/A)	N/A	`Balance RAM and speed`	`Maximize execution speed`	`Minimize RAM`	
Details						
Use memcpy for vector assignment	`Off`	`On`	`On`	`On`	`On`	“Use memcpy Function to Optimize Generated Code for Vector Assignments”
Memcpy threshold (bytes)	`Off`	`64`	`64`	`64`	`64`	“Use memcpy Function to Optimize Generated Code for Vector Assignments”
Enable local block outputs	`Off`	`On`	`On`	`On`	`On`	“Enable and Reuse Local Block Outputs in Generated Code”
Reuse local block outputs	`Off`	`On`	`On`	`On`	`On`	“Enable and Reuse Local Block Outputs in Generated Code”

Parameters	Settings					Example
Eliminate superfluous local variables (expression folding)	`Off`	`On`	`On`	`On`	`On`	"Minimize Computations and Storage for Intermediate Results at Block Outputs"
Reuse global block outputs	`Off`	`On`	`On`	`On`	`On`	"Reuse Global Block Outputs in the Generated Code"
Perform in-place updates for Assignment and Bus Assignment blocks	`Off`	`On`	`on`	`On`	`On`	"Data copy reduction for Bus Assignment block"
Reuse buffers for Data Store Read and Data Store Write blocks	`Off`	`On`	`On`	`On`	`On`	"Data Copy Reduction for Data Store Read and Data Store Write Blocks"
Simplify array indexing	`Off`	`Off`	`Off`	`On`	`Off`	"Simplify Multiply Operations in Array Indexing"
Pack Boolean data into bitfields	`Off`	`Off`	`Off`	`Off`	`On`	"Optimize Generated Code By Packing Boolean Data Into Bitfields"
Reuse buffers of different sizes and dimensions	`Off`	`Off`	`On`	`Off`	`On`	"Reuse Buffers of Different Sizes and Dimensions"

Parameters	Settings					Example
Optimize global data access	None	None	Use global to hold temporary results	None	Use global to hold temporary results	"Optimize Global Variable Usage"
Optimize block operation order in the generated code	Off	Off	Improved Code Execution Speed	Improved Code Execution Speed	Off	"Remove Data Copies by Reordering Block Operations in the Generated Code"
Use bitsets for storing state configuration	Off	Off	Off	Off	On	"Reduce Memory Usage for Boolean and State Configuration Variables"
Use bitsets for storing Boolean data	Off	Off	Off	Off	On	"Reduce Memory Usage for Boolean and State Configuration Variables"

If you plan on upgrading your software, be aware that:

- Setting the **Level** and **Priority** parameters enables the latest optimizations corresponding with the above parameter settings for each subsequent release.
- Selecting the **Specify custom optimizations** parameter enables you to select individual parameters in the **Details** section. When you load a model in a future release, optimization parameters that were introduced in releases after you adopted the software to when you upgrade are set to `off`. If you want to reduce the number of changes in the generated code when you upgrade your software, this option can be a good choice.

Command-Line Information

Parameter: `OptimizationPriority`
Value: `'Balanced' | 'Speed'| 'RAM'`
Default: `'Balanced'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	`Minimize RAM` (RAM), `Maximum execution speed` (Speed), `Balance RAM and speed` (Balanced)
Safety precaution	No impact

See Also

"Level" on page 10-11 | "Specify custom optimizations" on page 10-21

Related Examples

- "Model Configuration Parameters: Code Generation Optimization" on page 10-2

Specify custom optimizations

Description

Select this parameter to enable the optimization parameters in the **Details** section. Selecting this parameter enables you to choose individual optimization parameters rather than setting the **Priority** and **Level** parameters.

Settings

Default: Off

On

Enables you to individually select or clear parameters in the **Details** section.

Off

Disables the parameters in the **Details** section, so that you cannot individually select or clear these parameters.

Clearing this parameter enables the **Level** and **Priority** parameters and resets the parameters in the **Details** section to the current settings for the **Level** and **Priority** parameters.

Dependencies

This parameter:

- Enables parameters in the **Details** section.
- Requires an Embedded Coder license.
- Appears only for ERT-based targets.

Tips

If you plan on upgrading your software, be aware that:

- Setting the **Priority** and **Level** parameters enables the latest optimizations corresponding with these settings for each subsequent release.
- Selecting **Specify custom optimizations** means that when you load a model in a future release, optimization parameters that were introduced in releases after you adopted the software to when you upgrade are set to `off`. If you want to reduce the number of changes in the generated code when you upgrade your software, this option can be a good choice.

Command-Line Information

Parameter: `OptimizationCustomize`
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	`Off` and set **Level** parameter to `Minimize (debugging)`
Traceability	`Off` and set **Level** parameter to `Minimize (debugging)`
Efficiency	Depends on individual parameter settings
Safety precaution	No impact

See Also

“Level” on page 10-11 | “Priority” on page 10-16

Related Examples

- “Model Configuration Parameters: Code Generation Optimization” on page 10-2

Reuse global block outputs

Description

Reuse global memory for block outputs.

Category: Optimization

Settings

Default: On

On

- Software reuses signal memory whenever possible, reducing global variable use.
- Selecting this parameter trades code traceability for code efficiency.

Off

Signals are stored in unique locations.

Dependencies

This parameter:

- Is enabled by “Signal storage reuse” .
- Requires an Embedded Coder license.
- Appears only for ERT-based targets.

Command-Line Information

Parameter: `GlobalBufferReuse`
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	Off
Traceability	Off
Efficiency	On (execution, ROM, RAM)
Safety precaution	No impact

See Also

Related Examples

- “Reuse Global Block Outputs in the Generated Code”
- “Performance”
- “Model Configuration Parameters: Code Generation Optimization” on page 10-2

Perform in-place updates for Assignment and Bus Assignment blocks

Description

Reuse the input and output variables of Bus Assignment and Assignment blocks if possible.

Category: Optimization

Settings

Default: On

On

Embedded Coder reuses the input and output variables of Bus Assignment and Assignment blocks if possible. Reusing these variables reduces data copies, conserves RAM consumption and increases code execution speed.

Off

Embedded Coder does not reuse the input and output variables of Bus Assignment and Assignment blocks.

Dependency

reusable subsystems

- The parameter **Signal Storage Reuse** enables this parameter.
- This parameter requires an Embedded Coder license.
- This parameter appears only for ERT-based targets.

Command-Line Information

Parameter: `BusAssignmentInplaceUpdate`
Value: `'on' | 'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Optimization” on page 10-2
- “Reduce Data Copies for Bus Assignment Blocks”

Reuse buffers for Data Store Read and Data Store Write blocks

Description

Remove temporary buffers for Data Store Read and Data Store Write blocks. Use the Data Store Memory block directly if possible.

Category: Optimization

Settings

Default: On

On

Embedded Coder reads directly from the Data Store Memory block and writes directly to the Data Store Memory block, if possible. Using the Data Store Memory block directly eliminates data copies in the generated code, conserving RAM consumption and increasing code execution speed.

Off

Embedded Coder inserts buffers in the generated code for Data Store Read and Data Store Write blocks.

Dependency

- The parameter **Signal Storage Reuse** enables this parameter.
- This parameter requires an Embedded Coder license.
- This parameter appears only for ERT-based targets.

Command-Line Information

Parameter: `OptimizeDataStoreBuffers`
Value: `'on' | 'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Off
Efficiency	On
Safety precaution	No recommendation

See Also

Related Examples

- “Data Copy Reduction for Data Store Read and Data Store Write Blocks”
- “Model Configuration Parameters: Code Generation Optimization” on page 10-2

Simplify array indexing

Description

Replace multiply operations in array indices when accessing arrays in a loop.

Category: Optimization

Settings

Default: Off

On

In array indices, replace multiply operations with add operations when accessing arrays in a loop in the generated code. When the original signal is multidimensional, the Embedded Coder generates one-dimensional arrays, resulting in multiply operations in the array indices. Using this setting eliminates costly multiply operations when accessing arrays in a loop in the C/C++ program. This optimization (commonly referred to as strength reduction) is particularly useful if the C/C++ compiler on the target platform does not have similar functionality. The absence of multiply operations in the C/C++ program does not imply that the C/C++ compiler does not generate multiply instructions.

Off

Leave multiply operations in array indices when accessing arrays in a loop.

Dependencies

This parameter:

- Requires a Embedded Coder license to generate code.
- Appears only for ERT-based targets.

Command-Line Information

Parameter: `StrengthReduction`
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	`On` (execution speed)
Safety precaution	No impact

See Also

Related Examples

- “Simplify Multiply Operations in Array Indexing”
- “Performance”
- “Model Configuration Parameters: Code Generation Optimization” on page 10-2

Pack Boolean data into bitfields

Description

Specify whether Boolean signals are stored as one–bit bitfields or as a Boolean data type.

Category: Optimization

Note You cannot use this optimization when you generate code for a target that specifies an explicit structure alignment.

Settings

Default: Off

On

Stores Boolean signals into one–bit bitfields in global block I/O structures or DWork vectors. This will reduce RAM, but might cause more executable code.

Off

Stores Boolean signals as a Boolean data type in global block I/O structures or DWork vectors.

Dependencies

This parameter:

- Requires a Embedded Coder license.
- Appears only for ERT-based targets.

Command-Line Information

Parameter: `BooleansAsBitfields`
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Off (execution, ROM), On (RAM)
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Optimization” on page 10-2
- “Optimize Generated Code By Packing Boolean Data Into Bitfields”
- “Bitfield declarator type specifier” on page 10-33
- “Performance”

Bitfield declarator type specifier

Description

Specify the bitfield type when selecting configuration parameter "Pack Boolean data into bitfields" on page 10-31.

Category: Optimization

Note The optimization benefit is dependent upon your choice of target.

Settings

Default: `uint_T`

☑ **uint_T**

The type specified for a bitfield declaration is an `unsigned int`.

☐ **uchar_T**

The type specified for a bitfield declaration is an `unsigned char`.

Tip

The "Pack Boolean data into bitfields" on page 10-31 configuration parameter default setting uses unsigned integers. This might cause an increase in RAM if the bitfields are small and distributed. In this case, `uchar_T` might use less RAM depending on your target.

Dependency

Pack Boolean data into bitfields enables this parameter.

Command-Line Information

Parameter: `BitfieldContainerType`
Value: `uint_T` | `uchar_T`
Default: `uint_T`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Target dependent
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Optimization” on page 10-2
- “Performance”

Reuse buffers of different sizes and dimensions

Reduce memory consumption by reusing buffers to store data of different sizes and dimensions.

Settings

Default: On

On

The code generator tries to reuse the same buffers to store data of different sizes and dimensions. This optimization conserves RAM and ROM consumption.

Off

The code generator reuses buffers only if they have the same size and shape as the data.

Dependencies

- This parameter appears only for ERT-based targets.
- When generating code, this parameter requires an Embedded Coder license.
- This parameter is enabled by “Signal storage reuse”.

Tips

- If your model contains a reusable custom storage class to specify reuse on signals that have different sizes and shapes, you must select the **Reuse buffers of different sizes and dimensions** parameter or remove the specification. Otherwise, during simulation, the model produces an error.
- The code generator does not replace a buffer with a lower priority buffer that has a smaller size.
- The code generator does not reuse buffers that have different sizes and symbolic dimensions.

Command-Line Information

Parameter: `DifferentSizesBufferReuse`
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	`off`
Traceability	`off`
Efficiency	`on`
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Optimization” on page 10-2
- “Reuse Buffers of Different Sizes and Dimensions”

Generate parallel for-loops

Description

Specify, whether for-loops in the generated code are implemented in parallel for the MATLAB Function block, the MATLAB System block, and the For Each Subsystem block with large input data sizes. The MATLAB Function block and MATLAB System block are executed in parallel when the MATLAB code contains `parfor`-loops instead of traditional `for`-loops.

Category: Optimization

Settings

Default: Off

☑ On

Executes OpenMP parallel for-loops by using multiple threads.

☐ Off

Executes for-loops using a single thread.

Dependency

This parameter requires that Embedded Coder and Parallel Computing Toolbox be installed.

Command-Line Information

Parameter: `MultiThreadedLoops`
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	Off
Traceability	Off
Efficiency	On (execution speed)
Safety precaution	No impact

See Also

Related Examples

- “Algorithm Acceleration Using Parallel for-Loops (parfor)”
- “Speed Up for-Loop Implementation in Code Generated by Using parfor”

Optimize global data access

Description

Select global variable optimization.

Category: Optimization

Settings

Default: `Use global to hold temporary results`

`None`

Use default optimizations.

`Use global to hold temporary results`

Maximize use of global variables.

`Minimize global data access`

Minimize use of global variables by using local variables to hold intermediate values.

Dependencies

- This parameter is enabled by **Signal storage reuse**.
- This parameter requires an Embedded Coder license.
- Appears only for ERT-based targets.

Command-Line Information

Parameter: `GlobalVariableUsage`
Value: `'None'` | `'Use global to hold temporary results'` | `'Minimize global data access'`
Default: `'None'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	`'Use global to hold temporary results'` (RAM), `'Minimize global data access'` (ROM)
Safety precaution	No impact

See Also

Related Examples

- “Optimize Global Variable Usage”
- “Performance”
- “Model Configuration Parameters: Code Generation Optimization” on page 10-2

Optimize block operation order in the generated code

Description

Reorder block operations in the generated code for improved code execution speed.

Category: Optimization

Settings

Default: Improved Code Execution Speed

Off

Embedded Coder does not reorder block operation order in the generated code to create additional instances of buffer reuse.

Improved Code Execution Speed

Embedded Coder changes the block operation order in the generated code so that more instances of buffer reuse can occur. Reusing buffers conserves RAM and ROM consumption and improves code execution speed.

Dependency

- This parameter requires an Embedded Coder license.
- This parameter appears only for ERT-based targets.

Command-Line Information

Parameter: `OptimizeBlockOrder`
Value: `'Speed' | 'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Improved Code Execution Speed
Safety precaution	No recommendation

See Also

Related Examples

- “Remove Data Copies by Reordering Block Operations in the Generated Code”
- “Improve Execution Efficiency by Reordering Block Operations in the Generated Code”

- “Model Configuration Parameters: Code Generation Optimization” on page 10-2

Optimize using the specified minimum and maximum values

Description

Optimize generated code using the specified minimum and maximum values for signals and parameters in the model.

Category: Optimization

Settings

Default: Off

On

Optimizes the generated code using range information derived from the minimum and maximum specified values for signals and parameters in the model.

Off

Ignores specified minimum and maximum values when generating code.

Tips

- To detect mismatches between model and generated code simulations that arise from the use of this parameter, before running normal, accelerator, software-in-the-loop (SIL), or processor-in-the-loop (PIL) simulations, set **Diagnostics > Data Validity > Simulation range checking** to `Warning` or `Error`.
- Specify minimum and maximum values for signals and parameters in the model for:
  - Inport and Outport blocks.
  - Block outputs.
  - Block inputs, for example, for the MATLAB Function and Stateflow Chart blocks.
  - `Simulink.Signal` objects.
- This optimization does not take into account minimum and maximum values specified for:
  - Merge block inputs. To work around this, use a `Simulink.Signal` object on the Merge block output and specify the range on this object
  - Bus elements.
  - Conditionally-executed subsystem (such as a triggered subsystem) block outputs that are directly connected to an Outport block.

    Outport blocks in conditionally-executed subsystems can have an initial value specified for use only when the system is not triggered. In this case, the optimization cannot use the range of the block output because the range might not cover the initial value of the block.
- If you use the Polyspace Code Prover™ software to verify code generated using this optimization, it might mark code that was previously green as orange. For example, if your model contains a division where the range of the denominator does not include zero, the generated code does not include protection against division by zero. Polyspace Code Prover might mark this code orange

because it does not have information about the minimum and maximum values specified for the inputs to the division.

The Polyspace Code Prover software does automatically capture some minimum and maximum values specified in the MATLAB workspace, for example, for `Simulink.Signal` and `Simulink.Parameter` objects. In this example, to provide range information to the Polyspace Code Prover software, use a `Simulink.Signal` object on the input of the division and specify a range that does not include zero.

The Polyspace Code Prover software stores these values in a Data Range Specification (DRS) file. However, they do not capture minimum and maximum values specified in your Simulink model. To provide additional min/max information to Polyspace Code Prover, you can manually define a DRS file. For more information, see the Polyspace Code Prover documentation.

- If you are using double-precision data types and the **Code Generation > Interface > Support non-finite numbers** configuration parameter is selected, this optimization does not occur.
- If your model contains multiple instances of a reusable subsystem and each instance uses input signals with different specified minimum and maximum values, this optimization might result in different generated code for each subsystem so code reuse does not occur. Without this optimization, the Simulink Coder software generates code once for the subsystem and shares this code among the multiple instances of the subsystem.
- The Model Advisor check Check safety-related optimization settings generates a warning if this option is selected. For many safety-critical applications, removing dead code automatically is unacceptable because doing so might make code untraceable.
- Enabling this optimization improves the ability of the Fixed-Point Designer software to eliminate unnecessary utility functions and saturation code from the generated code.

Dependencies

- This parameter appears for ERT-based targets only.
- This parameter requires a Embedded Coder license when generating code.

Command-Line Information

Parameter: `UseSpecifiedMinMax`
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	Off
Traceability	Off
Efficiency	On
Safety precaution	No impact

See Also

Related Examples

- “Optimize Generated Code Using Minimum and Maximum Values”
- “Optimize Generated Code Using Specified Minimum and Maximum Values” (Fixed-Point Designer)
- “Model Configuration Parameters: Code Generation Optimization” on page 10-2

Remove Code from Tunable Parameter Expressions That Saturate Out-of-Range Values

Description

Specify whether to generate code that saturates the values of tunable parameters expressions.

Category: Optimization

Settings

Default: On

On

Does not generate code that saturates the value of out-of-range tunable parameter expressions to prevent integer overflow. Select this check box if code efficiency is critical to your application.

Off

Generates code that guards against overflow by saturating the value of tunable parameter expressions in between upper and lower limits.

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires a Embedded Coder license when generating code.

Command-Line Information

Parameter: `EfficientTunableParamExpr`
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	Off
Traceability	Off
Efficiency	On (execution, ROM)
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Optimization” on page 10-2
- “Performance”

- “Remove Code from Tunable Parameter Expressions That Saturate Against Integer Overflow”

Remove code that protects against division arithmetic exceptions

Description

Specify whether to generate code that guards against division by zero and `INT_MIN/-1` operations for integers and fixed-point data.

Category: Optimization

Settings

Default: Off

On

Does not generate code that guards against division by zero and `INT_MIN/-1` operations for integers and fixed-point data. To retain bit-true agreement between simulation results and results from generated code, check that your model does not produce division by zero or `INT_MIN/-1` operations, where the quotient cannot be represented in the data type.

Off

Generates code that guards against division by zero and `INT_MIN/-1` operations for integers and fixed-point data.

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires a Embedded Coder license when generating code.

Command-Line Information

Parameter: `NoFixptDivByZeroProtection`
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On (execution, ROM)
Safety precaution	Off

See Also

Related Examples

- “Remove Code That Guards Against Division Exceptions for Integers and Fixed-Point Data”
- “Model Configuration Parameters: Code Generation Optimization” on page 10-2
- “Performance”

Use signal labels to guide buffer reuse

Description

For signals with the same label, the code generator attempts to use the same signal memory.

Category: Optimization

Settings

Default: Off

☑ On

The code generator uses signal labels as a guide for which buffers to reuse.

☐ Off

The code generator ignores signal labels when implementing buffer reuse.

Dependencies

This parameter:

- Requires an Embedded Coder license.
- Appears only for ERT-based targets.

Tips

If your model has the optimal parameter settings for removing data copies, you might be able to remove additional data copies by using signal labels. After studying the generated code and the Static Code Metrics Report and identifying blocks for whose input and output signals you would like to reuse, you can add labels to signal lines and enable the **Use signal labels to guide buffer reuse** parameter. If possible, the code generator implements the reuse specification.

Command-Line Information

Parameter:`LabelGuidedReuse`
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	Off
Traceability	Off
Efficiency	On (execution, ROM, RAM)
Safety precaution	No impact

See Also

Related Examples

- “Optimize Generated Code by Using Signal Labels to Guide Buffer Reuse”
- “Model Configuration Parameters: Code Generation Optimization”

Operator to represent Bitwise and Logical Operator blocks

Description

Optimize the generated code with Logical or Bitwise operators or a combination of both. The supported data type is Boolean.

Settings

Default: `Same as modeled`

`Same as modeled`

Generate code with operators as modeled in Simulink. Generates code with a combination of Bitwise and Logical operators if the model contains both Logical and Bitwise blocks.

`Logical Operator`

Generate code with Logical operators. Converts Bitwise operator blocks in the model to Logical operators in the generated code.

`Bitwise Operator`

Generate code with Bitwise operators. Converts Logical operators blocks in the model to Bitwise operators in the generated code. Selecting this option may improve ROM efficiency when the code contains Bitwise operators.

Dependencies

- This parameter requires Embedded Coder.
- This parameter appears only for ERT-based targets.

Command-Line Information

Parameter: `BitwiseOrLogicalOp`
Value: `'Same as modeled'` | `'Logical operator'` | `'Bitwise operator'`
Default: `'Same as modeled'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	`Bitwise operator` (ROM efficiency)
Safety precaution	No impact

See Also

“Level” on page 10-11 | “Specify custom optimizations” on page 10-21

Related Examples

- “Model Configuration Parameters: Code Generation Optimization” on page 10-2
- “Control Operator Type in Generated Code”

Code Generation Parameters: Comments

- “Model Configuration Parameters: Comments” on page 11-2
- “Trace to model using” on page 11-5
- “Operator annotations” on page 11-7
- “Simulink block descriptions” on page 11-9
- “Simulink data object descriptions” on page 11-11
- “Custom comments (MPT objects only)” on page 11-13
- “Custom comments function” on page 11-15
- “Stateflow object descriptions” on page 11-17
- “Requirements in block comments” on page 11-19
- “MATLAB user comments” on page 11-21
- “Comment style” on page 11-22
- “Insert Polyspace comments” on page 11-24

Model Configuration Parameters: Comments

The **Code Generation > Comments** category includes parameters for configuring the comments in the generated code. These parameters require a Simulink Coder license. Additional parameters available with an ERT-based target require an Embedded Coder license.

Code Comments are generated automatically or you can add them to the code.

Code comments have the following uses:

- Enhance the readability and traceability of code
- Convey information among users
- Enhance code search in code base

Code Comments can be classified into Auto generated and Custom comments. Auto generated comments are automatically generated by the software during code generation and the user adds Custom comments.

On the Configuration Parameters dialog box, the following configuration parameters are on the **Code Generation > Comments** pane.

Auto generated comments

Parameter	Description
“Include comments”	Specify which comments are in generated files.
“Simulink block comments”	Specify whether to insert Simulink block comments.
“Trace to model using” on page 11-5	Specify format of comments for Simulink blocks, Stateflow elements and MATLAB function blocks.
“Stateflow object comments”	Specify whether to insert Stateflow object comments.
“MATLAB source code as comments”	Specify whether to insert MATLAB source code as comments.
“Show eliminated blocks”	Specify whether to insert eliminated block's comments.
“Verbose comments for 'Model default' storage class”	Reduce code size or improve code traceability by controlling the generation of comments.
“Operator annotations” on page 11-7	Specify whether to include operator annotations for Polyspace in the generated code as comments.

Custom comments

Parameter	Description
“Simulink block descriptions” on page 11-9	Specify whether to insert descriptions of blocks into generated code as comments.

Parameter	Description
"Stateflow object descriptions" on page 11-17	Specify whether to insert descriptions of Stateflow objects into generated code as comments.
"Simulink data object descriptions" on page 11-11	Specify whether to insert descriptions of Simulink data objects into generated code as comments.
"Requirements in block comments" on page 11-19	Specify whether to include requirement descriptions assigned to Simulink blocks in generated code as comments.
"Custom comments (MPT objects only)" on page 11-13	Specify whether to include custom comments for module packaging tool (MPT) signal and parameter data objects in generated code.
"MATLAB user comments" on page 11-21	Specify whether to include MATLAB user comments as comments.
"Custom comments function" on page 11-15	Specify a file that contains comments to be included in generated code for module packing tool (MPT) signal and parameter data objects.

The following configuration parameters are under the **Advanced parameters**.

Parameter	Description
"Comment style" on page 11-22	Specify a multi-line or single-line comment style for generated C or C++ code.
"Insert Polyspace comments" on page 11-24	Specify whether to insert code comments for Polyspace block annotations.

- The code generation software automatically inserts comments into the generated code for custom blocks. Therefore, you do not need to include block comments in the associated TLC file for a custom block.

  Note If you have existing TLC files with manually inserted comments for block descriptions, the code generation process emits these comments instead of the automatically generated comments. Consider removing existing block comments from your TLC files. Manually inserted comments might be poorly formatted in the generated code and code-to-model traceability might not work.

- For virtual blocks or blocks that have been removed due to block reduction, comments are not generated.
- When you configure the code generator to produce code that includes comments, the code generator includes text for model parameters, block names, signal names, and Stateflow object names in the generated code comments. If the text includes characters that are unrepresented in the character set encoding for the model, the code generator replaces the characters with XML escape sequences. For example, the code generator replaces the Japanese full-width Katakana letter ア with the escape sequence `ア`. For more information, see "Internationalization and Code Generation".
- When you set the model configuration parameter **Default parameter behavior** to `Tunable`, the code generator adds different comments about numeric block parameters in the generated code depending on the numeric value of the block parameter and the output data type. For instance, code generator adds:

- `Computed Parameter` as a comment when the numeric value of the block parameter needs type conversion to match the output data type.
- `Expression` as a comment when the numeric value of the block parameter matches the output data type without type conversion.

Simulink interprets the data type of a numeric parameter as double unless you explicitly specify otherwise. Generate code for the following model:

```
// Parameters (auto storage)
struct P_test_parameter_T_ {
  real_T Constant1_Value;              // Expression: 200
                                          //  Referenced by: '<Root>/Constant1'

  real_T Constant2_Value;              // Computed Parameter: Constant2_Value
                                          //  Referenced by: '<Root>/Constant2'

  int32_T Constant3_Value;             // Computed Parameter: Constant3_Value
                                          //  Referenced by: '<Root>/Constant3'
};
```

When the constant value is 200 and the output data type is double, the code generator adds `Expression` as a comment. Simulink interprets the data type of the constant value as double and without type conversion it matches the output data type.

When the constant value is uint8(200) and the output data type is double, the code generator adds `Computed Parameter` as a comment. The constant value requires type conversion to match the output data type.

When the constant value is 500 and the output data type is int32, the code generator adds `Computed Parameter` as a comment. The constant value requires type conversion to match the output data type.

See Also

More About

- "Configure Code Comments"
- "Verify Generated Code by Using Code Tracing"

Trace to model using

Description

Specify format of comments for Simulink blocks, Stateflow elements, and MATLAB function blocks.

Category: Code Generation > Comments

Settings

Default: `Block path`

Block path

The generated comment includes the entire block path from the root as the traceability link. For example:

```
/* Outport: '<Root>/Out1' */
rtwdemo_comments_Y.Out1 = 1;
```

Simulink identifier

The generated comment includes the Simulink identifier without the model name for the corresponding block or object. For example:

```
/* Outport: 'Out1' (':33') */
rtwdemo_comments_Y.Out1 = 1;
```

Dependency

- This parameter requires Embedded Coder.
- **Simulink block comments** or **Stateflow object comments** enable this parameter.

Command-Line Information

Parameter: `BlockCommentType`
Type: character vector
Value: `'Block path'` | `'Simulink identifier'`
Default: `'Block path'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Comments” on page 11-2
- “Simulink Identifiers”

Operator annotations

Description

Specify whether to include operator annotations for Polyspace in the generated code as comments.

Category: Code Generation > Comments

Settings

Default: On

On

Includes operator annotations in the generated code.

Off

Does not include operator annotations in the generated code.

Tips

- These annotations help document overflow behavior that is due to the way the code generator implements an operation. These operators cannot be traced to an overflow in the design.
- Justify operators that the Polyspace software cannot prove. When this option is enabled, if the code generator uses one of these operators, it adds annotations to the generated code to justify the operators for Polyspace.
- The code generator cannot justify operators that result from the design.

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- **Include comments** enables this parameter.

Command-Line Information

Parameter: `OperatorAnnotations`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	On
Efficiency	No impact

Application	Setting
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Comments” on page 11-2
- “Annotate Code for Justifying Polyspace Checks”

Simulink block descriptions

Description

Specify whether to insert descriptions of blocks into generated code as comments.

Category: Code Generation > Comments

Settings

Default: On

On

Includes the following comments in the generated code for each block in the model, with the exception of virtual blocks and blocks removed due to block reduction:

- The block name at the start of the code, regardless of whether you select **Simulink block comments**
- Text specified in the **Description** field of each Block Properties dialog box

For information on code generator treatment of strings that are unrepresented in the character set encoding for the model, see “Internationalization and Code Generation”.

Off

Suppresses the generation of block name and description comments in the generated code.

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: `InsertBlockDesc`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Noimpact
Safety precaution	Noimpact

See Also

Related Examples

- “Model Configuration Parameters: Comments” on page 11-2
- “Internationalization and Code Generation”

Simulink data object descriptions

Description

Specify whether to insert descriptions of Simulink data objects into generated code as comments.

This parameter does not affect `Simulink.LookupTable` or `Simulink.Breakpoint` objects that you configure to appear in the generated code as a structure (for example, by storing all of the table and breakpoint data in a single `Simulink.LookupTable` object).

Category: Code Generation > Comments

Settings

Default: On

On

Inserts contents of the **Description** field in the Model Explorer Object Properties pane for each Simulink data object (signal, parameter, and bus objects) in the generated code as comments.

For information on code generator treatment of strings that are unrepresented in the character set encoding for the model, see “Internationalization and Code Generation”.

Off

Suppresses the generation of data object property descriptions as comments in the generated code.

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: `SimulinkDataObjDesc`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Noimpact
Safety precaution	Noimpact

See Also

Related Examples

- "Model Configuration Parameters: Comments" on page 11-2

Custom comments (MPT objects only)

Description

Specify whether to include custom comments for module packaging tool (MPT) signal and parameter data objects in generated code. MPT data objects are objects of the classes `mpt.Parameter` and `mpt.Signal`.

Category: Code Generation > Comments

Settings

Default: Off

On

Inserts comments just above the identifiers for signal and parameter MPT objects in generated code.

Off

Suppresses the generation of custom comments for signal and parameter identifiers.

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter requires that you include the comments in a function defined in a MATLAB language file or TLC file that you specify with **Custom comments function**.

Command-Line Information

Parameter: `EnableCustomComments`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Noimpact
Safety precaution	Noimpact

See Also

Related Examples

- “Model Configuration Parameters: Comments” on page 11-2
- “Add Custom Comments for Variables in the Generated Code”

Custom comments function

Description

Specify a file that contains comments to be included in generated code for module packing tool (MPT) signal and parameter data objects. MPT data objects are objects of the classes `mpt.Parameter` and `mpt.Signal`.

Category: Code Generation > Comments

Settings

Default: `''`

Enter the name of the MATLAB language file or TLC file for the function that includes the comments to be inserted of your MPT signal and parameter objects. You can specify the file name directly or click **Browse** and search for a file.

Tip

You might use this option to insert comments that document some or all of the property values of an object.

For an example MATLAB function, see the function *`matlabroot`*`/toolbox/rtw/rtwdemos/rtwdemo_comments_mptfun.m`.

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- **Custom comments (MPT objects only)** enables this parameter.

Command-Line Information

Parameter: `CustomCommentsFcn`
Type: character vector
Value: valid file name
Default: `''`

Recommended Settings

Application	Setting
Debugging	Valid file name
Traceability	Valid file name
Efficiency	Noimpact
Safety precaution	Noimpact

See Also

Related Examples

- “Model Configuration Parameters: Comments” on page 11-2
- “Add Custom Comments for Variables in the Generated Code”

Stateflow object descriptions

Description

Specify whether to insert descriptions of Stateflow objects into generated code as comments.

Category: Code Generation > Comments

Settings

Default: On

On

Inserts descriptions of Stateflow states, charts, transitions, and graphical functions into generated code as comments. The descriptions come from the **Description** field in Object Properties pane in the Model Explorer for these Stateflow objects. The comments appear just above the code generated for each object.

For information on code generator treatment of strings that are unrepresented in the character set encoding for the model, see “Internationalization and Code Generation”.

Off

Suppresses the generation of comments for Stateflow objects.

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires a Stateflow license.

Command-Line Information

Parameter: `SFDataObjDesc`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Noimpact
Safety precaution	Noimpact

See Also

Related Examples

- “Model Configuration Parameters: Comments” on page 11-2
- “Internationalization and Code Generation”

Requirements in block comments

Description

Specify whether to include requirement descriptions assigned to Simulink blocks in generated code as comments.

Category: Code Generation > Comments

Settings

Default: Off

On

Inserts the requirement descriptions that you assign to Simulink blocks into the generated code as comments. The code generator includes the requirement descriptions in the generated code in the following locations.

Model Element	Requirement Description Location
Model	In the main header file *model*.h
Nonvirtual subsystems	At the call site for the subsystem
Virtual subsystems	At the call site of the closest nonvirtual parent subsystem. If a virtual subsystem does not have a nonvirtual parent, requirement descriptions are located in the main header file for the model, *model*.h.
Nonsubsystem blocks	In the generated code for the block

For information on code generator treatment of strings that are unrepresented in the character set encoding for the model, see “Internationalization and Code Generation”.

Off

Suppresses the generation of comments for block requirement descriptions.

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires Embedded Coder and Simulink Requirements™ licenses when generating code.

Tips

If you use an external `.slmx` file to store your requirement links, to avoid stale comments in generated code, before code generation, you must save any change in your requirement links.

Command-Line Information

Parameter: `ReqsInCode`
Type: character vector

Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Noimpact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Comments” on page 11-2
- “How Requirements Information Is Included in Generated Code” (Simulink Requirements)

MATLAB user comments

Description

Specify whether to include MATLAB user comments including both function description comments and other user comments from MATLAB code as comments in the generated code.

Category: Code Generation > Comments

Settings

Default: Off

☑ On

Inserts MATLAB user comments as comments.

☐ Off

Suppresses comments.

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- **Include comments** enables this parameter.

Command-Line Information

Parameter: `MATLABFcnDesc`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Noimpact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Comments” on page 11-2
- “Include MATLAB Code as Comments in the Generated Code”

Comment style

Description

Specify comment style in the generated C/C++ code.

Category: Code Generation > Comments

Settings

Default: `Auto`

`Auto`

For C code, generate single- or multiple-line comments delimited by `/*` and `*/`. For C++ code, generate single-line comments preceded by `//`.

`Multi-line`

Generate single- or multiple-line comments delimited by `/*` and `*/`.

Example of code generated by using the multiline comment style is:

```
/* Sum: '<Root>/Sum' incorporates:
 *  Constant: '<Root>/INC'
 *  UnitDelay: '<Root>/X'
 */
rtDW.X_g++;
```

`Single-line`

Generate single-line comments preceded by `//`.

Example of code generated by using the single-line comment style is:

```
// Sum: '<Root>/Sum' incorporates:
//   Constant: '<Root>/INC'
//   UnitDelay: '<Root>/X'

rtDW.X_g++;
```

Note For C code generation, select `Single-line` only if your compiler supports it.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Command-Line Information

Parameter: `CommentStyle`
Type: character vector

Value: `Auto` | `Multi-line` | `Single-line`
Default: `Auto`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Comments”

Insert Polyspace comments

Description

Specify whether to insert code comments for Polyspace block annotations.

Category: Code Generation > Comments

Settings

Default: Off

☑ On

Generate code comments for Polyspace block annotations.

☐ Off

Suppresses the generation of comments for Polyspace block annotations.

Dependency

- This parameter only appears for ERT-based targets.
- **Include comments** enables this parameter.

Command-Line Information

Parameter: `InsertPolySpaceComments`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- "Model Configuration Parameters: Comments" on page 11-2

Code Generation Parameters: Report

- “Model Configuration Parameters: Code Generation Report” on page 12-2
- “Generate static code metrics” on page 12-4
- “Code-to-model” on page 12-5
- “Model-to-code” on page 12-7
- “Configure” on page 12-9
- “Eliminated / virtual blocks” on page 12-10
- “Traceable Simulink blocks” on page 12-12
- “Traceable Stateflow objects” on page 12-14
- “Traceable MATLAB functions” on page 12-16
- “Summarize which blocks triggered code replacements” on page 12-18
- “Generate model Web view” on page 12-20

Model Configuration Parameters: Code Generation Report

The **Code Generation** > **Report** category includes parameters for generating and customizing the code generation report. These parameters require a Simulink Coder license. Additional parameters available with an ERT-based target require an Embedded Coder license.

On the Configuration Parameters dialog box, the following configuration parameters are on the **Code Generation** > **Report** pane.

Parameter	Description
“Create code generation report”	Document generated code in an HTML report.
“Open report automatically”	Specify whether to display code generation reports automatically.
“Generate model Web view” on page 12-20	Include the model Web view in the code generation report to navigate between the code and model within the same window.
“Generate static code metrics” on page 12-4	Include static code metrics report in the code generation report.

These configuration parameters are under the **Advanced parameters**.

Parameter	Description
“Code-to-model” on page 12-5	Include hyperlinks in the code generation report that link code to the corresponding Simulink blocks, Stateflow objects, and MATLAB functions in the model diagram.
“Model-to-code” on page 12-7	Link Simulink blocks, Stateflow objects, and MATLAB functions in a model diagram to corresponding code segments in a generated HTML report so that the generated code for a block can be highlighted on request.
“Configure” on page 12-9	Open the **Model-to-code navigation** dialog box for specifying a build folder containing previously-generated model code to highlight.
“Eliminated / virtual blocks” on page 12-10	Include summary of eliminated and virtual blocks in code generation report.
“Traceable Simulink blocks” on page 12-12	Include summary of Simulink blocks in code generation report.
“Traceable Stateflow objects” on page 12-14	Include summary of Stateflow objects in code generation report.
“Traceable MATLAB functions” on page 12-16	Include summary of MATLAB functions in code generation report.
“Summarize which blocks triggered code replacements” on page 12-18	Include code replacement report summarizing replacement functions used and their associated blocks in the code generation report.

See Also

More About

- “Report Generation”
- “Model Configuration”

Generate static code metrics

Description

Generate static code metrics, which can be viewed in the Code view or in a static code metrics report.

Category: Code Generation > Report

Settings

Default: Off

☑ On

Generate static code metrics.

☐ Off

Omit static code metrics.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: `GenerateCodeMetricsReport`
Type: Boolean
Value: `on` | `off`
Default: `off`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Report”
- “Static Code Metrics”

Code-to-model

Description

Include hyperlinks in the code generation report that link code to the corresponding Simulink blocks, Stateflow objects, and MATLAB functions in the model diagram.

Category: Code Generation > Report

Settings

Default: Off

On

Includes hyperlinks in the code generation report that link code to corresponding Simulink blocks, Stateflow objects, and MATLAB functions in the model diagram. The hyperlinks provide traceability for validating generated code against the source model.

Off

Omits hyperlinks from the generated report.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled by **Create code generation report**.
- You must select **Include comments** on the **Code Generation** > **Comments** pane to use this parameter.

Command-Line Information

Parameter: `IncludeHyperlinkInReport`
Type: character vector
Value: `'on'` | `'off`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Report”
- “HTML Code Generation Report Extensions”

Model-to-code

Description

Link Simulink blocks, Stateflow objects, and MATLAB functions in a model diagram to corresponding code segments in a generated HTML report so that the generated code for a block can be highlighted on request.

Category: Code Generation > Report

Settings

Default: Off

On

Includes model-to-code highlighting support in the code generation report. To highlight the generated code for a Simulink block, Stateflow object, or MATLAB script in the code generation report, right-click the item and select **C/C++ Code** > **Navigate to C/C++ Code**.

Off

Omits model-to-code highlighting support from the generated report.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled when you select **Create code generation report**.
- You must select the following parameters to use this parameter:
  - **Include comments** on the **Code Generation** > **Comments** pane
  - At least one of the following:
    - **Eliminated / virtual blocks** on page 12-10
    - **Traceable Simulink blocks** on page 12-12
    - **Traceable Stateflow objects** on page 12-14
    - **Traceable MATLAB functions** on page 12-16

Command-Line Information

Parameter: `GenerateTraceInfo`
Type: Boolean
Value: `on` | `off`
Default: `off`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Report”
- “HTML Code Generation Report Extensions”

Configure

Description

Open the Model-to-code navigation dialog box. Through this dialog box you specify a build folder containing previously generated model code to highlight. When you apply your build folder selection, Embedded Coder attempts to load traceability information from the earlier build, if you selected **Model-to-code** on page 12-7 parameter.

Category: Code Generation > Report

Dependency

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.
- Is enabled by the **Model-to-code** parameter.

See Also

Related Examples

- “Reload Existing Traceability Information”
- “Model Configuration Parameters: Code Generation Report”

Eliminated / virtual blocks

Description

Include summary of eliminated and virtual blocks in the **Traceability Report** section of the code generation report.

Category: Code Generation > Report

Settings

Default: Off

On

Include summary of eliminated and virtual blocks in the **Traceability Report** section of the code generation report.

Off

Does not include a summary of eliminated and virtual blocks.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.
- Is enabled by the **Create code generation report** parameter.

Command-Line Information

Parameter: `GenerateTraceReport`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Report”
- “HTML Code Generation Report Extensions”
- “Customize Traceability Reports”

Traceable Simulink blocks

Description

Category: Code Generation > Report

Includes a summary of Simulink blocks and the corresponding code locations in the **Traceability Report** section of the code generation report.

Settings

Default: Off

On

Includes a summary of Simulink blocks and the corresponding code locations in the **Traceability Report** section of the code generation report.

Off

Does not include a summary of Simulink blocks.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.
- Is enabled by the **Create code generation report** parameter.

Command-Line Information

Parameter: `GenerateTraceReportSl`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Report”
- “HTML Code Generation Report Extensions”
- “Customize Traceability Reports”

Traceable Stateflow objects

Description

Includes a summary of Stateflow objects and the corresponding code locations in the **Traceability Report** section of the code generation report.

Category: Code Generation > Report

Settings

Default: Off

On

Includes a summary of Stateflow objects and the corresponding code locations in the **Traceability Report** section of the code generation report.

Off

Does not include a summary of Stateflow objects.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.
- Is enabled by the **Create code generation report** parameter.

Command-Line Information

Parameter: `GenerateTraceReportSf`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Report”
- “HTML Code Generation Report Extensions”
- “Customize Traceability Reports”

Traceable MATLAB functions

Description

Includes a summary of MATLAB functions and corresponding code locations in the **Traceability Report** section of the code generation report.

Category: Code Generation > Report

Settings

Default: Off

On

Includes a summary of MATLAB functions and corresponding code locations in the **Traceability Report** section of the code generation report.

Off

Does not include a summary of MATLAB functions.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.
- Is enabled by the **Create code generation report** parameter.

Command-Line Information

Parameter: `GenerateTraceReportEml`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Report”
- “HTML Code Generation Report Extensions”
- “Customize Traceability Reports”

Summarize which blocks triggered code replacements

Description

Include code replacement report summarizing replacement functions used and their associated blocks in the code generation report.

Category: Code Generation > Report

Settings

Default: Off

On

Include code replacement report in the code generation report.

Note Selecting this option also generates code replacement trace information for viewing in the **Trace Information** tab of the Code Replacement Viewer. The generated information can help you determine why an expected code replacement did not occur.

Off

Omit code replacement report from the code generation report.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled when you select **Create code generation report**.

Command-Line Information

Parameter: `GenerateCodeReplacementReport`
Type: Boolean
Value: `on` | `off`
Default: `off`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Report”
- Analyze Code Replacements in the Generated Code
- “Verify Code Replacement Library”
- Determine Why Code Replacement Functions Were Not Used

Generate model Web view

Description

Include the model Web view in the code generation report to navigate between the code and model within the same window. You can share your model and generated code outside of the MATLAB environment. You must have a Simulink Report Generator™ license to include a Web view (Simulink Report Generator) of the model in the code generation report.

Category: Code Generation > Report

Settings

Default: Off

On

Include model Web view in the code generation report.

Off

Omit model Web view in the code generation report.

Dependencies

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled by **Create code generation report**.
- To enable traceability between the code and model, select **Code-to-model** and **Model-to-code**.

Command-Line Information

Parameter: `GenerateWebview`
Type: string
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

"Model Configuration Parameters: Code Generation Report"

Related Examples

- “Web View of Model in Code Generation Report”

13

Code Generation Parameters: Custom Code

Model Configuration Parameters: Code Generation Custom Code

The **Code Generation** > **Custom Code** category includes parameters for inserting custom C code into the generated code. These parameters require a Simulink Coder license.

On the Configuration Parameters dialog box, the following configuration parameters are on the **Code Generation** > **Custom Code** pane.

Parameter	Description
"Use the same custom code settings as Simulation Target"	Specify whether to use the same custom code settings as those in the **Simulation Target** > **Custom Code** pane.
"Source file"	Specify custom code to include near the top of the generated model source file.
"Header file"	Specify custom code to include near the top of the generated model header file.
"Initialize function"	Specify custom code to include in the generated model initialize function.
"Terminate function"	Specify custom code to include in the generated model terminate function.
"Include directories"	Specify a list of include folders to add to the include path.
"Source files"	Specify a list of additional source files to compile and link with the generated code.
"Libraries"	Specify a list of additional libraries to link with the generated code.
"Defines"	Specify preprocessor macro definitions to be added to the compiler command line.

See Also

More About

- "Model Configuration"

Model Configuration Parameters: Code Generation Interface

- “Model Configuration Parameters: Code Generation Interface” on page 14-2
- “Support: floating-point numbers” on page 14-8
- “Support: complex numbers” on page 14-10
- “Support: absolute time” on page 14-11
- “Support: continuous time” on page 14-13
- “Support: variable-size signals” on page 14-15
- “Pass root-level I/O as” on page 14-16
- “Remove error status field in real-time model data structure” on page 14-18
- “Include model types in model class” on page 14-20
- “Ignore custom storage classes” on page 14-22
- “Ignore test point signals” on page 14-24
- “Support non-inlined S-functions” on page 14-26
- “Use dynamic memory allocation for model initialization” on page 14-28
- “Use dynamic memory allocation for model block instantiation” on page 14-30
- “Terminate function required” on page 14-32
- “Combine signal/state structures” on page 14-33
- “Implement each data store block as a unique access point” on page 14-35
- “Generate separate internal data per entry-point function” on page 14-36
- “Multiword type definitions” on page 14-39
- “Generate destructor” on page 14-41
- “MAT-file variable name modifier” on page 14-42
- “Existing shared code” on page 14-43

Model Configuration Parameters: Code Generation Interface

The **Code Generation > Interface** category includes parameters for configuring the interface of the generated code. These parameters require a Simulink Coder license. Additional parameters available with an ERT-based target require an Embedded Coder license. Generating code for deep learning models using NVIDIA CUDA deep neural network library (cuDNN) or TensorRT high performance inference libraries for NVIDIA GPUs requires a GPU Coder license.

On the Configuration Parameters dialog box, the following configuration parameters are on the **Code Generation > Interface** pane.

Parameter	Description
"Code replacement library"	Specify a code replacement library the code generator uses when producing code for a model.
"Code replacement libraries"	Specify multiple code replacement libraries the code generator use when producing code for a model.
"Shared code placement"	Specify the location for generating utility functions, exported data type definitions, and declarations of exported data with custom storage class.
"Support: floating-point numbers" on page 14-8	Specify whether to generate floating-point data and operations.
"Support: non-finite numbers"	Specify whether to generate non-finite data and operations on non-finite data.
"Support: complex numbers" on page 14-10	Specify whether to generate complex data and operations.
"Support: absolute time" on page 14-11	Specify whether to generate and maintain integer counters for absolute and elapsed time values.
"Support: continuous time" on page 14-13	Specify whether to generate code for blocks that use continuous time.
"Support: variable-size signals" on page 14-15	Specify whether to generate code for models that use variable-size signals.
"Code interface packaging"	Select the packaging for the generated C or C++ code interface.
"Multi-instance code error diagnostic"	Select the severity level for diagnostics displayed when a model violates requirements for generating multi-instance code.
"Pass root-level I/O as" on page 14-16	Control how root-level model input and output are passed to the reusable `model_step` function.
"Remove error status field in real-time model data structure" on page 14-18	Specify whether to log or monitor error status.
"Include model types in model class" on page 14-20	Specify whether to generate model type definitions in a model class.
"Array layout"	Specify layout of array data for code generation as column-major or row-major

Parameter	Description
"External functions compatibility for row-major code generation"	Select diagnostic action if Simulink encounters a function that has no specified array layout
"Generate C API for: signals"	Generate C API data interface code with a signals structure.
"Generate C API for: parameters"	Generate C API data interface code with parameter tuning structures.
"Generate C API for: states"	Generate C API data interface code with a states structure.
"Generate C API for: root-level I/O"	Generate C API data interface code with a root-level I/O structure.
"ASAP2 interface"	Generate code for the ASAP2 data interface.
"External mode"	Generate code for the external mode data interface.
"Transport layer"	Specify the transport protocol for communications.
"MEX-file arguments"	Specify arguments to pass to an external mode interface MEX-file for communicating with executing targets.
"Static memory allocation"	Control memory buffer for external mode communication.
"Static memory buffer size"	Specify the memory buffer size for external mode communication.
"Target library"	Specify the target deep learning library used during code generation. `cuDNN` or `TensorRT` requires a GPU Coder license.
"ARM Compute Library version"	Specify the version of ARM® Compute Library.
"ARM Compute Library architecture"	Specify the ARM architecture supported in the target hardware.
"Auto tuning"	Use auto tuning for cuDNN library. Enabling auto tuning allows the cuDNN library to find the fastest convolution algorithms. This parameter requires a GPU Coder license.

These configuration parameters are under the **Advanced parameters**.

Parameter	Description
“Standard math library”	Specify the standard math library for your execution environment. Verify that your compiler supports the library you want to use; otherwise compile-time errors can occur. `C89/C90 (ANSI)` - ISO/IEC 9899:1990 C standard math library `C99 (ISO)` - ISO/IEC 9899:1999 C standard math library `C++03 (ISO)` - ISO/IEC 14882:2003 C++ standard math library
“Support non-inlined S-functions” on page 14-26	Specify whether to generate code for non-inlined S-functions.
“Maximum word length”	Specify a maximum word length, in bits, for which the code generation process generates system-defined multiword type definitions.
“Buffer size of dynamically-sized string (bytes)”	Number of bytes of the character buffer generated for dynamic string signals without maximum length.
“Multiword type definitions” on page 14-39	Specify whether to use system-defined or user-defined type definitions for multiword data types in generated code.
“Classic call interface”	Specify whether to generate model function calls compatible with the main program module of the GRT target in models created before R2012a.
“Use dynamic memory allocation for model initialization” on page 14-28	Control how the generated code allocates memory for model data.
“Single output/update function”	Specify whether to generate the `model_step` function.
“Terminate function required” on page 14-32	Specify whether to generate the `model_terminate` function.
“Combine signal/state structures” on page 14-33	Specify whether to combine global block signals and global state data into one data structure in the generated code
“Generate separate internal data per entry-point function” on page 14-36	Generate a model's block signals (block I/O) and discrete states (DWork) acting at the same rate into the same data structure.
“MAT-file logging”	Specify MAT-file logging.
“MAT-file variable name modifier” on page 14-42	Select the text to add to MAT-file variable names.
“Existing shared code” on page 14-43	Specify folder that contains existing shared code
“Remove disable function” on page 5-6	Remove unreachable (dead-code) instances of the `disable` functions from the generated code for ERT-based systems that include model referencing hierarchies.

Parameter	Description
"Remove reset function" on page 5-5	Remove unreachable (dead-code) instances of the `reset` functions from the generated code for ERT-based systems that include model referencing hierarchies.
"LUT object struct order for even spacing specification"	Change the order of the fields in the generated structure for a lookup table object whose specification parameter is set to even spacing.
"LUT object struct order for explicit value specification"	Change the order of the fields in the generated structure for a lookup table object whose specification parameter is set to explicit value.
"Generate destructor" on page 14-41	Specify whether to generate a destructor for the C++ model class.
"Use dynamic memory allocation for model block instantiation" on page 14-30	Specify whether generated code uses the operator `new`, during model object registration, to instantiate objects for referenced models configured with a C++ class interface.
"Code replacement library"	Create custom Code Replacement libraries using code replacement tool.
"Ignore custom storage classes" on page 14-22	Specify whether to apply or ignore custom storage classes.
"Ignore test point signals" on page 14-24	Specify allocation of memory buffers for test points.
"Implement each data store block as a unique access point" on page 14-35	Create unique variables for each read/write operation of a Data Store Memory block.

The following parameters under the **Advanced parameters** are infrequently used and have no other documentation.

Parameter	Description
`GenerateSharedConstants`	Control whether the code generator generates code with shared constants and shared functions. Default is `on`. When turned `off`, the code generator does not generate shared constants.
`InferredTypesCompatibility`	For compatibility with legacy code including `tmwtypes.h`, specify that the code generator creates a preprocessor directive `#define_TMWTYPES_ inside rtwtypes.h`

Parameter	Description
TargetLibSuffix *character vector* - ''	Control the suffix used for naming a target's dependent libraries (for example, _target.lib or _target.a). If specified, the character vector must include a period (.). (For generated model reference libraries, the library suffix defaults to _rtwlib.lib on Windows systems and _rtwlib.a on UNIX systems.). **Note** This parameter does not apply for model builds that use the toolchain approach, see "Library Control Parameters"
TargetPreCompLibLocation *character vector* - ''	Control the location of precompiled libraries. If you do not set this parameter, the code generator uses the location specified in rtwmakecfg.m.
IsERTTarget	Indicates whether or not the currently selected target is derived from the ERT target.
CPPClassGenCompliant	Indicates whether the target supports the ability to generate and configure C++ class interfaces to model code.
ConcurrentExecutionCompliant	Indicates whether the target supports concurrent execution
UseToolchainInfoCompliant	Indicate a custom target is toolchain-compliant.
ModelStepFunctionPrototypeControlCompliant	Indicates whether the target supports the ability to control the function prototypes of initialize and step functions that are generated for a Simulink model.
ParMdlRefBuildCompliant	Indicates if the model is configured for parallel builds when building a model that includes referenced models.
CompOptLevelCompliant off, on	Set in SelectCallback for a target to indicate whether the target supports the ability to use the **Compiler optimization level** parameter to control the compiler optimization level for building generated code. Default is off for custom targets and on for targets provided with the Simulink Coder and Embedded Coder products.
ModelReferenceCompliant character vector - off, **on**	Set in SelectCallback for a target to indicate whether the target supports model reference.
GenerateFullHeader	Generate full header including time stamp. For ERT targets, this parameter is on the **Code Generation > Templates** pane.

The following parameters are for MathWorks use only.

Parameter	Description
ExtModeTesting	For MathWorks use only.
ExtModeIntrfLevel	For MathWorks use only.
ExtModeMexFile	For MathWorks use only.

See Also

More About

- “Model Configuration”

Support: floating-point numbers

Description

Specify whether to generate floating-point data and operations.

Category: Code Generation > Interface

Settings

Default: On (GUI), `'off'` (command-line)

On

Generates floating-point data and operations.

Off

Generates pure integer code. If you clear this option, an error occurs if the code generator encounters floating-point data or expressions. The error message reports offending blocks and parameters.

Dependencies

- This option only appears for ERT-based targets.
- This option requires an Embedded Coder license when generating code.
- Selecting this parameter enables **Support: non-finite numbers** and clearing this parameter disables **Support: non-finite numbers**.
- This option must be the same for top-level and referenced models.
- When you select parameter **MAT-File Logging**, you must also select **Support: non-finite numbers** and **Support: floating-point numbers**.

Command-Line Information

Parameter: `PurelyIntegerCode`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Note The command-line values are reverse of the settings values. The value `'on'` in the command line corresponds to the description of “Off” in the settings section. The value `'off'` in the command line corresponds to the description of “On” in the settings section.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	Off (GUI), `'on'` (command-line) — for integer only
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 14-2

Support: complex numbers

Description

Specify whether to generate complex data and operations.

Category: Code Generation > Interface

Settings

Default: on

On

Generates complex numbers and related operations.

Off

Does not generate complex data and related operations. If you clear this option, an error occurs if the code generator encounters complex data or expressions. The error message reports offending blocks and parameters.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- If `off` for top model, must be `off` for referenced models.

Command-Line Information

Parameter: `SupportComplex`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Off (for real only)
Safety precaution	No impact

See Also

Related Examples

- "Model Configuration Parameters: Code Generation Interface" on page 14-2

Support: absolute time

Description

Specify whether to generate and maintain integer counters for absolute and elapsed time values.

Category: Code Generation > Interface

Settings

Default: on

On

Generates and maintains integer counters for blocks that require absolute or elapsed time values. Absolute time is the time from the start of program execution to the present time. An example of elapsed time is time elapsed between two trigger events.

If you select this option and the model does not include blocks that use time values, the target does not generate the counters.

Off

Does not generate integer counters to represent absolute or elapsed time values. If you do not select this option and the model includes blocks that require absolute or elapsed time values, an error occurs during code generation.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- Select this parameter if your model includes blocks that require absolute or elapsed time values.

Command-Line Information

Parameter: `SupportAbsoluteTime`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Off
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 14-2
- “Timers in Asynchronous Tasks”

Support: continuous time

Description

Specify whether to generate code for blocks that use continuous time.

Category: Code Generation > Interface

Settings

Default: off

☑ On

Generates code for blocks that use continuous time.

☐ Off

Does not generate code for blocks that use continuous time. If you do not select this option and the model includes blocks that use continuous time, an error occurs during code generation.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license to generate code.
- This parameter must be on for models that include blocks that require absolute or elapsed time values.
- This parameter is cleared if you select parameter **Remove error status field in real-time model data structure**.
- If both the following conditions exist, output values read from `ert_main` for a continuous output port can differ from the corresponding output values in logged data for a model:

  - You customize `ert_main.c` or `.cpp` to read model outputs after each base-rate model step.
  - You select parameters **Support: continuous time** and **Single output/update function**.

  The difference occurs because, while logged data captures output at major time steps, output read from `ert_main` after the base-rate model step can capture output at intervening minor time steps. The following table lists workarounds that eliminate the discrepancy.

Work Around	Customized ert_main.c	Customized ert_main.cpp
Separate the generated output and update functions (clear parameter **Single output/update function**), and insert code in `ert_main` to read model output values reflecting only the major time steps. For example, in `ert_main`, between the *`model`*`_output` call and the *`model`*`_update` call, read the model `External outputs` global data structure (defined in *`model`*`.h`).	X	

Work Around	Customized ert_main.c	Customized ert_main.cpp
Select parameter **Single output/update function**. Insert code in the generated *model*.c or .cpp file that returns model output values reflecting only major time steps. For example, in the model step function, between the output code and the update code, save the value of the model External outputs global data structure (defined in *model*.h). Then, restore the value after the update code completes.	X	X
Place a Zero-Order Hold block before the continuous output port.	X	X

Command-Line Information

Parameter: SupportContinuousTime
Type: character vector
Value: 'on' | 'off'
Default: 'off'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Off (execution, ROM), No impact (RAM)
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 14-2
- “Use Discrete and Continuous Time”

Support: variable-size signals

Description

Specify whether to generate code for models that use variable-size signals.

Category: Code Generation > Interface

Settings

Default: Off

On

Generates code for models that use variable-size signals.

Off

Does not generate code for models that use variable-size signals. If this parameter is off and the model uses variable-size signals, an error occurs during code generation.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: `SupportVariableSizeSignals`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- "Model Configuration Parameters: Code Generation Interface" on page 14-2

Pass root-level I/O as

Description

Control how root-level model input and output are passed to the reusable *model*_step function.

Category: Code Generation > Interface

Settings

Default: Individual arguments

Individual arguments

Passes each root-level model input and output value to *model*_step as a separate argument.

Structure reference

Packs root-level model input into a struct and passes struct to *model*_step as an argument. Similarly, packs root-level model output into a second struct and passes it to *model*_step.

Part of model data structure

Packages root-level model input and output into the real-time model data structure.

Dependencies

- This parameter only appears for ERT-based targets with parameter **Code interface packaging** set to Reusable function.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: RootIOFormat
Type: character vector
Value: 'Individual arguments' | 'Structure reference' | 'Part of model data structure'
Default: 'Individual arguments'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

model_step

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 14-2
- “Configure C Code Generation for Model Entry-Point Functions”
- “Generate Reentrant Code from Top Models”
- “Control Generation of Functions for Subsystems”
- “Generate Modular Function Code for Nonvirtual Subsystems”

Remove error status field in real-time model data structure

Description

Specify whether to log or monitor error status.

Category: Code Generation > Interface

Settings

Default: off

On

Omits the error status field from the generated real-time model data structure `rtModel`. This option reduces memory usage.

Be aware that selecting this option can cause the code generator to omit the `rtModel` data structure from generated code.

Off

Includes an error status field in the generated real-time model data structure `rtModel`. You can use available macros to monitor the field for error message data or set it with error message data.

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- Selecting this parameter clears parameter **Support: continuous time**.
- If your application contains multiple integrated models, the setting of this parameter must be the same for all of the models to avoid unexpected application behavior. For example, if you select the option for one model but not another, an error status might not get registered by the integrated application.

Command-Line Information

Parameter: `SuppressErrorStatus`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	Off
Traceability	No impact
Efficiency	On

Application	Setting
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 14-2
- “Real-Time Model Data Structure”

Include model types in model class

Description

Specify whether to generate model type definitions in a model class.

Category: Code Generation > Interface

Settings

Default: On

On

Includes model type definitions within the class namespace of the model. Model type definitions include:

- Root-level inports and outports
- Block inputs and outputs
- DWork vectors
- Block parameters and constant parameters
- Continuous states
- The real-time model data structure (`rtM`)

The generated code reduces the MISRA 7-3-1 violations.

A user-defined type such as `Simulink.Bus` object or a type defined in MATLAB Function blocks or Stateflow charts is still generated in the global namespace.

Off

Generate model type definitions within the global namespace.

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires Embedded Coder when generating code.

Command-Line Information

Parameter: `IncludeModelTypesInModelClass`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact

Application	Setting
Traceability	No impact
Efficiency	On
Safety precaution	No recommendation

See Also

Related Examples

- "Model Configuration Parameters: Code Generation Interface" on page 14-2

Ignore custom storage classes

Description

Specify whether to apply or ignore predefined storage classes.

Category: Code Generation > Interface

Settings

Default: off

On

If a storage class is individually specified (e.g., through the **Code Mappings editor** or a data object), it is treated as `Model default`. If a storage class is specified in `Data Defaults` in the Code Mappings editor, it is treated as "Default". `ExportedGlobal`, `ImportedExtern`, and `ImportedExternPointer` are three exceptions that are not affected by this option.

Off

Applies predefined storage classes as specified. You must clear this option if the model defines data objects that have predefined storage classes.

Tips

- Clear this parameter before configuring data objects that have predefined storage classes.
- The setting for top-level and referenced models must match.

Dependencies

- This parameter appears only for ERT-based targets.
- Clear this parameter to enable module packaging features.
- This parameter requires Embedded Coder when you generate code.

Command-Line Information

Parameter: `IgnoreCustomStorageClasses`
Type: character vector
Value: `'on'` | `'off`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact

Application	Setting
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface”
- “Organize Parameter Data into a Structure by Using Struct Storage Class”

Ignore test point signals

Description

Specify allocation of memory buffers for test points.

Category: Code Generation > Interface

Settings

Default: Off

On

Ignores test points during code generation, allowing optimal buffer allocation for signals with test points, facilitating transition from prototyping to deployment and avoiding accidental degradation of generated code due to workflow artifacts.

Off

Allocates separate memory buffers for test points, resulting in a loss of code generation optimizations such as reducing memory usage by storing signals in reusable buffers.

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: `IgnoreTestpoints`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	Off
Traceability	No impact
Efficiency	On
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface”
- “Appearance of Test Points in the Generated Code”

- “Configure Signals as Test Points”
- “How Generated Code Stores Internal Signal, State, and Parameter Data”

Support non-inlined S-functions

Description

Specify whether to generate code for non-inlined S-functions.

Category: Code Generation > Interface

Settings

Default: Off

☑ On

Generates code for non-inlined S-functions.

☐ Off

Does not generate code for non-inlined S-functions. If this parameter is off and the model includes a non-inlined S-function, an error occurs during the build process.

Tip

- Inlining S-functions is highly advantageous in production code generation, for example, for implementing device drivers. In such cases, clear this option to enforce use of inlined S-functions for code generation.
- Non-inlined S-functions require additional memory and computation resources, and can result in significant performance issues. Consider using an inlined S-function when efficiency is a concern.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- Selecting this parameter also selects **Support: floating-point numbers** and **Support: non-finite numbers**. If you clear **Support: floating-point numbers** or **Support: non-finite numbers**, a warning is displayed during code generation because these parameters are required by the S-function interface.

Command-Line Information

Parameter: `SupportNonInlinedSFcns`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact

Application	Setting
Traceability	No impact
Efficiency	Off
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface”
- “S-Functions and Code Generation”

Use dynamic memory allocation for model initialization

Description

Control how the generated code allocates memory for model data.

Category: Code Generation > Interface

Settings

Default: off

On

Generates a function to dynamically allocate memory (using `malloc`) for model data structures.

Off

Does not generate a dynamic memory allocation function. The generated code statically allocates memory for model data structures.

Dependencies

- This parameter only appears for ERT-based targets with **Code interface packaging** set to `Reusable function`.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: `GenerateAllocFcn`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No recommendation

See Also

`model_step`

Related Examples

- “Model Configuration Parameters: Code Generation Interface”
- “Configure C Code Generation for Model Entry-Point Functions”

- “Generate Reentrant Code from Top Models”
- “Control Generation of Functions for Subsystems”
- “Generate Modular Function Code for Nonvirtual Subsystems”

Use dynamic memory allocation for model block instantiation

Description

Specify whether generated code uses the operator `new`, during model object registration, to instantiate objects for referenced models configured with a C++ class interface.

Category: Code Generation > Interface

Settings

Default: off

On

Generates code that uses dynamic memory allocation to instantiate objects for referenced models configured with a C++ class interface. Specifically, during instantiation of an object for the top model in a model reference hierarchy, the generated code uses `new` to instantiate objects for referenced models.

Selecting this option frees a parent model from having to maintain information about referenced models beyond its direct children.

- If you select this option, be aware that a `bad_alloc` exception might be thrown, per the C++ standard, if an out-of-memory error occurs during the use of `new`. You must provide code to catch and process the `bad_alloc` exception in case an out-of-memory error occurs for a `new` call during construction of a top model object.
- If **Use dynamic memory allocation for model block instantiation** is selected and the base model contains a Model block, the build process might generate copy constructor and assignment operator functions in the private section of the model class. The purpose of the functions is to prevent pointer members within the model class from being copied by other code.

Off

Does not generate code that uses `new` to instantiate referenced model objects.

Clearing this option means that a parent model maintains information about its referenced models, including its direct and indirect children.

Dependencies

- This parameter appears only for ERT-based targets with **Language** set to `C++` and **Code interface packaging** set to `C++ class`.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: `UseOperatorNewForModelRefRegistration`
Type: character vector
Value: `'on'` | `'off'`

Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On
Safety precaution	No recommendation

See Also

Related Examples

- "Model Configuration Parameters: Code Generation Interface"

Terminate function required

Description

Specify whether to generate the *model*_terminate function.

Category: Code Generation > Interface

Settings

Default: on

On

Generates a *model*_terminate function. This function contains model termination code and should be called as part of system shutdown.

Off

Does not generate a *model*_terminate function. Suppresses the generation of this function if you designed your application to run indefinitely and does not require a terminate function.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: IncludeMdlTerminateFcn
Type: character vector
Value: 'on' | 'off'
Default: 'on'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No recommendation

See Also

model_terminate

Related Examples

- “Model Configuration Parameters: Code Generation Interface”

Combine signal/state structures

Description

Specify whether to combine global block signals and global state data into one data structure in the generated code

Category: Code Generation > Interface

Settings

Default: Off

On

Combine global block signal data (block I/O) and global state data (DWork vectors) into one data structure in the generated code.

Off

Store global block signals and global states in separate data structures, block I/O and DWork vectors, in the generated code.

Tips

The benefits to setting this parameter to `On` are:

- Enables tighter memory representation through fewer bitfields, which reduces RAM usage
- Enables better alignment of data structure elements, which reduces RAM usage
- Reduces the number of arguments to reusable subsystem and model reference block functions, which reduces stack usage
- Better readable data structures with more consistent element sorting

Example

For a model that generates the following code:

```
/* Block signals (auto storage) */
typedef struct {
  struct {
    uint_T LogicalOperator:1;
    uint_T UnitDelay1:1;
  } bitsForTID0;
} BlockIO;
/* Block states (auto storage) */
typedef struct {
  struct {
    uint_T UnitDelay_DSTATE:1
    uint_T UnitDelay1_DSTATE:1
  } bitsForTID0;
} D_Work;
```

If you select **Combine signal/state structures**, the generated code now looks like this:

```
/* Block signals and states (auto storage)
   for system */
typedef struct {
  struct {
    uint_T LogicalOperator:1;
    uint_T UnitDelay1:1;
    uint_T UnitDelay_DSTATE:1;
    uint_T UnitDelay1_DSTATE:1;
  } bitsForTID0;
} D_Work;
```

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: `CombineSignalStateStructs`
Type: character vector
Value: `'on'` | `'off'`
Default: `off`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface”
- “How Generated Code Stores Internal Signal, State, and Parameter Data”

Implement each data store block as a unique access point

Description

Specifies whether the generated code contains a single unique variable to hold the value for every Data Store Read and Write operation performed on a Data Store Memory block.

Category: Code Generation > Interface

Settings

Default: off

On

Creates a unique variable for each Data Store Memory block read/write operation. This unique variable enhances data coherency.

Off

Does not allocate a unique variable in the generated code for each Data Store Memory block read/write operation. This absence of an unique variable, diminishes data coherency.

Dependencies

This parameter requires Embedded Coder license.

Command-Line Information

Parameter: `DSAsUniqueAccess`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface”
- “Improve Data Coherency in Generated Code”

Generate separate internal data per entry-point function

Description

Generate a model block signals (block I/O) and discrete states (DWork) acting at the same rate into the same data structure. Depending on how many rates a model has, these structures contain the prefixes `FuncInternalData0`, `FunctionInternalData1`, and so on.

Category: Code Generation > Interface

Settings

Default: off

On

Store global block signal data (block I/O) and global state data (DWork vectors) operating at the same rate in one data structure in the generated code.

Off

Do not store global block signal data (block I/O) and global state data (DWork vectors) operating at the same rate in one data structure in the generated code

Tips

Setting this parameter to `On` improves cache performance when deploying a model to a multicore hardware environment that meets these requirements:

- The model has multiple rates and has the **Treat each discrete rate as a separate task** parameter set to `on`.
- The model contains multiple exported functions that run at different rates.

The previous models have separate entry-point functions that different cores can call. A core has its own data cache. Placing data for a single entry-point function in the same core data cache improves execution efficiency because the cache accesses are contiguous rather than spread out over multiple cores.

Example

For a model that generates this code:

```
/* Block signals and states (default storage) for system '<Root>' */
typedef struct {
  real_T RTBS2F;                      /* '<Root>/RTBS2F' */
  real_T UDS;                         /* '<Root>/UDS' */
  real_T Sum3;                        /* '<Root>/Sum3' */
  real_T Sum1;                        /* '<Root>/Sum1' */
  real_T UDF_DSTATE;                  /* '<Root>/UDF' */
  real_T UDS_DSTATE;                  /* '<Root>/UDS' */
  real_T RTBS2F_Buffer0;              /* '<Root>/RTBS2F' */
  real_T MIXEDDSM;                    /* '<Root>/DSMM' */
  real_T SLOWDSM;                     /* '<Root>/DSMS' */
} DW_demo1_T;
```

If you select **Generate separate internal data per entry-point function**, the generated code now looks like this code:

```
/* Block signals and states (default storage) for system '<Root>' */
typedef struct {
  real_T RTBS2F_Buffer0;              /* '<Root>/RTBS2F' */
  real_T MIXEDDSM;                    /* '<Root>/DSMM' */
} DW_demo1_T;

/* Internal Data Grouped For Same Function, for system '<Root>' */
typedef struct {
  real_T RTBS2F;                      /* '<Root>/RTBS2F' */
  real_T Sum3;                        /* '<Root>/Sum3' */
  real_T UDF_DSTATE;                  /* '<Root>/UDF' */
} FuncInternalData0_demo1_T;

/* Internal Data Grouped For Same Function, for system '<Root>' */
typedef struct {
  real_T UDS;                         /* '<Root>/UDS' */
  real_T Sum1;                        /* '<Root>/Sum1' */
  real_T UDS_DSTATE;                  /* '<Root>/UDS' */
  real_T SLOWDSM;                     /* '<Root>/DSMS' */
} FuncInternalData1_demo1_T;
```

Dependencies

- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled by selecting the **Combine signal/state structures** parameter.

Command-Line Information

Parameter: `GroupInternalDataByFunction`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface”
- “Multicore Processor Targets”

- “Time-Based Scheduling”

Multiword type definitions

Description

Specify whether to use system-defined or user-defined type definitions for multiword data types in generated code.

Category: Code Generation > Interface

Settings

Default: `System defined`

`System defined`

Use the default system type definitions for multiword data types in generated code. During code generation, if multiword usage is detected, multiword type definitions are generated into the file `multiword_types.h`.

`User defined`

Allows you to control how multiword type definitions are handled during the code generation process. Selecting this value enables the associated parameter **Maximum word length**, which allows you to specify a maximum word length, in bits, for which the code generation process generates multiword type definitions into the file `multiword_types.h`. The default maximum word length is 256. If you select 0, multiword type definitions are not generated into the file `multiword_types.h`.

The maximum word length for multiword types only determines the type definitions generated and does not impact the efficiency of the generated code. If the maximum word length for multiword types is set to 0 or too small, an error occurs when the generated code is compiled. This error is caused by the generated code using a type that does not have the required type definition. To resolve the error, increase the maximum word length and regenerate the code. If the maximum word length for multiword types is larger than required, then `multiword_types.h` might contain unused type definitions. Unused type definitions do not consume target resources.

Tips

- Adding a model to a model hierarchy or changing an existing model in the hierarchy can result in updates to the shared `multiword_types.h` file during code generation. These updates occur when the new model uses multiword types of length greater than those of the other models. You must then recompile and, depending on your development process, reverify previously generated code. To prevent updates to `multiword_types.h`, determine a maximum word length sufficiently big to cover the needs of all models in the hierarchy. Configure every model in the hierarchy to use that same maximum word length.
- The majority of embedded designs do not need multiword types. By setting maximum word length for multiword types to 0, you can prevent use of multiword variables on the target. If you use multiword variables with a maximum word length that is 0 or smaller than required, you are alerted with an error when the generated code is compiled.

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- Selecting the value `User defined` for this parameter enables the associated parameter **Maximum word length**.

Command-Line Information

Parameter: MultiwordTypeDef
Type: character vector
Value: `'System defined'` | `'User defined'`
Default: `'System defined'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface”

Generate destructor

Description

Specify whether to generate a destructor for the C++ model class.

Category: Code Generation > Interface

Settings

Default: on

On

Generates a destructor for the C++ model class.

Off

Does not generate a destructor for the C++ model class.

Dependencies

- This parameter appears only for ERT-based targets with **Language** set to `C++` and **Code interface packaging** set to `C++ class`.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: `GenerateDestructor`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface”
- “Interactively Configure C++ Interface”

MAT-file variable name modifier

Description

Select the text to add to MAT-file variable names.

Category: Code Generation > Interface

Settings

Default: `rt_`

`rt_`
    Adds prefix text.

`_rt`
    Adds suffix text.

`none`
    Does not add text.

Dependency

If you have an Embedded Coder license, for the GRT target or ERT-based targets, this parameter is enabled by **MAT-file logging**.

Command-Line Information

Parameter: `LogVarNameModifier`
Type: character vector
Value: `'none'` | `'rt_'` | `'_rt'`
Default: `'rt_'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- "Model Configuration Parameters: Code Generation Interface"
- "Log Program Execution Results"
- "Log Data for Analysis"

Existing shared code

Description

Specify folder that contains existing shared code

Category: Code Generation Advanced Parameters

Settings

Default: none

Path to folder that contains existing shared code. Specify the absolute path or a path relative to the Simulink preference **Code generation folder** (`CodeGenFolder`). The model build process uses the code in this folder instead of locally generated shared utility code.

Command-Line Information

Parameter: `ExistingSharedCode`
Type: character vector
Value: valid MATLAB variable name
Default: none

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Valid MATLAB variable name
Efficiency	No impact
Safety precaution	No impact

See Also

`sharedCodeUpdate`

Related Examples

- “Cross-Release Shared Utility Code Reuse”
- “Cross-Release Code Integration”

Code Generation Parameters: Identifiers

- “Model Configuration Parameters: Code Generation Identifiers” on page 15-2
- “Global variables” on page 15-4
- “Global types” on page 15-6
- “Field name of global types” on page 15-8
- “Subsystem methods” on page 15-10
- “Subsystem method arguments” on page 15-12
- “Local temporary variables” on page 15-14
- “Local block output variables” on page 15-16
- “Constant macros” on page 15-18
- “Shared utilities identifier format” on page 15-20
- “Minimum mangle length” on page 15-22
- “System-generated identifiers” on page 15-24
- “Generate scalar inlined parameters as” on page 15-29
- “Signal naming” on page 15-32
- “M-function” on page 15-34
- “Parameter naming” on page 15-36
- “M-function” on page 15-38
- “#define naming” on page 15-40
- “M-function” on page 15-42
- “Custom token text” on page 15-44
- “Shared checksum length” on page 15-45
- “EMX array utility functions identifier format” on page 15-46
- “EMX array types identifier format” on page 15-48

Model Configuration Parameters: Code Generation Identifiers

The **Code Generation > Identifiers** category includes parameters for configuring the comments in the generated code. These parameters require a Simulink Coder license. Additional parameters available with an ERT-based target require an Embedded Coder license.

On the Configuration Parameters dialog box, the following configuration parameters are on the **Code Generation > Identifiers** pane.

Parameter	Description
"Global variables" on page 15-4	Customize generated global variable identifiers.
"Global types" on page 15-6	Customize generated global type identifiers.
"Field name of global types" on page 15-8	Customize generated field names of global types.
"Subsystem methods" on page 15-10	Customize generated function names for reusable subsystems.
"Subsystem method arguments" on page 15-12	Customize generated function argument names for reusable subsystems.
"Local temporary variables" on page 15-14	Customize generated local temporary variable identifiers.
"Local block output variables" on page 15-16	Customize generated local block output variable identifiers.
"Constant macros" on page 15-18	Customize generated constant macro identifiers.
"Shared utilities identifier format" on page 15-20	Customize shared utility identifiers.
"Minimum mangle length" on page 15-22	Specify the minimum number of characters for generating name-mangling text to help avoid name collisions.
"Maximum identifier length"	Specify maximum number of characters in generated function, type definition, variable names.
"System-generated identifiers" on page 15-24	Specify whether the code generator uses shorter, more consistent names for the $N token in system-generated identifiers.
"Generate scalar inlined parameters as" on page 15-29	Control expression of scalar inlined parameter values in the generated code.
"Use the same reserved names as Simulation Target"	Specify whether to use the same reserved names as those specified in the **Simulation Target** pane.
"Reserved names"	Enter the names of variables or functions in the generated code that match the names of variables or functions specified in custom code.

The following configuration parameters are under the **Advanced parameters**.

Parameter	Description
“Shared checksum length” on page 15-45	Specify character length of `$C` token.
“EMX array utility functions identifier format” on page 15-46	Customize generated identifiers for `emxArray` (embeddable `mxArray`) utility functions.
“EMX array types identifier format” on page 15-48	Customize generated identifiers for `emxArray` (embeddable `mxArray`) types.
“Custom token text” on page 15-44	Specify text to insert for $U token.
“Duplicate enumeration member names”	Select the diagnostic action to take if the code generator detects two enumeration types with same member names. This parameter applies to only enumeration with imported data scope and the same storage type and value.
“Signal naming” on page 15-32	Specify rules for naming signals in generated code.
“M-function” on page 15-34	
“Parameter naming” on page 15-36	Specify rule for naming parameters in generated code.
“M-function” on page 15-38	
“#define naming” on page 15-40	Specify rule for naming `#define` parameters (defined with storage class `Define (Custom)`) in generated code.
“M-function” on page 15-42	

See Also

More About

- “Code Appearance”
- “Model Configuration”

Global variables

Description

Customize generated global variable identifiers.

Category: Code Generation > Identifiers

Settings

Default: `$R$N$M`

Enter a macro that specifies whether, and in what order, certain text is to be included in the generated identifier. The macro can include a combination of the following format tokens.

Token	Description
`$M`	Insert name-mangling text if required to avoid naming collisions. Required.
`$N`	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.
`$R`	Insert root model name into identifier, replacing unsupported characters with the underscore (_) character. Required for model referencing.
`$U`	Insert text that you specify for the `$U` token. Use the **Custom token text** parameter to specify this text.
`$G`	Insert the name of a storage class that is associated with the data item.

Tips

- Avoid name collisions in general. One way is to avoid using default block names (for example, `Gain1`, `Gain2`...) when your model has many blocks of the same type.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for name-mangling text.
- To control the case (upper or lower case) of the text that each token represents, include decorators such as `[U_]` in your macro. See “Control Case with Token Decorators”.
- If you specify `$R`, the value you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the `$R` and `$M` tokens.
- When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model, the code generator preserves the identifier from the referenced model. Name mangling is performed on the identifier in the higher-level model.
- This parameter setting only determines the name of objects, such as signals and parameters, if the object is set to `Auto`.
- For referenced models, if the **Global variables** parameter does not contain a `$R` token (which represents the name of the reference model), code generation prepends the `$R` token to the identifier format.

You can use the Model Advisor to identify models in a model referencing hierarchy for which code generation changes configuration parameter settings.

1. On the **Modeling** tab, select **Model Advisor**.
2. Select **By Task**.
3. Run the **Check code generation identifier formats used for model reference** check.

Dependency

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Command-Line Information

Parameter: `CustomSymbolStrGlobalVar`
Type: character vector
Value: valid combination of tokens
Default: `$R$N$M`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Identifiers” on page 15-2
- “Identifier Format Control”
- “Control Name Mangling in Generated Identifiers”
- “Avoid Identifier Name Collisions with Referenced Models”
- “Identifier Format Control Parameters Limitations”

Global types

Description

Customize generated global type identifiers.

Category: Code Generation > Identifiers

Settings

Default: `$N$R$M_T`

Enter a macro that specifies whether, and in what order, certain text is to be included in the generated identifier. The macro can include a combination of the following format tokens.

Token	Description
`$M`	Insert name-mangling text if required to avoid naming collisions. Required.
`$N`	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.
`$R`	Insert root model name into identifier, replacing unsupported characters with the underscore (_) character. Required for model referencing.
`$U`	Insert text that you specify for the `$U` token. Use the **Custom token text** parameter to specify this text.
`$G`	Insert the name of a storage class that is associated with the data item.

Tips

- Avoid name collisions in general. One way is to avoid using default block names (for example, `Gain1`, `Gain2`...) when your model has many blocks of the same type.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for name-mangling text.
- To control the case (upper or lower case) of the text that each token represents, include decorators such as `[U_]` in your macro. See “Control Case with Token Decorators”.
- If you specify `$R`, the value you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the `$R` and `$M` tokens.
- When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model, the code generator preserves the identifier from the referenced model. Name mangling is performed on the identifier in the higher-level model.
- Name mangling conventions do not apply to type names (that is, `typedef` statements) generated for global data types. The **Maximum identifier length** setting does not apply to type definitions. If you specify `$R`, the code generator includes the model name in the `typedef`.
- This option does not impact objects (such as signals and parameters) that have a storage class other than `Auto` (such as `ImportedExtern` or `ExportedGlobal`).

- For referenced models, if the **Global types** parameter does not contain a $R token (which represents the name of the reference model), code generation prepends the $R token to the identifier format.

  You can use the Model Advisor to identify models in a model referencing hierarchy for which code generation changes configuration parameter settings.

  1. On the **Modeling** tab, select **Model Advisor**.
  2. Select **By Task**.
  3. Run the **Check code generation identifier formats used for model reference** check.

Dependency

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Command-Line Information

Parameter: CustomSymbolStrType
Type: character vector
Value: valid combination of tokens
Default: NR$M_T

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Identifiers” on page 15-2
- “Identifier Format Control”
- “Control Name Mangling in Generated Identifiers”
- “Avoid Identifier Name Collisions with Referenced Models”
- “Identifier Format Control Parameters Limitations”

Field name of global types

Description

Customize generated field names of global types.

Category: Code Generation > Identifiers

Settings

Default: NM

Enter a macro that specifies whether, and in what order, certain text is to be included in the generated identifier. The macro can include a combination of the following format tokens.

Token	Description
$A	Insert data type acronym into signal and work vector identifiers. For example, i32 for int32_t.
$H	Insert tag indicating system hierarchy level. For root-level blocks, the tag is the text root_. For blocks at the subsystem level, the tag is of the form sN_, where N is a unique system number assigned by the Simulink software.
$M	Insert name-mangling text if required to avoid naming collisions. Required.
$N	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.
$U	Insert text that you specify for the $U token. Use the **Custom token text** parameter to specify this text.

Tips

- Avoid name collisions in general. One way is to avoid using default block names (for example, Gain1, Gain2...) when your model has many blocks of the same type.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for name-mangling text.
- To control the case (upper or lower case) of the text that each token represents, include decorators such as [U_] in your macro. See “Control Case with Token Decorators”.
- The **Maximum identifier length** setting does not apply to type definitions.
- This option does not impact objects (such as signals and parameters) that have a storage class other than Auto (such as ImportedExtern or ExportedGlobal).

Dependency

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Command-Line Information

Parameter: `CustomSymbolStrField`
Type: character vector
Value: valid combination of tokens
Default: `$N$M`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Identifiers” on page 15-2
- “Identifier Format Control”
- “Control Name Mangling in Generated Identifiers”
- “Identifier Format Control Parameters Limitations”

Subsystem methods

Description

Customize generated function names for reusable subsystems.

Category: Code Generation > Identifiers

Settings

Default: `$R$N$M$F`

Enter a macro that specifies whether, and in what order, certain text is to be included in the generated identifier. The macro can include a combination of the following format tokens.

Token	Description
`$F`	Insert method name (for example, `_Update` for update method).
`$H`	Insert tag indicating system hierarchy level. For root-level blocks, the tag is the text `root_`. For blocks at the subsystem level, the tag is of the form `sN_`, where `N` is a unique system number assigned by the Simulink software. Empty for Stateflow functions.
`$M`	Insert name-mangling text if required to avoid naming collisions. Required.
`$N`	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.
`$R`	Insert root model name into identifier, replacing unsupported characters with the underscore (_) character. Required for model referencing.
`$U`	Insert text that you specify for the `$U` token. Use the **Custom token text** parameter to specify this text.

Tips

- Avoid name collisions in general. One way is to avoid using default block names (for example, `Gain1`, `Gain2`...) when your model has many blocks of the same type.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for name-mangling text.
- To control the case (upper or lower case) of the text that each token represents, include decorators such as `[U_]` in your macro. See "Control Case with Token Decorators".
- If you specify `$R`, the value you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the `$R` and `$M` tokens.
- When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model, the code generator preserves the identifier from the referenced model. Name mangling is performed on the identifier in the higher-level model.

- Name mangling conventions do not apply to type names (that is, `typedef` statements) generated for global data types. The **Maximum identifier length** setting does not apply to type definitions. If you specify `$R`, the code generator includes the model name in the `typedef`.
- This option does not impact objects (such as signals and parameters) that have a storage class other than `Auto` (such as `ImportedExtern` or `ExportedGlobal`).
- For referenced models, if the **Subsystem methods** parameter does not contain a `$R` token (which represents the name of the reference model), code generation prepends the `$R` token to the identifier format.

  You can use the Model Advisor to identify models in a model referencing hierarchy for which code generation changes configuration parameter settings.

  1. On the **Modeling** tab, select **Model Advisor**.
  2. Select **By Task**.
  3. Run the **Check code generation identifier formats used for model reference** check.

Dependency

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Command-Line Information

Parameter: `CustomSymbolStrFcn`
Type: character vector
Value: valid combination of tokens
Default: `$R$N$M$F`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Identifiers” on page 15-2
- “Identifier Format Control”
- “Control Name Mangling in Generated Identifiers”
- “Avoid Identifier Name Collisions with Referenced Models”
- “Identifier Format Control Parameters Limitations”

Subsystem method arguments

Description

Customize generated function argument names for reusable subsystems.

Category: Code Generation > Identifiers

Settings

Enter a macro that specifies whether, and in what order, certain text is to be included in the generated argument name. The macro can include a combination of the following format tokens.

Token	Description
`$I`	• Insert `u` if the argument is an input. • Insert `y` if the argument is an output. • Insert `uy` if the argument is an input and output. Optional.
`$M`	Insert name-mangling text if required to avoid naming collisions. Required.
`$N`	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated. Recommended to maximize readability of generated code.
`$U`	Insert text that you specify for the `$U` token. Use the **Custom token text** parameter to specify this text.

Tips

- Avoid name collisions in general. One way is to avoid using default block names (for example, `Gain1`, `Gain2`...) when your model has many blocks of the same type.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for name-mangling text.
- To control the case (upper or lower case) of the text that each token represents, include decorators such as `[U_]` in your macro. See “Control Case with Token Decorators”.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Command-Line Information

Parameter: `CustomSymbolStrFcnArg`
Type: character vector
Value: valid combination of tokens
Default: `rt$I$N$M`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Identifiers” on page 15-2
- “Identifier Format Control”
- “Control Name Mangling in Generated Identifiers”
- “Identifier Format Control Parameters Limitations”

Local temporary variables

Description

Customize generated local temporary variable identifiers.

Category: Code Generation > Identifiers

Settings

Default: `$N$M`

Enter a macro that specifies whether, and in what order, certain text is to be included in the generated identifier. The macro can include a combination of the following format tokens.

Token	Description
`$A`	Insert data type acronym (for example, `i32` for integers) into signal and work vector identifiers.
`$M`	Insert name-mangling text if required to avoid naming collisions. Required.
`$N`	Insert name of object (block, signal or signal object, state, parameter, or parameter object) for which identifier is generated.
`$R`	Insert root model name into identifier, replacing unsupported characters with the underscore (_) character. Required for model referencing.
`$U`	Insert text that you specify for the `$U` token. Use the **Custom token text** parameter to specify this text.

Tips

- Avoid name collisions. One way is to avoid using default block names (for example, `Gain1`, `Gain2`...) when your model has many blocks of the same type.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers that you expect to generate. Reserve at least three characters for name-mangling text.
- To control the case (upper or lower case) of the text that each token represents, include decorators such as `[U_]` in your macro. See "Control Case with Token Decorators".
- If you specify `$R`, the value you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the `$R` and `$M` tokens.
- When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model, the code generator preserves the identifier from the referenced model. Name mangling is performed on the identifier in the higher-level model.
- This option does not impact objects (such as signals and parameters) that have a storage class other than `Auto` (such as `ImportedExtern` or `ExportedGlobal`).

Dependency

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Command-Line Information

Parameter: `CustomSymbolStrTmpVar`
Type: character vector
Value: valid combination of tokens
Default: `$N$M`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Identifiers” on page 15-2
- “Identifier Format Control”
- “Control Name Mangling in Generated Identifiers”
- “Avoid Identifier Name Collisions with Referenced Models”
- “Identifier Format Control Parameters Limitations”

Local block output variables

Description

Customize generated local block output variable identifiers.

Category: Code Generation > Identifiers

Settings

Default: `rtb_$N$M`

Enter a macro that specifies whether, and in what order, certain text is to be included in the generated identifier. The macro can include a combination of the following format tokens.

Token	Description
`$A`	Insert data type acronym (for example, `i32` for integers) into signal and work vector identifiers.
`$M`	Insert name-mangling text if required to avoid naming collisions. Required.
`$N`	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.
`$U`	Insert text that you specify for the `$U` token. Use the **Custom token text** parameter to specify this text.

Tips

- Avoid name collisions in general. One way is to avoid using default block names (for example, `Gain1`, `Gain2`...) when your model has many blocks of the same type.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for name-mangling text.
- To control the case (upper or lower case) of the text that each token represents, include decorators such as `[U_]` in your macro. See "Control Case with Token Decorators".
- This option does not impact objects (such as signals and parameters) that have a storage class other than `Auto` (such as `ImportedExtern` or `ExportedGlobal`).

Dependency

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Command-Line Information

Parameter: `CustomSymbolStrBlkIO`
Type: character vector

Value: valid combination of tokens
Default: `rtb_$N$M`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- "Model Configuration Parameters: Code Generation Identifiers" on page 15-2
- "Identifier Format Control"
- "Control Name Mangling in Generated Identifiers"
- "Identifier Format Control Parameters Limitations"

Constant macros

Description

Customize generated constant macro identifiers.

Category: Code Generation > Identifiers

Settings

Default: `$R$N$M`

Enter a macro that specifies whether, and in what order, certain text is to be included in the generated identifier. The macro can include a combination of the following format tokens.

Token	Description
`$M`	Insert name-mangling text if required to avoid naming collisions. Required.
`$N`	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.
`$R`	Insert root model name into identifier, replacing unsupported characters with the underscore (_) character. Required for model referencing.
`$U`	Insert text that you specify for the `$U` token. Use the **Custom token text** parameter to specify this text.

Tips

- Avoid name collisions in general. One way is to avoid using default block names (for example, `Gain1`, `Gain2`...) when your model has many blocks of the same type.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for name-mangling text.
- To control the case (upper or lower case) of the text that each token represents, include decorators such as `[U_]` in your macro. See “Control Case with Token Decorators”.
- If you specify `$R`, the value you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the `$R` and `$M` tokens.
- When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model, the code generator preserves the identifier from the referenced model. Name mangling is performed on the identifier in the higher-level model.
- This option does not impact objects (such as signals and parameters) that have a storage class other than `Auto` (such as `ImportedExtern` or `ExportedGlobal`).
- For referenced models, if the **Constant macros** parameter does not contain a `$R` token (which represents the name of the reference model), code generation prepends the `$R` token to the identifier format.

You can use the Model Advisor to identify models in a model referencing hierarchy for which code generation changes configuration parameter settings.

1. On the **Modeling** tab, select **Model Advisor**.
2. Select **By Task**.
3. Run the **Check code generation identifier formats used for model reference** check.

Dependency

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Command-Line Information

Parameter: `CustomSymbolStrMacro`
Type: character vector
Value: valid combination of tokens
Default: `$R$N$M`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Identifiers” on page 15-2
- “Identifier Format Control”
- “Control Name Mangling in Generated Identifiers”
- “Avoid Identifier Name Collisions with Referenced Models”
- “Identifier Format Control Parameters Limitations”

Shared utilities identifier format

Description

Customize shared utility identifiers.

Category: Code Generation > Identifiers

Note Starting in R2018a, for new models, do not use this configuration parameter. Instead, use the Embedded Coder Dictionary to create a function customization template that specifies the naming rule, then apply the template by using the Code Mapping editor. See “Migration of Memory Section and Shared Utility Settings from Configuration Parameters to Code Mappings” and “Configure Naming of Generated Functions”.

Settings

Default: `$N$C`

Customize generated shared utility identifier names.

Enter a macro that specifies whether, and in what order, certain text is to be included in the generated identifier. The macro can include a combination of the following format tokens.

Token	Description
`$N`	Insert system-generated name of the object for which the shared utility identifier is generated. Optional.
`$C`	Insert eight-character conditional checksum when `$N` is not specified or the **Maximum identifier length** does not accommodate the full length of `$N`. Modify checksum character length by using **Shared checksum length** parameter. Required.
`$R`	Insert root model name into identifier, replacing unsupported characters with the underscore (_) character.
`$U`	Insert text that you specify for the `$U` token. Use the **Custom token text** parameter to specify this text.

Tips

- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers that you expect to generate.
- The checksum token `$C` is required. If `$C` is specified without `$N` or `$R`, the checksum is included in the identifier name. Otherwise, the code generator includes the checksum when necessary to prevent name collisions.
- To control the case (upper or lower case) of the text that each token represents, include decorators such as `[U_]` in your macro. See “Control Case with Token Decorators”.
- If you specify `$N` or `$R`, then the checksum is included in the identifier name only when the identifier length is too short to accommodate the fully expanded format text. The code generator includes the checksum and truncates `$N` or `$R` until the length is equal to **Maximum identifier length**. When necessary, an underscore is inserted to separate tokens.

- If you specify `$N` and `$R`, then the checksum is included in the identifier name only when the identifier length is too short to accommodate the fully expanded format text. The code generator includes the checksum and truncates `$N`, and if required, `$R`, until the length is equal to **Maximum identifier length**. When necessary, an underscore is inserted to separate tokens.
- Descriptive text helps make the identifier name more accessible.
- For versions prior to R2016a, the **Shared utilities identifier format** parameter does not support the `$R` token. For a model, if the **Shared utilities identifier format** parameter includes a `$R` token, and you export the model to a version prior to R2016a, the **Shared utilities identifier format** parameter defaults to `$N$C`.

Dependency

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Command-Line Information

Parameter: `CustomSymbolStrUtil`
Type: character vector
Value: valid combination of tokens
Default: `$N$C`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Identifiers” on page 15-2
- “Identifier Format Control”
- “Exceptions to Identifier Formatting Conventions”
- “Generate Shared Utility Code”

Minimum mangle length

Description

Increase the minimum number of characters for generating name-mangling text to help avoid name collisions.

Category: Code Generation > Identifiers

Settings

Default: 1

Specify an integer value that indicates the minimum number of characters the code generator uses when generating name-mangling text. The maximum possible value is 15. The minimum value automatically increases during code generation as a function of the number of collisions. A larger value reduces the chance of identifier disturbance when you modify the model.

Tips

- Minimize disturbance to the generated code during development by specifying a value of 4. This value is conservative. It allows for over 1.5 million collisions for a particular identifier before the mangle length increases.
- Set the value to reserve at least three characters for the name-mangling text. The length of the name-mangling text increases as the number of name collisions increases.

Dependency

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Command-Line Information

Parameter: `MangleLength`
Type: integer
Value: value between 1 and 15
Default: 1

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	1
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Identifiers” on page 15-2
- “Control Name Mangling in Generated Identifiers”
- “Maintain Traceability for Generated Identifiers”

System-generated identifiers

Description

Specify whether the code generator uses shorter, more consistent names for the `$N` token in system-generated identifiers.

Category: Code Generation > Identifiers

Settings

Default: `Shortened`

`Classic`

Generate longer identifier names for the `$N` token. For example, for a model named `sym`, if:

- “Global variables” on page 15-4 is `$N$R$M`, the block state identifier is `DWork_sym`.
- “Global types” on page 15-6 is `$R$N$M`, the block state type is a structure named `sym_D_Work`.

`Shortened`

Shorten identifier names for the `$N` token to allow more space for user names. This option provides a more predictable and consistent naming system that uses camel case, no underscores or plurals, and consistent abbreviations for both a type and a variable. For example, for a model named `sym`, if:

- “Global variables” on page 15-4 is `$N$R$M`, the block state identifier is `DW_sym`.
- “Global types” on page 15-6 is `$R$N$M`, the block state type is a structure named `sym_DW`.

System-generated identifiers per model

Classic	Shortened	Data Representation	Description
BlockIO, B	B	Type, Global Variable	Block signals of the system
ExternalInputs	ExtU	Type	Block input data for root system
ExternalInputSizes	ExtUSize	Type	Size of block input data for the root system (used when inputs are variable dimensions)
ExternalOutputs	ExtY	Type	Block output data for the root system
ExternalOutputSizes	ExtYSize	Type	Size of block output data for the root system
U	U	Global Variable	Input data
USize	USize	Global Variable	Size of input data
Y	Y	Global Variable	Output data
YSize	YSize	Global Variable	Size of output data
Parameters	P	Type, Global Variable	Parameters for the system
ConstBlockIO	ConstB	Const Type, Global Variable	Block inputs and outputs that are constants
MachineLocalData, Machine	MachLocal	Const Type, Global Variable	Used by ERT S-function targets
ConstParam, ConstP	ConstP	Const Type, Global Variable	Constant parameters in the system
ConstParamWithInit, ConstWithInitP	ConstInitP	Const Type, Global Variable	Initialization data for constant parameters in the system
D_Work, DWork	DW	Type, Global Variable	Block states in the system
MassMatrixGlobal	MassMatrix	Type, Global Variable	Used for physical modeling blocks
PrevZCSigStates, PrevZCSigState	PrevZCX	Type, Global Variable	Previous zero-crossing signal state
ContinuousStates, X	X	Type, Global Variable	Continuous states
StateDisabled, Xdis	XDis	Type, Global Variable	Status of an enabled subsystem
StateDerivatives, Xdot	XDot	Type, Global Variable	Derivatives of continuous states at each time step
ZCSignalValues, ZCSignalValue	ZCV	Type, Global Variable	Zero-crossing signals
DefaultParameters	DefaultP	Global Variable	Default parameters in the system

Classic	Shortened	Data Representation	Description
GlobalTID	GlobalTID	Global Variable	Used for sample time for states in referenced models
InvariantSignals	Invariant	Global Variable	Invariant signals
NSTAGES	NSTAGES	Global Variable	Solver macro
Object	Obj	Global Variable	Used by ERT C++ code generation to refer to referenced model objects
TimingBridge	TimingBrdg	Global Variable	Timing information stored in different data structures
SharedDSM	SharedDSM	Type, Global Variable	Shared local data stores, which are Data Store Memory blocks with **Share across model instances** selected
InstP	InstP	Type, Global Variable	Parameter arguments for the system

System-generated identifier names per referenced model or reusable subsystem

Classic	Shortened	Data Representation	Description
`rtB`, `B`	`B`	Type, Global Variable	Block signals of the system
`rtC`, `C`	`ConstB`	Type, Global Variable	Block inputs and outputs that are constants
`rtDW`, `DW`	`DW`	Type, Global Variable	Block states in the system
`rtMdlrefDWork, MdlrefDWork`	`MdlRefDW`	Type, Global Variable	Block states in referenced model
`rtP, P`	`P`	Type, Global Variable	Parameters for the system
`rtRTM, RTM`	`RTM`	Type, Global Variable	RT_Model structure
`rtX, X`	`X`	Type, Global Variable	Continuous states in model reference
`rtXdis, Xdis`	`XDis`	Type, Global Variable	Status of an enabled subsystem
`rtXdot, Xdot`	`XDot`	Type, Global Variable	Derivatives of the S-function's continuous states at each time step
`rtZCE, ZCE`	`ZCE`	Type, Global Variable	Zero-crossing enabled
`rtZCSV, ZCSV`	`ZCV`	Type, Global Variable	Zero-crossing signal values
`rtSharedDSM, SharedDSM`	`SharedDSM`	Type, Global Variable	Shared local data stores, which are Data Store Memory blocks with **Share across model instances** selected
`rtInstP, InstP`	`InstP`	Type, Global Variable	Parameter arguments for the system

Note When you set the **System-generated identifiers** parameter to `Shortened`, while generating an identifier for a global variable of a referenced model which is a run-time parameter, the code generator adds a prefix `rtP_` to the variable name for the `$N` token.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Command-Line Information

Parameter: `InternalIdentifier`
Type: character vector
Value: `Classic` | `Shortened`
Default: `Shortened`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Identifiers” on page 15-2
- “Construction of Generated Identifiers”
- “Identifier Name Collisions and Mangling”
- “Specify Identifier Length to Avoid Naming Collisions”
- “Specify Reserved Names for Generated Identifiers”
- “Data Structures in the Generated Code”
- “Customize Generated Identifier Naming Rules”
- “Identifier Format Control”

Generate scalar inlined parameters as

Description

Control expression of scalar inlined parameter values in the generated code. Block parameters appear inlined in the generated code when you set **Configuration Parameters** > **Optimization** > **Default parameter behavior** to `Inlined`.

Category: Code Generation > Identifiers

Settings

Default: `Literals`

`Literals`

Generates scalar inlined parameters as numeric constants.

`Macros`

Generates scalar inlined parameters as variables with `#define` macros. This setting makes generated code more readable.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Command-Line Information

Parameter: `InlinedPrmAccess`
Type: character vector
Value: `Literals` | `Macros`
Default: `Literals`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	`Macros`
Efficiency	No impact
Safety precaution	No impact

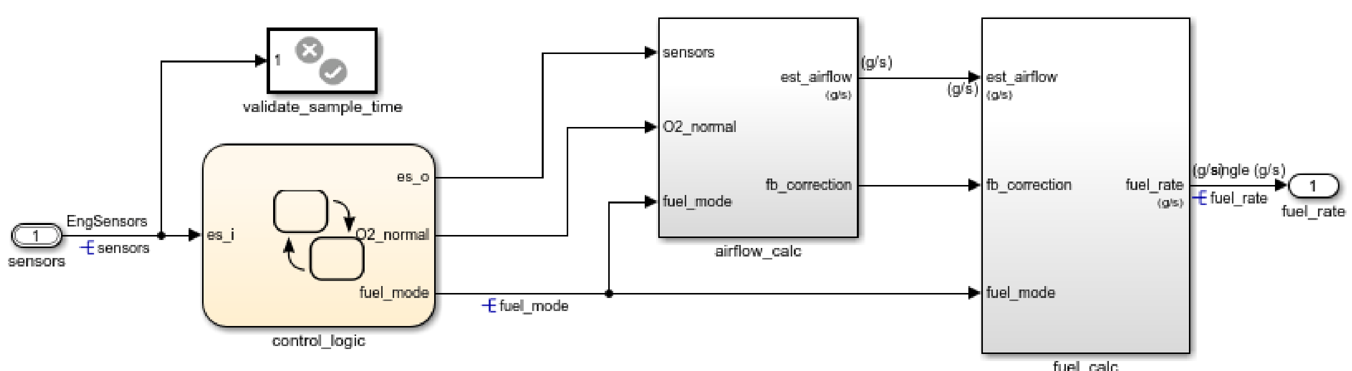
Improve Code Readability by Generating Block Parameter Values as Macros

When you generate efficient code by inlining the numeric values of block parameters (with the configuration parameter **Default parameter behavior**), you can configure scalar parameters to

appear as macros instead of literal numbers. Each macro has a unique name that is based on the name of the corresponding block parameter.

Open the example model `sldemo_fuelsys_dd_controller`.

```
sldemo_fuelsys_dd_controller
```



The model uses these configuration parameter settings:

- **Default parameter behavior** set to `Inlined`.
- **System target file** set to `ert.tlc`.

Set the configuration parameter **Generate scalar inlined parameters as** to `Macros`.

```
set_param('sldemo_fuelsys_dd_controller','InlinedPrmAccess','Macros')
```

Generate code from the model.

```
slbuild('sldemo_fuelsys_dd_controller')
```

```
### Starting build procedure for: sldemo_fuelsys_dd_controller
### Successful completion of code generation for: sldemo_fuelsys_dd_controller

Build Summary

Top model targets built:

Model                         Action          Rebuild Reason
===========================================================================================
sldemo_fuelsys_dd_controller  Code generated  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 22.119s
```

The header file `sldemo_fuelsys_dd_controller_private.h` defines several macros that represent inlined (nontunable) block parameters. For example, the macros

rtCP_DiscreteFilter_NumCoe_EL_0 and rtCP_DiscreteFilter_NumCoe_EL_1 represent floating-point constants.

```
file = fullfile('sldemo_fuelsys_dd_controller_ert_rtw',...
    'sldemo_fuelsys_dd_controller_private.h');
rtwdemodbtype(file,'#define rtCP_DiscreteFilter_NumCoe_EL_0',...
    'rtCP_DiscreteFilter_NumCoe_EL_1',1,1)
```

```
#define rtCP_DiscreteFilter_NumCoe_EL_0 (8.7696F)
#define rtCP_DiscreteFilter_NumCoe_EL_1 (-8.5104F)
```

The comments above the macro definitions indicate that the code generated for a Discrete Filter block uses the macros.

```
rtwdemodbtype(file,'Computed Parameter: DiscreteFilter_NumCoef',...
    'Referenced by: ''<S12>/Discrete Filter''',1,1)
```

```
/* Computed Parameter: DiscreteFilter_NumCoef
 * Referenced by: '<S12>/Discrete Filter'
```

Click the hyperlink to navigate to the block in the model.

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Identifiers” on page 15-2

Signal naming

Description

Specify rules for naming `mpt` signals in generated code.

Category: Code Generation > Identifiers

Settings

Default: `None`

`None`

Does not change signal names when creating corresponding identifiers in generated code. Signal identifiers in the generated code match the signal names that appear in the model.

`Force upper case`

Uses uppercase characters when creating identifiers for signal names in the generated code.

`Force lower case`

Uses lowercase characters when creating identifiers for signal names in the generated code.

`Custom M-function`

Uses the MATLAB function specified with the **M-function** parameter to create identifiers for signal names in the generated code.

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- Setting this parameter to `Custom M-function` enables **M-function**.
- This parameter must be the same for top-level and referenced models.
- If you give a value to the **Identifier** parameter of a `mpt.Signal` data object, that value overrides the specification of the **Signal naming** parameter.

Limitation

This parameter works only for `mpt.Signal` data objects.

Command-Line Information

Parameter: `SignalNamingRule`
Type: character vector
Value: `None` | `UpperCase` | `LowerCase` | `Custom`
Default: `None`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	`Force upper case`
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Identifiers” on page 15-2
- “Apply Naming Rules to Simulink Data Objects”
- “Programming”

M-function

Description

Specify rule for naming identifiers in generated code.

Category: Code Generation > Identifiers

Settings

Default: `''`

Enter the name of a MATLAB language file that contains the naming rule to be applied to signal, parameter, or `#define` parameter identifiers in generated code. Examples of rules you might program in such a MATLAB function include:

- Remove underscore characters from signal names.
- Add an underscore before uppercase characters in parameter names.
- Make identifiers uppercase in generated code.

For example, the following function returns an identifier name by appending the text `_signal` to a signal data object name.

```
function revisedName = append_text(name, object)
% APPEND_TEXT: Returns an identifier for generated
% code by appending text to a data object name.
%
% Input arguments:
% name: data object name as spelled in model
% object: target data object
%
% Output arguments:
% revisedName: altered identifier returned for use in
% generated code.
%
%
text = '_signal';

revisedName = [name,text];
```

Tip

The MATLAB language file must be in the MATLAB path.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.
- Is enabled by **Signal naming**.

- Must be the same for top-level and referenced models.

Command-Line Information

Parameter: `SignalNamingFcn`
Type: character vector
Value: MATLAB language file
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Identifiers” on page 15-2
- “Specify Naming Rule Using a Function”
- “Programming”

Parameter naming

Description

Specify rule for naming `mpt` parameters in generated code.

Category: Code Generation > Identifiers

Settings

Default: `None`

`None`

Does not change parameter names when creating corresponding identifiers in generated code. Parameter identifiers in the generated code match the parameter names that appear in the model.

`Force upper case`

Uses uppercase characters when creating identifiers for parameter names in the generated code.

`Force lower case`

Uses lowercase characters when creating identifiers for parameter names in the generated code.

`Custom M-function`

Uses the MATLAB function specified with the **M-function** parameter to create identifiers for parameter names in the generated code.

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- Setting this parameter to `Custom M-function` enables **M-function**.
- This parameter must be the same for top-level and referenced models.

Limitation

This parameter works only for `mpt.Parameter` data objects.

Command-Line Information

Parameter: `ParamNamingRule`
Type: character vector
Value: `None` | `UpperCase` | `LowerCase` | `Custom`
Default: `None`

Recommended Settings

Application	Setting
Debugging	No impact

Application	Setting
Traceability	`Force upper case`
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Identifiers” on page 15-2
- “Apply Naming Rules to Simulink Data Objects”
- “Programming”

M-function

Description

Specify rule for naming identifiers in generated code.

Category: Code Generation > Identifiers

Settings

Default: `''`

Enter the name of a MATLAB language file that contains the naming rule to be applied to signal, parameter, or `#define` parameter identifiers in generated code. Examples of rules you might program in such a MATLAB function include:

- Remove underscore characters from signal names.
- Add an underscore before uppercase characters in parameter names.
- Make identifiers uppercase in generated code.

For example, the following function returns an identifier name by appending the text `_param` to a parameter data object name.

```
function revisedName = append_text(name, object)
% APPEND_TEXT: Returns an identifier for generated
% code by appending text to a data object name.
%
% Input arguments:
% name: data object name as spelled in model
% object: target data object
%
% Output arguments:
% revisedName: altered identifier returned for use in
% generated code.
%
%
text = '_param';

revisedName = [name,text];
```

Tip

The MATLAB language file must be in the MATLAB path.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.
- Is enabled by **Parameter naming**.

- Must be the same for top-level and referenced models.

Command-Line Information

Parameter: `ParamNamingFcn`
Type: character vector
Value: MATLAB language file
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Identifiers” on page 15-2
- “Specify Naming Rule Using a Function”
- “Programming”

#define naming

Description

Specify rule for naming `#define` parameters (defined with storage class `Define (Custom)`) in generated code.

Category: Code Generation > Identifiers

Settings

Default: `None`

`None`

Does not change `#define` parameter names when creating corresponding identifiers in generated code. Parameter identifiers in the generated code match the parameter names that appear in the model.

`Force upper case`

Uses uppercase characters when creating identifiers for `#define` parameter names in the generated code.

`Force lower case`

Uses lowercase characters when creating identifiers for `#define` parameter names in the generated code.

`Custom M-function`

Uses the MATLAB function specified with the **M-function** parameter to create identifiers for `#define` parameter names in the generated code.

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- Setting this parameter to `Custom M-function` enables **M-function**.
- This parameter must be the same for top-level and referenced models.

Command-Line Information

Parameter: `DefineNamingRule`
Type: character vector
Value: `None` | `UpperCase` | `LowerCase` | `Custom`
Default: `None`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	`Force upper case`

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Identifiers” on page 15-2
- “Specify Naming Rule for Storage Class Define”
- “Programming”

M-function

Description

Specify rule for naming identifiers in generated code.

Category: Code Generation > Identifiers

Settings

Default: `''`

Enter the name of a MATLAB language file that contains the naming rule to be applied to signal, parameter, or `#define` parameter identifiers in generated code. Examples of rules you might program in such a MATLAB function include:

- Remove underscore characters from signal names.
- Add an underscore before uppercase characters in parameter names.
- Make identifiers uppercase in generated code.

For example, the following function returns an identifier name by appending the text `_define` to a data object name.

```
function revisedName = append_text(name, object)
% APPEND_TEXT: Returns an identifier for generated
% code by appending text to a #define data object name.
%
% Input arguments:
% name: data object name as spelled in model
% object: target data object
%
% Output arguments:
% revisedName: altered identifier returned for use in
% generated code.
%
%
text = '_define';

revisedName = [name,text];
```

Tip

The MATLAB language file must be in the MATLAB path.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.
- Is enabled by **#define naming**.

- Must be the same for top-level and referenced models.

Command-Line Information

Parameter: `DefineNamingFcn`
Type: character vector
Value: MATLAB language file
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Identifiers” on page 15-2
- “Specify Naming Rule Using a Function”
- “Programming”

Custom token text

Description

Specify text to insert for $U token.

Category: Code Generation > Identifiers

Settings

Default: `''`

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code

Command-Line Information

Parameter: `CustomUserTokenString`
Type: `character vector`
Value: `''` or `user-specified name`
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Set a custom string and use $U in symbols
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Identifier Format Control”
- “Model Configuration Parameters: Code Generation Identifiers”

Shared checksum length

Description

Specify character length of `$C` token.

Category: Code Generation > Identifiers

Settings

Default: 8 **Minimum:** 1 **Maximum:** 15

Specify an integer value that indicates the number of characters to expand the `$C` token for the **Shared utilities identifier format** parameter.

Tip

To avoid the possibility of a naming collision, consider increasing this parameter value.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Command-Line Information

Parameter: `SharedChecksumLength`
Type: integer
Value: valid value
Default: 8

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Identifiers”
- “Shared utilities identifier format” on page 15-20

EMX array utility functions identifier format

Description

Customize generated identifiers for emxArray (embeddable mxArray) utility functions. The code generator produces emxArray types for variable-size arrays that use dynamically allocated memory. It produces emxArray utility functions that create and interact with variables that have an emxArray type. This parameter applies to MATLAB code in a MATLAB Function block, a Stateflow chart, or a System object™ associated with a MATLAB System block. This parameter does not apply to:

- Input or output signals
- Parameters
- Global variables
- Discrete state properties of System objects associated with a MATLAB System block

Category: Code Generation > Identifiers

Settings

Default: emx$M$N

Enter a macro that specifies whether, and in what order, certain text is to be included in the generated identifier. The macro can include a combination of the following format tokens.

Token	Description
$M	Insert name-mangling text if required to avoid naming collisions. Required.
$N	Insert the utility function name into identifier. For example, Init_real.
$R	Insert root model name into identifier, replacing unsupported characters with the underscore (_) character. Required for model referencing.

Tips

- The code generator applies the identifier format specified by this parameter before it applies the formats specified by other identifier format control parameters.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for name-mangling text.
- If you specify $R, the value that you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the $R and $M tokens.

Dependencies

This parameter:

- Appears only for ERT-based targets.

- Requires an Embedded Coder when generating code.

Command-Line Information

Parameter: `CustomSymbolStrEmxFcn`
Type: character vector
Value: valid combination of tokens
Default: `emx$M$N`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Identifiers”
- “Identifier Format Control”
- “Control Name Mangling in Generated Identifiers”
- “Identifier Format Control Parameters Limitations”

EMX array types identifier format

Description

Customize generated identifiers for emxArray (embeddable mxArray) types. The code generator produces emxArray types for variable-size arrays that use dynamically allocated memory. This parameter applies to MATLAB code in a MATLAB Function block, a Stateflow chart, or a System object associated with a MATLAB System block. This parameter does not apply to:

- Input or output signals
- Parameters
- Global variables
- Discrete state properties of System objects associated with a MATLAB System block

Category: Code Generation > Identifiers

Settings

Default: emxArray_MN

Enter a macro that specifies whether, and in what order, certain text is to be included in the generated identifier. The macro can include a combination of the following format tokens.

Token	Description
$M	Insert name-mangling text if required to avoid naming collisions. Required.
$N	Insert type name. For example, real_T
$R	Insert root model name into identifier, replacing unsupported characters with the underscore (_) character. Required for model referencing.

Tips

- The code generator applies the identifier format specified by this parameter before it applies the formats specified by other identifier format control parameters.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for name-mangling text.
- If you specify $R, the value you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the $R and $M tokens.

Dependencies

This parameter:

- Appears only for ERT-based targets.

- Requires an Embedded Coder when generating code.

Command-Line Information

Parameter: `CustomSymbolStrEmxType`
Type: character vector
Value: valid combination of tokens
Default: `emxArray_$M$N`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Identifiers”
- “Identifier Format Control”
- “Control Name Mangling in Generated Identifiers”
- “Identifier Format Control Parameters Limitations”

Code Generation Parameters: Data Type Replacement

Model Configuration Parameters: Code Generation Data Type Replacement

The **Code Generation > Data Type Replacement** category includes parameters for replacing built-in data type names with user-defined names in the generated code. On the Configuration Parameters dialog box, the following configuration parameters are on the **Code Generation > Data Type Replacement** pane.

Parameter	Description
"Replace data type names in the generated code" on page 16-5	Specify whether to replace built-in data type names with user-defined data type names in generated code.
"Replacement Name: double" on page 16-7	Specify a name for `double` built-in data types in generated code.
"Replacement Name: single" on page 16-9	Specify a name for `single` built-in data types in generated code.
"Replacement Name: int32" on page 16-11	Specify a name for `int32_T` built-in data types in generated code.
"Replacement Name: int16" on page 16-13	Specify a name for `int16_T` built-in data types in generated code.
"Replacement Name: int8" on page 16-15	Specify a name for `int8_T` built-in data types in generated code.
"Replacement Name: uint32" on page 16-17	Specify a name for `uint32_T` built-in data types in generated code.
"Replacement Name: uint16" on page 16-19	Specify a name for `uint16_T` built-in data types in generated code.
"Replacement Name: uint8" on page 16-21	Specify a name for `uint8_T` built-in data types in generated code.
"Replacement Name: boolean" on page 16-23	Specify a name for `boolean_T` built-in data types in generated code.
"Replacement Name: int" on page 16-25	Specify a name for `int_T` built-in data types in generated code.
"Replacement Name: uint" on page 16-27	Specify a name for `uint_T` built-in data types in generated code.
"Replacement Name: char" on page 16-29	Specify a name for `char_T` built-in data types in generated code.
"Replacement Name: uint64" on page 16-31	Specify a name for `uint64_T` built-in data types in generated code.
"Replacement Name: int64" on page 16-33	Specify a name for `int64_T` built-in data types in generated code.

Configure Data Type Replacements Programmatically

To programmatically replace the built-in data type names for your model, adjust the `ReplacementTypes` model parameter, which is a structure. This example code shows how to modify

the `ReplacementTypes` parameter to replace the built-in data type names `int8`, `uint8`, and `boolean` with the custom data type names `my_T_S8`, `my_T_U8`, and `my_T_BOOL`.

```
model = bdroot;
cs = getActiveConfigSet(model);
set_param(cs,'EnableUserReplacementTypes','on');

struc = get_param(cs,'ReplacementTypes');
struc.int8 = 'my_T_S8';
struc.uint8 = 'my_T_U8';
struc.boolean = 'my_T_BOOL';

set_param(cs,'ReplacementTypes',struc);
```

See Also

More About

- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- “Model Configuration”

Code Generation: Data Type Replacement Tab

Replace built-in data type names with user-defined replacement data type names in the generated code for your model.

Configuration

This tab is visible only if you specify an ERT-based system target file.

1. Select **Replace data type names in the generated code**.
2. In the **Replacement Name** fields, selectively specify replacement data type names to use for built-in Simulink data types.

To get help on an option

1. Right-click the option text label.
2. From the context menu, select **What's This**.

See Also

Related Examples

- "Model Configuration Parameters: Code Generation Data Type Replacement" on page 16-2
- "Control Data Type Names in Generated Code"
- "Data Type Replacement Limitations"

Replace data type names in the generated code

Description

Specify whether to replace built-in data type names with user-defined data type names in generated code.

Category: Code Generation > Data Type Replacement

Settings

Default: off

On

Displays the **Data type names** table. The table provides a way for you to replace the names of built-in data types used in generated code. This mechanism can be particularly useful for generating code that adheres to application or site data type naming standards.

You can choose to specify new data type names for some or all Simulink built-in data types listed in the table. Specify the replacement name as one of the following:

- A `Simulink.AliasType` object.
- A `Simulink.NumericType` object.
- The **Simulink Name** built-in data type name.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Off

Uses Simulink Coder names for built-in Simulink data types in generated code.

Dependencies

This parameter enables replacement for all built-in data type name in the **Data type names** table with user-defined data type names in generated code.

Command-Line Information

Parameter: `EnableUserReplacementTypes`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	On
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 16-2
- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- `Simulink.AliasType`
- `Simulink.NumericType`

Replacement Name: double

Description

Specify a name for `double` built-in data types in generated code.

Category: Code Generation > Data Type Replacement

Settings

Default: `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `real_T`.

Specify a character vector for the code generator to use as a name for `double` built-in data types.

Specify the replacement name as one of the following:

- A `Simulink.AliasType` object.
- A `Simulink.NumericType` object.
- The **Simulink Name** built-in data type name.

To use the **Simulink Name**, specify `double` in the **Replacement Name** column.

To replace the **Code Generation Name** for `double` with an object:

- For a `Simulink.AliasType` object, set the `BaseType` object property to `double`.
- For a `Simulink.NumericType` object, set the `DataTypeMode` object property to `Double`.
- Specify the object name in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Dependency

Replace data type names in the generated code on page 16-5 enables this parameter.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.double`
Type: character vector
Value: The **Simulink Name** , a `Simulink.AliasType` object, or a `Simulink.NumericType` object, where the object exists in the base workspace.
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 16-2
- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- `Simulink.AliasType`
- `Simulink.NumericType`

Replacement Name: single

Description

Specify a name for `single` built-in data types in generated code.

Category: Code Generation > Data Type Replacement

Settings

Default: `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `real32_T`.

Specify a character vector for the code generator to use as a name for `single` built-in data types.

Specify the replacement name as one of the following:

- A `Simulink.AliasType` object.
- A `Simulink.NumericType` object.
- The **Simulink Name** built-in data type name.

To use the **Simulink Name**, specify `single` in the **Replacement Name** column.

To replace the **Code Generation Name** for `single` with an object:

- For a `Simulink.AliasType` object, set the `BaseType` object property to `single`.
- For a `Simulink.NumericType` object, set the `DataTypeMode` object property to `Single`.
- Specify the object name in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Dependency

Replace data type names in the generated code on page 16-5 enables this parameter.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.single`
Type: character vector
Value: The **Simulink Name** , the name of a `Simulink.AliasType` object, or the name of a `Simulink.NumericType` object, where the object exists in the base workspace.
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 16-2
- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- `Simulink.AliasType`
- `Simulink.NumericType`

Replacement Name: int32

Description

Specify names for built-in Simulink data types in generated code.

Category: Code Generation > Data Type Replacement

Settings

Default: `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `int32_T`.

Specify character vectors for the code generator to use as names for built-in Simulink data types.

- A `Simulink.AliasType` object.
- The **Simulink Name** built-in data type name.

To replace the **Code Generation Name** `int32_T`:

- For a `Simulink.AliasType` object, set the `BaseType` object property to `int32`.
- To use the built-in data type name that matches the **Code Generation Name**, specify `int32` in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Dependency

Replace data type names in the generated code on page 16-5 enables this parameter.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.int32`
Type: character vector
Value: The **Simulink Name** or the name of a `Simulink.AliasType` object, where the object exists in the base workspace.
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact

Application	Setting
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 16-2
- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- `Simulink.AliasType`

Replacement Name: int16

Description

Specify names for built-in Simulink data types in generated code.

Category: Code Generation > Data Type Replacement

Settings

Default: `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `int16_T`.

Specify character vectors for the code generator to use as names for built-in Simulink data types.

- A `Simulink.AliasType` object.
- The **Simulink Name** built-in data type name.

To replace the **Code Generation Name** `int16_T`:

- For a `Simulink.AliasType` object, set the `BaseType` object property to `int16`.
- To use the built-in data type name that matches the **Code Generation Name**, specify `int16` in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Dependency

Replace data type names in the generated code on page 16-5 enables this parameter.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.int16`
Type: character vector
Value: The **Simulink Name** or the name of a `Simulink.AliasType` object, where the object exists in the base workspace.
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact

Application	Setting
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 16-2
- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- `Simulink.AliasType`

Replacement Name: int8

Description

Specify names for built-in Simulink data types in generated code.

Category: Code Generation > Data Type Replacement

Settings

Default: `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `int8_T`.

Specify character vectors for the code generator to use as names for built-in Simulink data types.

To replace the **Code Generation Name** `int8_T`:

- A `Simulink.AliasType` object.
- The **Simulink Name** built-in data type name.

- For a `Simulink.AliasType` object, set the `BaseType` object property to `int8`.
- To use the built-in data type name that matches the **Code Generation Name**, specify `int8` in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Dependency

Replace data type names in the generated code on page 16-5 enables this parameter.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.int8`
Type: character vector
Value: The **Simulink Name** or the name of a `Simulink.AliasType` object, where the object exists in the base workspace.
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact

Application	Setting
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 16-2
- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- `Simulink.AliasType`

Replacement Name: uint32

Description

Specify names for built-in Simulink data types in generated code.

Category: Code Generation > Data Type Replacement

Settings

Default: `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `uint32_T`.

Specify character vectors for the code generator to use as names for built-in Simulink data types.

- A `Simulink.AliasType` object.
- The **Simulink Name** built-in data type name.

To replace the **Code Generation Name** `uint32_T`:

- For a `Simulink.AliasType` object, set the `BaseType` object property to `uint32`.
- To use the built-in data type name that matches the **Code Generation Name**, specify `uint32 c` in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Dependency

Replace data type names in the generated code on page 16-5 enables this parameter.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.uint32`
Type: character vector
Value: The **Simulink Name** or the name of a `Simulink.AliasType` object, where the object exists in the base workspace.
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact

Application	Setting
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 16-2
- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- `Simulink.AliasType`

Replacement Name: uint16

Description

Specify names for built-in Simulink data types in generated code.

Category: Code Generation > Data Type Replacement

Settings

Default: `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `uint16_T`.

Specify character vectors for the code generator to use as names for built-in Simulink data types.

- A `Simulink.AliasType` object.
- The **Simulink Name** built-in data type name.

To replace the **Code Generation Name** `uint16_T`:

- For a `Simulink.AliasType` object, set the `BaseType` object property to `uint16`.
- To use the built-in data type name that matches the **Code Generation Name**, specify `uint16` in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Dependency

Replace data type names in the generated code on page 16-5 enables this parameter.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.uint16`
Type: character vector
Value: The **Simulink Name** or the name of a `Simulink.AliasType` object, where the object exists in the base workspace.
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact

Application	Setting
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 16-2
- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- `Simulink.AliasType`

Replacement Name: uint8

Description

Specify names for built-in Simulink data types in generated code.

Category: Code Generation > Data Type Replacement

Settings

Default: `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `uint8_T`.

Specify character vectors for the code generator to use as names for built-in Simulink data types.

- A `Simulink.AliasType` object.
- The **Simulink Name** built-in data type name.

To replace the **Code Generation Name** `uint8_T`:

- For a `Simulink.AliasType` object, set the `BaseType` object property to `uint8`.
- To use the built-in data type name that matches the **Code Generation Name**, specify `uint8` in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Dependency

Replace data type names in the generated code on page 16-5 enables this parameter.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.uint8`
Type: character vector
Value: The **Simulink Name** or the name of a `Simulink.AliasType` object, where the object exists in the base workspace.
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact

Application	Setting
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 16-2
- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- `Simulink.AliasType`

Replacement Name: boolean

Description

Specify names for built-in Simulink data types in generated code.

Category: Code Generation > Data Type Replacement

Settings

Default: `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `boolean_T`.

Specify character vectors for the code generator to use as names for built-in Simulink data types.

For ERT S-functions, the replacement data type can be only an 8-bit integer, `int8`, or `uint8`.

Specify the replacement name as one of the following:

- A `Simulink.AliasType` object.
- A `Simulink.NumericType` object.
- The **Simulink Name** built-in data type name.

To replace the **Code Generation Name** `boolean_T`:

- For a `Simulink.AliasType` object, set the `BaseType` object property to `boolean`, `uint8` , `int8`, or `int`*n*, where *n* is the number of bits set for **Configuration Parameters** > **Hardware Implementation** > **Number of bits: int**. Specify the name of the `Simulink.AliasType` object in the **Replacement Name** column.
- For a `Simulink.NumericType` object, to replace `real_T`, set the `DataTypeMode` object property to `Boolean`. Specify the name of the `Simulink.NumericType` object in the **Replacement Name** column.
- To use the Simulink Name built-in data type name which matches the Code Generation name, in the **Replacement Name** column, specify `uint8`, `int8`, or `int`*n*, where *n* is the number of bits set for **Configuration Parameters > Hardware Implementation > Number of bits: int**.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Dependency

Replace data type names in the generated code on page 16-5 enables this parameter.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.boolean`
Type: character vector

Value: The **Simulink Name** , a `Simulink.AliasType` object, or a `Simulink.NumericType` object, where the object exists in the base workspace.
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 16-2
- “Replace boolean with Specific Integer Data Type”
- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- `Simulink.AliasType`
- `Simulink.NumericType`

Replacement Name: int

Description

Specify names for built-in Simulink data types in generated code.

Category: Code Generation > Data Type Replacement

Settings

Default: `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `int_T`.

Specify character vectors for the code generator to use as names for built-in Simulink data types.

Specify the replacement name as one of the following:

- A `Simulink.AliasType` object.
- The **Simulink Name** built-in data type name.

To replace the **Code Generation Name** `int_T`:

- For a `Simulink.AliasType` object

  Set the `BaseType` object property to `int`*n*. Specify the name of the `Simulink.AliasType` object in the **Replacement Name** column.
- To use the **Simulink Name** for int_T, in the **Replacement Name** column, specify `int`*n*.

n is the number of bits displayed in the Configuration Parameters dialog box, **Hardware Implementation** pane > **Number of bits: int**.

An error occurs, if

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Dependency

Replace data type names in the generated code on page 16-5 enables this parameter.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.int`
Type: character vector
Value: The **Simulink Name** or the name of a `Simulink.AliasType`, where the object exists in the base workspace.
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid value
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 16-2
- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- `Simulink.AliasType`

Replacement Name: uint

Description

Specify names to use for built-in Simulink data types in generated code.

Category: Code Generation > Data Type Replacement

Settings

Default: `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `uint_T`.

Specify character vectors for the code generator to use as names for built-in Simulink data types.

Specify the replacement name as one of the following:

- A `Simulink.AliasType` object.
- The **Simulink Name** built-in data type name.

To replace the **Code Generation Name** `uint_T`:

- For a `Simulink.AliasType` object

  Set the `BaseType` object property to `uint`*n*. Specify the name of the `Simulink.AliasType` object in the **Replacement Name** column.
- To use the **Simulink Name** for `uint_T`, in the **Replacement Name** column, specify `uint`*n*.

n is the number of bits displayed in the Configuration Parameters dialog box, **Hardware Implementation** pane > **Number of bits: int**.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Dependency

Replace data type names in the generated code on page 16-5 enables this parameter.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.uint`
Type: character vector
Value: The **Simulink Name** or the name of a `Simulink.NumericType`, where the object exists in the base workspace.
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 16-2
- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- `Simulink.AliasType`

Replacement Name: char

Description

Specify names for built-in Simulink data types in generated code.

Category: Code Generation > Data Type Replacement

Settings

Default: `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `char_T`.

Specify character vectors for the code generator to use as names for built-in Simulink data types.

To replace the **Code Generation Name** `char_T`, create a `Simulink.AliasType` object in the Command Window.

Set the `BaseType` object property to `int`*n*. Specify the name of the `Simulink.AliasType` object in the **Replacement Name** column. *n* is the number of bits set for **Configuration Parameters > Hardware ImplementationNumber of bits: char**.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Dependency

Replace data type names in the generated code on page 16-5 enables this parameter.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.char`
Type: character vector
Value: The name of a `Simulink.AliasType` object, where the object exists in the base workspace.
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 16-2
- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- `Simulink.AliasType`

Replacement Name: uint64

Description

Specify a name for a 64-bit unsigned integer Simulink data type in generated code.

Category: Code Generation > Data Type Replacement

Settings

Default: `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `uint64_T`.

Specify character vectors for the code generator to use as names for 64-bit unsigned integer Simulink data types.

Specify the replacement name as one of the following:

- A `Simulink.NumericType` object.
- The **Simulink Name** built-in data type name.

To replace the **Code Generation Name** `uint64_T`:

- For a `Simulink.NumericType` object, set these properties:
  - `DataTypeMode` - `Fixed-point: binary point scaling`
  - `Signedness` - `Unsigned`
  - `WordLength` - `64`
  - `IsAlias` - `true`
- To use the built-in data type name that matches the **Code Generation Name**, specify `uint64` in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The object is `Simulink.AliasType` .

Dependency

Replace data type names in the generated code on page 16-5 enables this parameter.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.uint64`
Type: character vector
Value: The name of a `Simulink.NumericType` object, where the object exists in the base workspace.
Default: `''`

Recommended Settings

Application	Setting
Debugging	Noimpact
Traceability	A valid character vector
Efficiency	Noimpact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 16-2
- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- `Simulink.NumericType`

Replacement Name: int64

Description

Specify a name for a 64-bit integer Simulink data type in generated code.

Category: Code Generation > Data Type Replacement

Settings

Default: `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `int64_T`.

Specify character vectors for the code generator to use as names for 64-bit integer Simulink data types.

Specify the replacement name as one of the following:

- A `Simulink.NumericType` object.
- The **Simulink Name** built-in data type name.

To replace the **Code Generation Name** `int64_T`:

- For a `Simulink.NumericType` object, set these properties:
  - `DataTypeMode` - `Fixed-point: binary point scaling`
  - `Signedness` - `Signed`
  - `WordLength` - `64`
  - `IsAlias` - `true`
- To use the built-in data type name that matches the **Code Generation Name**, specify `int64` in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The object is `Simulink.AliasType`.

Dependency

Replace data type names in the generated code on page 16-5 enables **Replacement Name: int64** parameter.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.int64`
Type: character vector
Value: The name of a `Simulink.NumericType` object, where the object exists in the base workspace.
Default: `''`

Recommended Settings

Application	Setting
Debugging	Noimpact
Traceability	A valid character vector
Efficiency	Noimpact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 16-2
- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- `Simulink.NumericType`

Memory Sections Parameters on the Code Generation Pane

Code Generation: Memory Sections Tab Overview

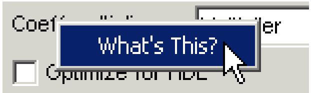
Insert comments and pragmas into the generated code for data and functions.

Configuration

This tab appears only if you specify an ERT based system target file.

To get help on an option

1. Right-click the option text label.
2. From the context menu, select **What's This**.



See Also

Related Examples

- “Model Configuration Parameters: Code Generation”
- “Control Data and Function Placement in Memory by Inserting Pragmas”

Package

Description

Specify a package that contains memory sections you want to apply to model-level functions and internal data.

Category: Code Generation

Settings

Memory section specifications for model-level functions and internal data apply to the top level of the model and to subsystems except atomic subsystems that contain overriding memory section specifications.

Default: `---None---`

`---None---`

Suppresses memory sections.

`Simulink`

Applies the built-in Simulink package.

`mpt`

Applies the built-in mpt package.

Tip

If you have defined packages of your own, click **Refresh package list**. This action adds user-defined packages on your search path to the package list.

Command-Line Information

Parameter: `MemSecPackage`
Type: character vector
Value: `'--- None ---'` | `'Simulink'` | `'mpt'`
Default: `'--- None ---'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation”
- “Control Data and Function Placement in Memory by Inserting Pragmas”

Refresh package list

Description

Add user-defined packages that are on the search path to list of packages displayed by **Packages**.

Category: Code Generation

Tip

If you have defined packages of your own, click **Refresh package list**. This action adds user-defined packages on your search path to the package list.

See Also

Related Examples

- "Model Configuration Parameters: Code Generation"
- "Control Data and Function Placement in Memory by Inserting Pragmas"

Initialize/Terminate

Description

Specify whether to apply a memory section to Initialize/Start and Terminate functions.

Category: Code Generation

Settings

Memory section specifications for model-level functions and internal data apply to the top level of the model and to subsystems except atomic subsystems that contain overriding memory section specifications.

Default: `Default`

`Default`
: Suppresses the use of a memory section for Initialize, Start, and Terminate functions.

`memory-section-name`
: Applies a memory section to Initialize, Start, and Terminate functions.

Command-Line Information

Parameter: `MemSecFuncInitTerm`
Type: character vector
Value: `'Default'` | `'MemConst'` | `'MemVolatile'` | `'MemConstVolatile'`
Default: `'Default'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation”
- “Control Data and Function Placement in Memory by Inserting Pragmas”

Execution

Description

Specify whether to apply a memory section to execution functions.

Category: Code Generation

Settings

Memory section specifications for model-level functions and internal data apply to the top level of the model and to subsystems except atomic subsystems that contain overriding memory section specifications.

Default: `Default`

`Default`

Suppresses the use of a memory section for Step, Run-time initialization, Derivative, Enable, and Disable functions.

`memory-section-name`

Applies a memory section to Step, Run-time initialization, Derivative, Enable, and Disable functions.

Command-Line Information

Parameter: `MemSecFuncExecute`
Type: character vector
Value: `'Default'` | `'MemConst'` | `'MemVolatile'` | `'MemConstVolatile'`
Default: `'Default'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation”
- “Control Data and Function Placement in Memory by Inserting Pragmas”

Shared utility

Description

Specify whether to apply memory sections to shared utility functions.

Category: Code Generation

Settings

Memory section specifications for model-level functions and internal data apply to the top level of the model and to subsystems except atomic subsystems that contain overriding memory section specifications.

Default: `Default`

`Default`

Suppresses the use of memory sections for shared utility functions.

`memory-section-name`

Applies a memory section to shared utility functions, such as fixed-point functions, lookup table functions, binary search functions, and MATLAB functions defined outside MATLAB Function blocks.

Command-Line Information

Parameter: `MemSecFuncSharedUtil`
Type: character vector
Value: `'Default'` | `'MemConst'` | `'MemVolatile'` | `'MemConstVolatile'`
Default: `'Default'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- "Model Configuration Parameters: Code Generation"
- "Control Data and Function Placement in Memory by Inserting Pragmas"

Constants

Description

Specify whether to apply a memory section to constants.

Category: Code Generation

Settings

Memory section specifications for model-level functions and internal data apply to the top level of the model and to subsystems except atomic subsystems that contain overriding memory section specifications.

Default: `Default`

`Default`

Suppresses the use of a memory section for constants.

`memory-section-name`

Applies a memory section to constants.

This parameter applies to the generated global data structures that contain:

- Constant parameters
- Constant block I/O

For basic information about the global data structures generated for models, see “Data Structures in the Generated Code”.

Command-Line Information

Parameter: `MemSecDataConstants`
Type: character vector
Value: `'Default'` | `'MemConst'` | `'MemVolatile'` | `'MemConstVolatile'`
Default: `'Default'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation”
- “Control Data and Function Placement in Memory by Inserting Pragmas”

Inputs/Outputs

Description

Specify whether to apply a memory section to root input and output.

Category: Code Generation

Settings

Memory section specifications for model-level functions and internal data apply to the top level of the model and to subsystems except atomic subsystems that contain overriding memory section specifications.

Default: `Default`

`Default`

Suppresses the use of a memory section for root-level input and output.

`memory-section-name`

Applies a memory section for root-level input and output.

This parameter applies to the generated global data structures that contain:

- Root-level inputs
- Root-level outputs

For basic information about the global data structures generated for models, see “Data Structures in the Generated Code”.

Command-Line Information

Parameter: `MemSecDataIO`
Type: character vector
Value: `'Default'` | `'MemConst'` | `'MemVolatile'` | `'MemConstVolatile'`
Default: `'Default'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation”
- “Control Data and Function Placement in Memory by Inserting Pragmas”

Internal data

Description

Specify whether to apply a memory section to internal data.

Category: Code Generation

Settings

Memory section specifications for model-level functions and internal data apply to the top level of the model and to subsystems except atomic subsystems that contain overriding memory section specifications.

Default: `Default`

`Default`

Suppresses the use of a memory section for internal data.

`memory-section-name`

Applies a memory section for internal data.

This parameter applies to the generated global data structures that contain:

- Block I/O
- DWork vectors
- Zero-crossings

For basic information about the global data structures generated for models, see “Standard Data Structures”.

Command-Line Information

Parameter: `MemSecDataInternal`
Type: character vector
Value: `'Default'` | `'MemConst'` | `'MemVolatile'` | `'MemConstVolatile'`
Default: `'Default'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation”
- “Control Data and Function Placement in Memory by Inserting Pragmas”

Parameters

Description

Specify whether to apply a memory section to parameters.

Category: Code Generation

Settings

Memory section specifications for model-level functions and internal data apply to the top level of the model and to subsystems except atomic subsystems that contain overriding memory section specifications.

Default: `Default`

`Default`

Suppress the use of a memory section for parameters.

`memory-section-name`

Apply memory section for parameters.

This parameter applies to the generated global data structure that contains block parameter data.

For basic information about the global data structures generated for models, see “Standard Data Structures”.

Command-Line Information

Parameter: `MemSecDataParameters`
Type: character vector
Value: `'Default'` | `'MemConst'` | `'MemVolatile'` | `'MemConstVolatile'`
Default: `'Default'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation”
- Memory Sections

Validation results

Description

Display the results of memory section validation.

Category: Code Generation

Settings

The code generation software checks and reports whether the currently chosen package is on the MATLAB path and that the selected memory sections exist inside the package.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation”

18

Code Generation Parameters: Templates

Model Configuration Parameters: Code Generation Templates

The **Code Generation** > **Templates** category includes parameters for customizing the organization of your generated code. On the Configuration Parameters dialog box, the following configuration parameters are on the **Code Generation** > **Templates** pane.

Parameter	Description
“Code templates: Source file (*.c) template” on page 18-4	Specify the code generation template (CGT) file to use when generating a source code file.
“Code templates: Header file (*.h) template” on page 18-5	Specify the code generation template (CGT) file to use when generating a code header file.
“Data templates: Source file (*.c) template” on page 18-6	Specify the code generation template (CGT) file to use when generating a data source file.
“Data templates: Header file (*.h) template” on page 18-7	Specify the code generation template (CGT) file to use when generating a data header file.
“File customization template” on page 18-8	Specify the custom file processing (CFP) template file to use when generating code.
“Generate an example main program” on page 18-9	Control whether to generate an example main program for a model.
“Target operating system” on page 18-11	Specify a target operating system to use when generating model-specific example main program module.

The following parameters on **Advanced parameters** section are infrequently used and have no other documentation.

Parameter	Description
`GenerateFullHeader`	Generate full header including time stamp. For GRT targets, this parameter is on the **Code Generation > Interface** pane.
`ERTCustomFileBanners`	If this option is cleared, the configurations for Code and Data templates are ignored.

See Also

More About

- “Model Configuration”

Code Generation: Templates Tab Overview

Customize the organization of your generated code.

Configuration

This tab appears only if you specify an ERT based system target file.

To get help on an option

1. Right-click the option text label.
2. From the context menu, select **What's This**.

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Templates” on page 18-2

Code templates: Source file (*.c) template

Description

Specify the code generation template (CGT) file to use when generating a source code file.

Category: Code Generation > Templates

Settings

Default: `ert_code_template.cgt`

You can use a CGT file to define the top-level organization and formatting of generated source code files (`.c` or `.cpp`).

Note The CGT file must be located on the MATLAB path.

Command-Line Information

Parameter: `ERTSrcFileBannerTemplate`
Type: character vector
Value: valid CGT file
Default: `'ert_code_template.cgt'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- "Model Configuration Parameters: Code Generation Templates" on page 18-2
- Selecting and Defining Templates
- Custom File Processing

Code templates: Header file (*.h) template

Description

Specify the code generation template (CGT) file to use when generating a code header file.

Category: Code Generation > Templates

Settings

Default: `ert_code_template.cgt`

You can use a CGT file to define the top-level organization and formatting of generated header files (`.h`).

Note The CGT file must be located on the MATLAB path.

Command-Line Information

Parameter: `ERTHdrFileBannerTemplate`
Type: character vector
Value: valid CGT file
Default: `'ert_code_template.cgt'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Templates” on page 18-2
- Selecting and Defining Templates
- Custom File Processing

Data templates: Source file (*.c) template

Description

Specify the code generation template (CGT) file to use when generating a data source file.

Category: Code Generation > Templates

Settings

Default: `ert_code_template.cgt`

You can use a CGT file to define the top-level organization and formatting of generated data source files (`.c` or `.cpp`) that contain definitions of variables of global scope.

Note The CGT file must be located on the MATLAB path.

Command-Line Information

Parameter: `ERTDataSrcFileTemplate`
Type: character vector
Value: valid CGT file
Default: `'ert_code_template.cgt'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Templates” on page 18-2
- Selecting and Defining Templates
- Custom File Processing

Data templates: Header file (*.h) template

Description

Specify the code generation template (CGT) file to use when generating a data header file.

Category: Code Generation > Templates

Settings

Default: `ert_code_template.cgt`

You can use a CGT file to define the top-level organization and formatting of generated data header files (`.h`) that contain declarations of variables of global scope.

Note The CGT file must be located on the MATLAB path.

Command-Line Information

Parameter: `ERTDataHdrFileTemplate`
Type: character vector
Value: valid CGT file
Default: `'ert_code_template.cgt'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Templates” on page 18-2
- Selecting and Defining Templates
- Custom File Processing

File customization template

Description

Specify the custom file processing (CFP) template file to use when generating code.

Category: Code Generation > Templates

Settings

Default: `'example_file_process.tlc'`

You can use a CFP template file to customize generated code. A CFP template file is a TLC file that organizes types of code (for example, includes, `typedefs`, and functions) into sections. The primary purpose of a CFP template is to assemble code to be generated into buffers, and to call a code template API to emit the buffered code into specified sections of generated source and header files. The CFP template file must be located on the MATLAB path.

Command-Line Information

Parameter: `ERTCustomFileTemplate`
Type: character vector
Value: valid TLC file
Default: `'example_file_process.tlc'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Templates” on page 18-2
- Selecting and Defining Templates
- Custom File Processing

Generate an example main program

Description

Control whether to generate an example main program for a model.

Category: Code Generation > Templates

Settings

Default: on

On

Generates an example main program, `ert_main.c` (or `.cpp`). The file includes:

- The `main()` function for the generated program
- Task scheduling code that determines how and when block computations execute on each time step of the model

The operation of the main program and the scheduling algorithm employed depend primarily on whether your model is single-rate or multirate, and also on your model's solver mode (`SingleTasking` or `MultiTasking`).

Off

Does not generate an example main program.

Note The software provides static versions of the main file, *`matlabroot`*`/rtw/c/src/common/rt_main.c` and *`matlabroot`*`/rtw/c/src/common/rt_cppclass_main.cpp`, as a basis for custom modifications. You can use either static main file as a template for developing embedded applications.

Tips

- After you generate and customize the main program, clear this parameter to prevent regenerating the main module and overwriting your customized version.
- You can use a custom file processing (CFP) template file to override normal main program generation, and generate a main program module customized for your target environment.

Dependencies

- This parameter enables **Target operating system**.
- You must enable this parameter and select `VxWorksExample` for **Target operating system** if you use VxWorks[3] library blocks.

3. VxWorks is a registered trademark of Wind River® Systems, Inc.

Command-Line Information

Parameter: `GenerateSampleERTMain`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Templates” on page 18-2
- “Deploy Generated Standalone Executable Programs To Target Hardware”
- Custom File Processing

Target operating system

Description

Specify a target operating system to use when generating model-specific example main program module.

Category: Code Generation > Templates

Settings

Default: `BareBoardExample`

`BareBoardExample`

Generates a bareboard main program designed to run under control of a real-time clock, without a real-time operating system.

`VxWorksExample`

Generates a fully commented example showing how to deploy the code under the VxWorks real-time operating system.

`NativeThreadsExample`

Generates a fully commented example showing how to deploy the threaded code under the host operating system. This option requires you to configure your model for concurrent execution.

Dependencies

- This parameter is enabled by **Generate an example main program**.
- This parameter must be the same for top-level and referenced models.

Command-Line Information

Parameter: `TargetOS`
Type: character vector
Value: `'BareBoardExample'` | `'VxWorksExample'` | `'NativeThreadsExample'`
Default: `'BareBoardExample'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Templates” on page 18-2
- “Deploy Generated Standalone Executable Programs To Target Hardware”
- Custom File Processing

Code Generation Parameters: Verification

Model Configuration Parameters: Code Generation Verification

The **Code Generation > Verification** category includes code verification and performance analysis parameters for SIL and PIL simulations. These parameters require an Embedded Coder license.

Parameter	Description
"Measure task execution time" on page 19-4	Measure execution times and generate metrics for tasks in generated code.
"Measure function execution times" on page 19-6	Measure execution times and generate metrics for functions inside generated code.
"Workspace variable" on page 19-8	Specify workspace variable that collects measurements and allows viewing and analysis of execution profiles.
"Save options" on page 19-9	Specify whether to save code profiling measurement and analysis data to base workspace.
"Third-party tool" on page 19-11	Specify a third-party tool for code coverage.
"Enable portable word sizes" on page 19-13	Allow portability across host and target processors that support different word sizes.
"Enable source-level debugging for SIL" on page 19-15	Allow debugging of generated code during a SIL simulation.

This parameter belongs to the **Advanced parameters** category.

Parameter	Description
"Create block" on page 19-16	Generate a SIL or PIL block.

See Also

More About

- "Numerical Equivalence Testing"
- "Code Execution Profiling"
- "Code Coverage"
- "Model Configuration"

Code Generation: Verification Tab Overview

Create SIL block and configure word size portability, code coverage for SIL testing, and code execution profiling

Configuration

This tab appears only if you specify an ERT–based system target file.

To get help on an option

1. Right-click the option text label.
2. From the context menu, select **What's This**.

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Verification” on page 19-2
- “SIL and PIL Simulations”

Measure task execution time

Description

Measure execution times and generate metrics for tasks in generated code.

Category: Code Generation > Verification

Settings

Default: off

On

During SIL and PIL simulations, collect execution-time measurements for tasks. The software obtains data from instrumentation in the SIL or PIL application.

Off

Do not collect measurements of execution times

Dependencies

When you use this parameter, you must also specify a workspace variable. The software uses this variable to collect execution-time measurements.

In a model reference hierarchy, the top-model parameter value applies to the whole hierarchy. The software ignores the value of this parameter in referenced models.

Command-Line Information

Parameter: `CodeExecutionProfiling`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Off
Safety precaution	No recommendation

See Also

Related Examples

- "Model Configuration Parameters: Code Generation Verification" on page 19-2

- “Code Execution Profiling with SIL and PIL”
- “View and Compare Code Execution Times”
- “Analyze Code Execution Data”

Measure function execution times

Description

Measure execution times and generate metrics for functions inside generated code.

Category: Code Generation > Verification

Settings

Default: `Off`

`Off`

No function-level instrumentation, so execution times for functions in generated code are not collected.

`Coarse (referenced models and subsystems only)`

Measure execution times only for function code generated from referenced models and subsystems.

`Detailed (all function call sites)`

Measure execution times for all functions in generated code.

Dependencies

To use this parameter, you must also select **Measure task execution time** for the top model of the model reference hierarchy.

For a model in a reference hierarchy, the software does not support simultaneous function execution-time measurement and code coverage.

Command-Line Information

Parameter: `CodeProfilingInstrumentation`
Type: character vector
Value: `'off'` | `'coarse'` | `'detailed'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Off
Safety precaution	No recommendation

See Also

Related Examples

- "Model Configuration Parameters: Code Generation Verification" on page 19-2
- "Code Execution Profiling with SIL and PIL"
- "View and Compare Code Execution Times"
- "Analyze Code Execution Data"

Workspace variable

Description

Specify workspace variable that collects measurements and allows viewing and analysis of execution profiles.

Category: Code Generation > Verification

Settings

Default: `executionProfile`

When you run simulation, software generates specified workspace variable as an `coder.profile.ExecutionTime` object. To view and analyze execution profiles, use methods from the `coder.profile.ExecutionTime` and `coder.profile.ExecutionTimeSection` classes.

Dependency

You can only specify this parameter if you select the **Measure task execution time** check box. Otherwise the field appears dimmed.

Command-Line Information

Parameter: `CodeExecutionProfileVariable`
Type: character vector
Value: valid MATLAB variable name
Default: none

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Valid MATLAB variable name
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- "Model Configuration Parameters: Code Generation Verification" on page 19-2
- "Code Execution Profiling with SIL and PIL"
- "View and Compare Code Execution Times"
- "Analyze Code Execution Data"

Save options

Description

Options for saving code profiling measurement and analysis data.

Category: Code Generation > Verification

Settings

Default: `Summary data only`

`Summary data only`

Save only code profiling summary data to a `coder.profile.ExecutionTime` object in the base workspace. Use this option to limit the amount of data that the software saves to the base workspace. For example, if you are concerned that your computer might not have enough memory to store the time measurements for a long simulation. The software calculates metrics for the code execution report as the simulation proceeds, without saving raw data to memory. To view these metrics, use the `coder.profile.ExecutionTime` `report` method.

Selecting this value disables the streaming of execution times to the Simulation Data Inspector during simulations.

`All data`

Save the code profiling measurement and analysis data to a `coder.profile.ExecutionTime` object in the base workspace. In addition to viewing the code execution report, this option allows you to analyze data using the `coder.profile.ExecutionTime` and `coder.profile.ExecutionTimeSection` methods.

`Metrics only`

During the simulation, store only these profiling metrics on the target hardware:

- Maximum execution time of code section
- Average execution time of code section
- Number of calls to code section

At the end of the simulation, Simulink uploads the stored metrics from the target hardware to your development computer.

The storage of profiling data on the target hardware during the simulation reduces bandwidth usage for the communication channel between Simulink and the target application.

This option does not support overhead filtering.

Dependency

You can only specify this parameter if you select the **Measure task execution time** check box. Otherwise the field appears dimmed.

Command-Line Information

Parameter: `CodeProfilingSaveOptions`
Type: character vector
Value: `'SummaryOnly'` | `'AllData'` | `'MetricsOnly'`
Default: `'SummaryOnly'`

Recommended Settings

Application	Setting
Debugging	`All data`
Traceability	`All data`
Efficiency	`Summary data only`
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Verification” on page 19-2
- “Code Execution Profiling with SIL and PIL”
- “View and Compare Code Execution Times”
- “Analyze Code Execution Data”
- “Remove Instrumentation Overheads from Execution Time Measurements”
- “Capture Main Profiling Metrics on Target Hardware”

Third-party tool

Description

Specify a third-party tool for code coverage.

Category: Code Generation > Verification

Settings

Default: `None (use Simulink Coverage)`

`None (use Simulink Coverage)`

No third-party tool specified for code coverage. You can use Simulink Coverage™ to analyze code coverage.

`BullseyeCoverage`

Specifies the BullseyeCoverage tool from Bullseye Testing Technology

`LDRA Testbed`

Specifies the LDRA Testbed® tool from LDRA Software Technology

Dependencies

Code coverage is not supported if the **Create block** configuration parameter is either `SIL` or `PIL`.

If you do not specify a third-party tool, **Configure Coverage** appears dimmed. Otherwise, click **Configure Coverage** to open the Code Coverage Settings dialog box.

Command-Line Information

Parameter: `CoverageTool` field of `CodeCoverageSettings`
Type: character vector
Value: `'None'` | `'BullseyeCoverage'` | `'LDRA Testbed'`
Default: `'None'`

Tip To access the `CoverageTool` value, type:

```
covSettings = get_param(gcs, 'CodeCoverageSettings');
covSettings.CoverageTool
```

Recommended Settings

Application	Setting
Debugging	`BullseyeCoverage` or `LDRA Testbed`
Traceability	`BullseyeCoverage` or `LDRA Testbed`
Efficiency	`None` (code coverage off)

Application	Setting
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Verification” on page 19-2
- “Configure Code Coverage with Third-Party Tools”
- “Configure Code Coverage Programmatically”

Enable portable word sizes

Description

Allow portability across host and target processors that support different word sizes.

You can enable portable word sizes to support software-in-the-loop (SIL) testing of your generated code. For a SIL simulation, you use the top-model or Model block SIL simulation mode, or select `SIL` in the **Configuration Parameters** > **Create block** field.

Category: Code Generation > Verification

Settings

Default: off

On

Generate conditional processing macros to support compilation of generated code on a processor that supports a different word size than the target processor on which production code is run. This option allows you to use the same generated code for SIL testing on your development computer and production deployment on the target processor. For example, you can perform SIL testing on a 64-bit development computer and deploy the code on a 16-bit target processor.

Off

Does not generate portable code.

Dependencies

When you use this option, you should select **Test hardware is the same as production hardware** on the **Hardware Implementation** pane.

Command-Line Information

Parameter: `PortableWordSizes`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Off
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Verification” on page 19-2
- “Configure Hardware Implementation Settings”

Enable source-level debugging for SIL

Description

Allow debugging of generated code during a SIL simulation.

Category: Code Generation > Verification

Settings

Default: off

☑ On

Source-level debugging is enabled.

☐ Off

Source-level debugging is disabled.

Command-Line Information

Parameter: `SILDebugging`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Off
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Verification” on page 19-2
- “Debug Generated Code During SIL Simulation”

Create block

Description

Generate a SIL or PIL block

Category: Code Generation > Verification

Settings

Default: `None`

`None`
: SIL or PIL block not generated.

`SIL`
: Generate a SIL block that represents a top-model or subsystem.

  If you select this option, the software creates and opens an untitled model with a SIL block. The SIL block contains an S-function, through which the software runs compiled object code on the host computer. With this block, you can verify the behavior of source code generated from top-model or subsystem components.

  If the subsystem is an export-function subsystem, the software creates a Model block with **Simulation mode** set to `Software-in-the-loop (SIL)`.

`PIL`
: Generate a PIL block that represents a top-model or subsystem.

  If you select this option, the software creates and opens an untitled model with a PIL block. The PIL block contains an S-function, through which the software runs cross-compiled object code on a target processor or instruction set simulator. With this block, you can verify the behavior of object code generated from top-model or subsystem components.

  If the subsystem is an export-function subsystem, the software creates a Model block with **Simulation mode** set to `Processor-in-the-loop (PIL)`.

  To control the way code compiles and executes in the target environment, use Target Connectivity API.

Command-Line Information

Parameter: `CreateSILPILBlock`
Type: character vector
Value: `'None'` | `'SIL'` | `'PIL'`
Default: `'None'`

Recommended Settings

Application	Setting
Debugging	On

Application	Setting
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Simulation with Subsystem Blocks”
- “Generate Code for Export-Function Subsystems”
- “SIL and PIL Simulations”
- “Generate Code for Export-Function Subsystems”

20

Configuration Parameters

Recommended Settings Summary for Model Configuration Parameters

The following tables summarizes the impact of each configuration parameter on debugging, traceability, efficiency, and safety considerations, and indicates the factory default configuration settings for the ERT target. The Simulink Coder configuration parameters are documented in "Recommended Settings Summary for Model Configuration Parameters". For additional details, click the links in the Configuration Parameter column.

Mapping of Application Requirements to the Optimization Pane

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
Application lifespan (days)	No impact	No impact	Optimal finite value	`inf`	`auto`
Optimize using the specified minimum and maximum values on page 10-42	Off	Off	On	No impact	Off
Remove root level I/O zero initialization on page 10-7	No impact	No impact	On (GUI) `off` (command line) (execution, ROM), No impact (RAM)	No recommendation	On
Remove internal data zero initialization on page 10-9	No impact	No impact	On (execution, ROM)	No recommendation	On
Remove Code from Tunable Parameter Expressions That Saturates Out-of-Range Values on page 10-45	Off	Off	On (execution, ROM)	No recommendation	On
Remove code that protects against division arithmetic exceptions on page 10-47	No impact	No impact	On (execution, ROM)	Off	Off
Pack Boolean data into bitfields on page 10-31	No impact	No Impact	Off (execution, ROM), On (RAM)	No impact	Off
Pass reusable subsystem outputs as on page 10-5	No impact	No impact	`Structure reference` (ROM), `Individual arguments` (execution, RAM)	No impact	`Individual Arguments`

Mapping of Application Requirements to the Code Generation Pane: Memory Sections Parameters

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
Package on page 17-3	No impact	No impact	No impact	No impact	`---None---`
Initialize/-Terminate on page 17-6	No impact	No impact	No impact	No impact	`Default`
Execution on page 17-7	No impact	No impact	No impact	No impact	`Default`
Shared utility on page 17-8	No impact	No impact	No impact	No impact	`Default`
Constants on page 17-9	No impact	No impact	No impact	No impact	`Default`
Inputs/Outputs on page 17-11	No impact	No impact	No impact	No impact	`Default`
Internal data on page 17-13	No impact	No impact	No impact	No impact	`Default`
Parameters on page 17-15	No impact	No impact	No impact	No impact	`Default`
Validation results on page 17-16	No impact	No impact	No impact	No impact	No package selected.

Mapping of Application Requirements to the Code Generation Pane: Report Tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
Code-to-model on page 12-5	On	On	No impact	No recommendation	Off
Model-to-code on page 12-7	On	On	No impact	No recommendation	Off
Generate model Web view on page 12-20	No impact	No impact	No impact	No impact	Off
Eliminated / virtual blocks on page 12-10	On	On	No impact	No recommendation	Off
Traceable Simulink blocks on page 12-12	On	On	No impact	No recommendation	Off
Traceable Stateflow objects on page 12-14	On	On	No impact	No recommendation	Off
Traceable MATLAB functions on page 12-16	On	On	No impact	No recommendation	Off
Generate static code metrics on page 12-4	No impact	No impact	No impact	No impact	Off
Summarize which blocks triggered code replacements on page 12-18	No impact	No impact	No impact	No impact	Off

Mapping of Application Requirements to the Code Generation Pane: Comments Tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
Simulink block descriptions on page 11-9	On	On	No impact	No impact	On
Simulink data object descriptions on page 11-11	On	On	No impact	No impact	On
Custom comments (MPT objects only) on page 11-13	On	On	No impact	No impact	Off
Custom comments function on page 11-15	Valid file name	Valid file name	No impact	No impact	''
Stateflow object descriptions on page 11-17	On	On	No impact	No impact	On
Requirements in block comments on page 11-19	On	On	No impact	No recommendation	Off

Mapping of Application Requirements to the Code Generation Pane: Identifiers Tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
Global variables on page 15-4	No impact	Use default	No impact	No recommendation	`$R$N$M`
Global types on page 15-6	No impact	Use default	No impact	No recommendation	`&N$R$M_T`
Field name of global types on page 15-8	No impact	Use default	No impact	No recommendation	`$N$M`
Subsystem methods on page 15-10	No impact	Use default	No impact	No recommendation	`$R$N$M$F`
Subsystem method arguments on page 15-12	No impact	Use default	No impact	No recommendation	`rt$I$N$M`
Local temporary variables on page 15-14	No impact	Use default	No impact	No recommendation	`$N$M`
Local block output variables on page 15-16	No impact	Use default	No impact	No recommendation	`rtb_$N$M`
Constant macros on page 15-18	No impact	Use default	No impact	No recommendation	`$R$N$M`
Shared utilities identifier format on page 15-20	No impact	Use default	No impact	No recommendation	`$N$C`
Minimum mangle length on page 15-22	No impact	1	No impact	No impact	1
Maximum identifier length	Valid value	>30	No impact	>30	31
System-generated identifiers on page 15-24	No impact	No impact	No impact	No impact	`Shortened`
Generate scalar inlined parameters as on page 15-29	No impact	`Macros`	`Literals`	No impact	`Literals`
Use the same reserved names as Simulation Target	No impact	No impact	No impact	No impact	Off

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
Shared checksum length on page 15-45	No impact	No impact	No impact	No impact	8
EMX array utility functions identifier format on page 15-46	No impact	No impact	No impact	No recommendation	`emx$M$N`
EMX array types identifier format on page 15-48	No impact	No impact	No impact	No recommendation	`emxArray_$M$N`
Custom token text on page 15-44	No impact	Set a custom string and use $U in symbols	No impact	No impact	`''`
#define naming on page 15-40	No impact	`Force uppercase`	No impact	No impact	`None`
Parameter naming on page 15-36	No impact	`Force uppercase`	No impact	No impact	`None`
Signal naming on page 15-32	No impact	`Force uppercase`	No impact	No impact	`None`
MATLAB function on page 15-34	No impact	No impact	No impact	No impact	`''`

Mapping of Application Requirements to the Code Generation Pane: Interface Tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
Support: floating-point numbers on page 14-8	No impact	No impact	Off (GUI), `'on'` (command-line) for integer only	No impact	On (GUI), `'off'` (command-line)
Support: complex numbers on page 14-10	No impact	No impact	Off for real only	No impact	On
Support: absolute time on page 14-11	No impact	No impact	Off	No recommendation	On
Support: continuous time on page 14-13	No impact	No impact	Off (execution, ROM), No impact (RAM)	No recommendation	Off
Support non-inlined S-functions on page 14-26	No impact	No impact	Off	No recommendation	Off
Support: variable-size signals on page 14-15	No impact	No impact	No impact	No recommendation	Off
Multiword type definitions on page 14-39	No impact	No impact	No impact	No recommendation	`System defined`
Maximum word length	No impact	No impact	No impact	No recommendation	256 for ERT targets 2048 for GRT targets
Pass root-level I/O as on page 14-16	No impact	No impact	No impact	No impact	`Individual arguments`
Use dynamic memory allocation for model initialization on page 14-28	No impact	No impact	No impact	No recommendation	Off
Terminate function required on page 14-32	No impact	No impact	No impact	No recommendation	On
Remove error status field in real-time model data structure on page 14-18	Off	No impact	On	No recommendation	Off

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
Include model types in model class on page 14-20	No impact	No impact	On	No recommendation	On
Combine signal/ state structures on page 14-33	Off	No impact	No impact	On	No impact
Generate destructor on page 14-41	No impact	No impact	No impact	No recommendation	On
Use dynamic memory allocation for model block instantiation on page 14-30	No impact	No impact	On	No recommendation	Off

Mapping of Application Requirements to the Code Generation Pane: Verification Tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
Measure task execution time on page 19-4	On	On	Off	No recommendation	Off
Measure function execution times on page 19-6	On	On	Off	No recommendation	Off
Workspace variable on page 19-8	No impact	Valid MATLAB variable name	No impact	No impact	Off
Save options on page 19-9	`All data`	`All data`	`Summary data only`	No impact	`Summary data only`
Third-party tool on page 19-11	`BullseyeCoverage` or `LDRA Testbed`	`BullseyeCoverage` or `LDRA Testbed`	`None` (code coverage off)	No recommendation	`None` (code coverage off)
Enable portable word sizes on page 19-13	On	On	Off	No impact	Off
Enable source-level debugging for SIL on page 19-15	On	On	Off	No impact	Off

Mapping of Application Requirements to the Code Generation Pane: Code Style Tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
Parentheses level on page 8-5	`Nominal (Optimize for readability)`	`Nominal (Optimize for readability)`	`Minimum (Rely on C/C++ operators for precedence)`	No recommendation	`Nominal (Optimize for readability)`
Preserve operand order in expression on page 8-7	On	On	Off	No recommendation	Off
Preserve condition expression in if statement on page 8-8	On	On	Off	No recommendation	Off
Convert if-elseif-else patterns to switch-case statements on page 8-10	No impact	Off	On (execution, ROM), No impact (RAM)	No impact	On
Preserve extern keyword in function declarations on page 8-12	No impact	No impact	No impact	No impact	On
Preserve static keyword in function declarations on page 8-14	No impact	No impact	On (execution, ROM)	No impact	On
Suppress generation of default cases for Stateflow switch statements if unreachable on page 8-16	On	On	Off	No recommendation	On
Replace multiplications by powers of two with signed bitwise shifts on page 8-18	No impact	No impact	On	No impact	On
Casting modes on page 8-22	`Nominal`	`Nominal`	`Nominal`	`Standards Compliant`	`Nominal`
Array container type on page 8-24	No impact	No impact	No impact	No recommendation	`C-style array`
Indent style on page 8-25	K&R	K&R	K&R	K&R	K&R
Indent size on page 8-27	2	2	2	2	2

Mapping of Application Requirements to the Code Generation Pane: Templates Tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
Code templates: Source file (*.c) template on page 18-4	No impact	No impact	No impact	No impact	`ert_code_-template.cgt`
Code templates: Header file (*.h) template on page 18-5	No impact	No impact	No impact	No impact	`ert_code_-template.cgt`
Data templates: Source file (*.c) template on page 18-6	No impact	No impact	No impact	No impact	`ert_code_-template.cgt`
Data templates: Header file (*.h) template on page 18-7	No impact	No impact	No impact	No impact	`ert_code_-template.cgt`
File customization template on page 18-8	No impact	No impact	No impact	No impact	`example_file_-process.tlc`
Generate an example main program on page 18-9	No impact	No impact	No impact	No impact	On
Target operating system on page 18-11	No impact	No impact	No impact	No impact	`BareBoard-Example`

Mapping of Application Requirements to the Code Generation Pane: Code Placement Tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
Data definition on page 7-4	No impact	Valid value	No impact	No impact	Auto
Data definition filename on page 7-6	No impact	Valid value	No impact	No impact	`global.c`
Data declaration on page 7-8	No impact	Valid value	No impact	No impact	Auto
Data declaration filename on page 7-10	No impact	Valid value	No impact	No impact	`global.h`
#include file delimiter on page 7-12	No impact	Valid value	No impact	No impact	`off`
#include file delimiter on page 7-13	No impact	Valid value	No impact	No impact	Auto
Signal display level on page 7-14	No impact	Valid integer	No impact	No impact	10
Parameter tune level on page 7-15	No impact	Valid integer	No impact	No impact	10
File packaging format on page 7-16	No impact	No impact	No impact	No impact	`Modular`

Mapping of Application Requirements to the Code Generation Pane: Data Type Replacement Tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
Replace data type names in the generated code on page 16-5	No impact	On	No impact	No impact	Off
Replacement Name on page 16-7	No impact	Valid character vector	No impact	No recommendation	`''`

See Also

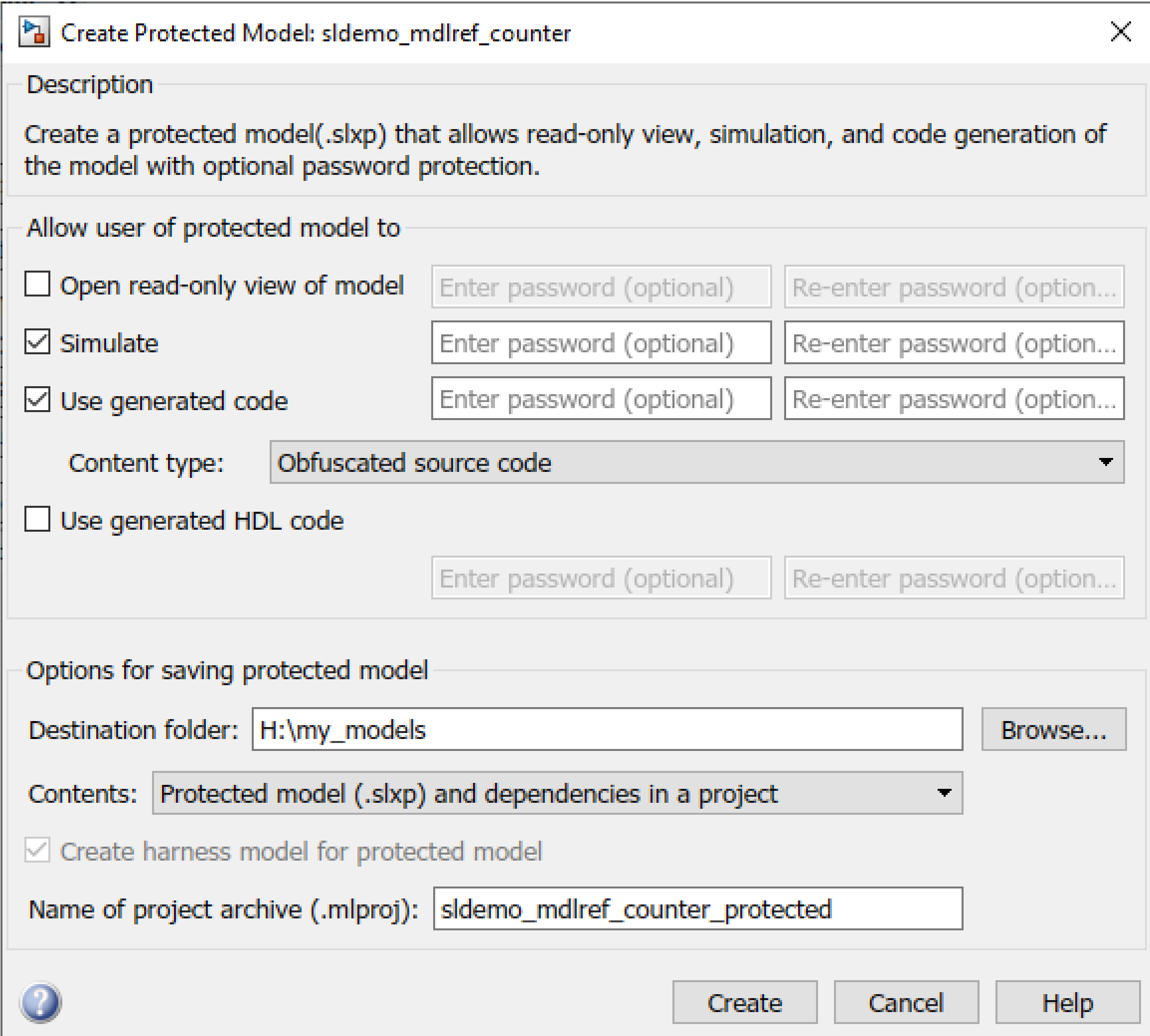
Related Examples

- “Configure Model for Code Generation Objectives by Using Code Generation Advisor”
- “Code Generation Advisor Checks”

Parameters for Creating Protected Models

Create Protected Model

This figure illustrates the various options in the Create Protected Model dialog box.



Create Protected Model: Overview

Create a protected model (.slxp) that allows read-only view, simulation, and code generation of the model with optional password protection.

To open the Create Protected Model dialog box, right-click the model block that references the model for which you want to generate protected model code. From the context menu, select **Subsystem & Model Reference > Create Protected Model for Selected Model Block**.

See Also

- “Reference Protected Models from Third Parties”
- “Protect Models to Conceal Contents”

Open read-only view of model

Share a view-only version of your protected model with optional password protection. View-only version includes the contents and block parameters of the model.

Settings

Default: Off

On

Share a Web view of the protected model. For password protection, create and verify a password with a minimum of four characters.

Off

Do not share a Web view of the protected model.

Alternatives

`Simulink.ModelReference.protect`

See Also

- “Protect Models to Conceal Contents”

Simulate

Enable user to simulate a protected model with optional password-protection. Selecting **Simulate**:

- Enables protected model Simulation Report.
- Sets Mode to Accelerator. You can run normal, accelerator, and rapid accelerator mode simulations.
- Displays only binaries and headers.
- Enables code obfuscation.

Settings

Default: On

On

User can simulate the protected model. For password protection, create and verify a password with a minimum of four characters.

Off

User cannot simulate the protected model.

Alternatives

`Simulink.ModelReference.protect`

See Also

- "Protect Models to Conceal Contents"

Use generated code

Allows user to generate code for the protected model with optional password protection. Selecting **Use generated code**:

- Enables Simulation Report and Code Generation Report for the protected model.
- Enables code generation.
- Enables support for simulation.

Settings

Default: Off

On

User can generate code for the protected model. For password protection, create and verify a password with a minimum of four characters.

Off

User cannot generate code for the protected model.

Dependencies

- To generate code, you must also select the **Simulate** check box.
- This parameter enables **Code interface** and **Content type**.

Alternatives

`Simulink.ModelReference.protect`

See Also

- "Code Generation Requirements and Limitations"
- "Protect Models to Conceal Contents"

Code interface

Specify the interface for the generated code.

Settings

Default: `Model reference`

`Model reference`

Specifies the model reference interface, which allows use of the protected model within a model reference hierarchy. Users of the protected model can generate code from a parent model that contains the protected model. In addition, users can run Model block software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations to verify code.

`Top model`

Specifies the standalone interface. Users of the protected model can run Model block SIL or PIL simulations to verify the protected model code.

Dependencies

- Requires an Embedded Coder license
- This parameter is enabled if you:
  - Specify an ERT (`ert.tlc`) system target file.
  - Select the **Use generated code** check box.

Alternatives

`Simulink.ModelReference.protect`

See Also

- "Code Generation Requirements and Limitations"
- "Protect Models to Conceal Contents"

Content type

Select the appearance of the generated code.

Settings

Default: `Obfuscated source code`

`Binaries`

Includes only compiled binaries for the generated code.

`Obfuscated source code`

Includes obfuscated source code.

`Readable source code`

Includes readable source code and readable code comments.

The options `Obfuscated source code` and `Readable source code` by default include only the minimal header files required to build the code with the chosen build settings. These options correspond to using the `Simulink.ModelReference.protect` with the `'OutputFormat'` option set to `'MinimalCode'`. To include header files found on the include path in the protected model, use the `Simulink.ModelReference.protect` function and set the `'OutputFormat'` option to `'AllReferencedHeaders'`.

The `Binaries` option corresponds to using the `Simulink.ModelReference.protect` function with the `'OutputFormat'` option set to `'CompiledBinaries'`.

Dependencies

This parameter is enabled by selecting the **Use generated code** check box.

Alternatives

`Simulink.ModelReference.protect`

See Also

- “Protect Models to Conceal Contents”

Use generated HDL code

Allows user to generate HDL code for the protected model with optional password protection. Selecting **Use generated HDL code**:

- Enables Simulation Report and HDL Code Generation Report for the protected model.
- Enables support for HDL code generation.
- Enables support for simulation.

Settings

Default: Off

On

User can generate HDL code for the protected model. For password protection, create and verify a password with a minimum of eight characters.

Off

User can simulate but cannot generate HDL code for the protected model.

Dependencies

To generate HDL code, you must also select the **Simulate** check box.

Alternatives

`Simulink.ModelReference.protect`

See Also

- “Code Generation Requirements and Limitations”
- “Protect Models to Conceal Contents”

Destination folder

Specify the path of the folder to contain the protected model.

Settings

Default: Current working folder

Dependencies

A model that you protect must be available on the MATLAB path.

Alternatives

`Simulink.ModelReference.protect`

See Also

- "Protect Models to Conceal Contents"

Contents

Option to package supporting files, including a harness model, with the protected model in a project archive. The type and number of supporting files depends on the model being protected. Examples of supporting files are a MAT-file with base workspace definitions and a data dictionary pruned to relevant definitions. The supporting files are not protected.

Note Before sharing the project, check whether the project contains the necessary supporting files. If supporting files are missing, simulating or generating code for the related harness model can help identify them. Add the missing dependencies to the project and update the harness model as needed.

Settings

Default: `Protected model (.slxp) and dependencies in a project`

`Protected model (.slxp) and dependencies in a project`

Create a project archive that contains the protected model, its dependencies, and its harness model. The supporting files are not protected. The project archive is a single file that allows for easy sharing.

`Protected model (.slxp) only`

Create only the protected model. If the protected model has dependencies, you must share them separately. Similarly, if you create a harness model for the protected model, you must share the harness model separately.

Alternatives

`Simulink.ModelReference.protect`

See Also

- "Protect Models to Conceal Contents"

Create harness model for protected model

Create a harness model for the protected model. The harness model provides an isolated environment for the protected model, which is referenced by a Model block.

Settings

Default: Off

On

Create a harness model for the protected model.

☐ Off

Do not create a harness model for the protected model.

Dependencies

To clear the check box for this parameter, set **Contents** to `Protected model (.slxp) only`.

Alternatives

`Simulink.ModelReference.protect`

See Also

- “Protect Models to Conceal Contents”

Name of project archive (.mlproj)

Name of the project archive that contains the generated files. The project inside the archive uses the same name.

Settings

Default: *`modelname`*`_protected`

Dependencies

To enable this parameter, set **Contents** to `Protected model (.slxp) and dependencies in a project`.

Alternatives

`Simulink.ModelReference.protect`

See Also

- “Protect Models to Conceal Contents”

Model Advisor Checks

Embedded Coder Checks

In this section...

“Embedded Coder Checks Overview” on page 22-2
“Check for blocks not recommended for C/C++ production code deployment” on page 22-3
“Check configuration parameters for generation of inefficient saturation code” on page 22-4
“Identify lookup table blocks that generate expensive out-of-range checking code” on page 22-5
“Check output types of logic blocks” on page 22-6
“Check the hardware implementation” on page 22-7
“Identify questionable software environment specifications” on page 22-8
“Identify questionable code instrumentation (data I/O)” on page 22-9
“Identify blocks generating inefficient algorithms” on page 22-10
“Check configuration parameters for MISRA C:2012” on page 22-11
“Check for blocks not recommended for MISRA C:2012” on page 22-13
“Check for unsupported block names” on page 22-15
“Check usage of Assignment blocks” on page 22-15
“Check for switch case expressions without a default case” on page 22-17
“Check for missing error ports for AUTOSAR receiver interfaces” on page 22-18
“Check bus object names that are used as bus element names” on page 22-19
“Check configuration parameters for secure coding standards” on page 22-19
“Check for blocks not recommended for secure coding standards” on page 22-21
“Identify questionable subsystem settings” on page 22-22
“Check for blocks not supported for row-major code generation” on page 22-23
“Identify TLC S-Functions with unset array layout” on page 22-24
“Identify blocks that generate expensive fixed-point and saturation code” on page 22-24
“Check for missing const qualifiers in model functions” on page 22-26
“Identify questionable fixed-point operations” on page 22-27
“Identify blocks that generate expensive rounding code” on page 22-29
“Check for bitwise operations on signed integers” on page 22-29
“Check for recursive function calls” on page 22-30
“Check for equality and inequality operations on floating-point values” on page 22-30
“Check integer word length” on page 22-31
“Check block names” on page 22-32

Embedded Coder Checks Overview

Use Embedded Coder Model Advisor checks to configure your model for code generation.

See Also

- “Run Model Advisor Checks”

- “Simulink Checks”
- “Simulink Coder Checks”

Check for blocks not recommended for C/C++ production code deployment

Check ID: `mathworks.codegen.PCGSupport`

Identify blocks not supported by code generation or not recommended for C/C++ production code deployment.

Description

This check partially identifies model constructs that are not recommended for C/C++ production code generation. For Simulink Coder and Embedded Coder, these model construct identities appear in tables of Simulink Block Support.

In some instances, this check flags blocks that are supported for code generation. For these blocks, you should review the footnote information that is provided in the support notes and adhere to the recommended action provided by the Model Advisor.

Following the recommendations of this check increases the likelihood of generating code that complies with the CERT C, CWE, and ISO/IEC TS 17961 standards.

Available with Embedded Coder and Simulink Check™.

Results and Recommended Actions

Condition	Recommended Action
The model or subsystem contains blocks that should not be used for production code deployment.	Consider replacing the blocks listed in the results. Click an element from the list of questionable items to locate condition.
The model or subsystem contains blocks that are supported but not recommended for production code generation.	Review the support notes and adhere to the recommended action provided by the Model Advisor.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Analyze content of library linked blocks.
- Analyze content in masked subsystems.
- Exclude blocks and charts if you have a Simulink Check license.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- “Blocks and Products Supported for Code Generation”
- “Model Advisor Exclusion Overview” (Simulink Check)
- Secure Coding Standards

Check configuration parameters for generation of inefficient saturation code

Check ID: `mathworks.codegen.EfficientTunableParamExpr`

Check model configuration for optimization settings that can improve code efficiency.

Description

This check identifies model configuration parameters that are recommended for C/C++ production code generation. For Embedded Coder, these model configuration parameters improve code efficiency.

Available with Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
The optimization suppresses code generation to guard against integer overflow for tunable parameter expression that you select. Select saturation code elimination.	If you have Embedded Coder and are using an ERT-based system target file, select Configuration Parameter “Remove Code from Tunable Parameter Expressions That Saturate Out-of-Range Values” on page 10-45 or set the parameter `EfficientTunableParamExpr` to `on`.
The optimization suppresses code generation that handles floating-point to integer conversion results for `NaN` values. Select conversion code elimination.	If you have Embedded Coder and are using an ERT-based system target file, select Configuration Parameter “Remove code from floating-point to integer conversions with saturation that maps NaN to zero” or set the parameter `EfficientMapNaN2IntZero` to `on`.

Action Results

Clicking **Modify Settings** configures model optimization settings can impact the efficiency of code generation. There are no safety concerns for:

- The **Remove code from tunable parameter expressions that saturates out-of-range values** parameter if your simulation contains the entire range of values for the parameters that are the terms of tunable expressions and Simulink does not generate a saturation warning.
- The **Remove code from floating-point to integer conversions with saturation that maps NaN to zero** parameter if your model simulation does not contain a NaN input value.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks or charts.

See Also

- "Remove Code from Tunable Parameter Expressions That Saturate Out-of-Range Values" on page 10-45
- "Remove Code from Tunable Parameter Expressions That Saturate Against Integer Overflow"
- "Remove code from floating-point to integer conversions with saturation that maps NaN to zero"
- "Remove Code That Maps NaN to Integer Zero"

Identify lookup table blocks that generate expensive out-of-range checking code

Check ID: `mathworks.codegen.LUTRangeCheckCode`

Identify lookup table blocks that generate code to protect against out-of-range inputs for breakpoint or index values.

Description

This check verifies that the following blocks do not generate code to protect against inputs that fall outside the range of valid breakpoint values:

- 1-D Lookup Table
- 2-D Lookup Table
- n-D Lookup Table
- Prelookup

This check also verifies that Interpolation Using Prelookup blocks do not generate code to protect against inputs that fall outside the range of valid index values.

Following the recommended actions increases both execution and ROM efficiency of the generated code.

Available with Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
The lookup table block generates out-of-range checking code.	Change the setting on the block dialog box so that out-of-range checking code is not generated. • For the 1-D Lookup Table, 2-D Lookup Table, n-D Lookup Table, and Prelookup blocks, select the check box for **Remove protection against out-of-range input in generated code**. • For the Interpolation Using Prelookup block, select the check box for **Remove protection against out-of-range index in generated code**.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Check license.

Action Results

Clicking **Modify** prevents lookup table blocks from generating out-of-range checking code, which makes the generated code more efficient.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- n-D Lookup Table
- Prelookup
- Interpolation Using Prelookup
- “Optimize Generated Code for Lookup Table Blocks”
- “Model Advisor Exclusion Overview” (Simulink Check)

Check output types of logic blocks

Check ID: `mathworks.codegen.LogicBlockUseNonBooleanOutput`

Identify logic blocks that do not use `boolean` for the output data type.

Description

This check verifies that the output data type of the following blocks is `boolean`:

- Compare To Constant
- Compare To Zero
- Detect Change
- Detect Decrease
- Detect Fall Negative
- Detect Fall Nonpositive
- Detect Increase
- Detect Rise Nonnegative
- Detect Rise Positive
- Interval Test
- Interval Test Dynamic
- Logical Operator
- Relational Operator

Using output data type `boolean` increases execution efficiency of the generated code.

Available with Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
The output data type of a logic block is not `boolean`.	In the block dialog box, set **Output data type** to `boolean`.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Check license.

See Also

- “Model Advisor Exclusion Overview” (Simulink Check)

Action Results

Clicking **Modify** forces logic blocks to use `boolean` as the output data type. If a logic block uses `uint8` for the output type, clicking **Modify** changes the output type to `boolean`.

Check the hardware implementation

Check ID: `mathworks.codegen.HWImplementation`

Identify inconsistent or underspecified hardware implementation settings

Description

The Simulink and Simulink Coder software require two sets of target specifications. The first set describes the final intended production target. The second set describes the currently selected target. If the configurations do not match, the code generator creates extra code to emulate the behavior of the production target. Inconsistencies or underspecification of hardware attributes can lead to inefficient or incorrect code generation for the target hardware.

Available with Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
Hardware implementation parameters are not set to recommended values.	In the Configuration Parameters dialog box, on the Hardware Implementation pane, specify the following parameters: • **Byte ordering** (`ProdEndianess`) • **Production hardware signed integer division rounds to** (`ProdIntDivRoundTo`) In the Configuration Parameters dialog box, specify the following parameters: • **Byte ordering in test hardware** (`TargetEndianess`) • **Test hardware signed integer division rounds to** (`TargetIntDivRoundTo`)
Hardware implementation **Production hardware** settings do not match **Test hardware** settings.	In the Configuration Parameters dialog box, consider selecting the **Test hardware is the same as production hardware** (`ProdEqTarget`) check box, or modify the settings to match.

See Also

"Run-Time Environment Configuration"

Identify questionable software environment specifications

Check ID: `mathworks.codegen.SWEnvironmentSpec`

Identify questionable software environment settings.

Description

- Support for some software environment settings can lead to inefficient code generation and nonoptimal results.
- Industry standards for C, such as ISO and MISRA, require identifiers to be unique within the first 31 characters.
- Stateflow charts with weak Simulink I/O data types lead to inefficient code.

Available with Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
The maximum identifier length does not conform with industry standards for C.	In the Configuration Parameters dialog box, on the **Code Generation > Identifiers** pane, set the **Maximum identifier length** parameter to `31` characters.
In the Configuration Parameters dialog box, the parameters on the **Code Generation > Interface** pane are not set to recommended values.	In the Configuration Parameters dialog box, on the **Code Generation > Interface** pane, clear the following parameters: • **Support: continuous time** • **Support: non-finite numbers** In the Configuration Parameters dialog box, clear **Support non-inlined S-functions**.
In the Configuration Parameters dialog box, the parameters on the **Code Generation > Identifiers** pane are not set to recommended values.	In the Configuration Parameters dialog box, on the **Code Generation > Identifiers** pane, set the **Generate scalar inlined parameters as** on page 15-29 parameter to `Literals`.
In the Configuration Parameters dialog box, on the **Code Generation > Interface** pane, **Support: variable-size signals** is selected. This might lead to inefficient code.	If you do not intend to support variable-sized signals, in the Configuration Parameters dialog box, on the **Code Generation > Interface** pane, clear **Support: variable-size signals** on page 14-15.
The model contains Stateflow charts with weak Simulink I/O data type specifications.	Select the Stateflow chart property **Use Strong Data Typing with Simulink I/O**. You might need to adjust the data types in your model after selecting the property.

Limitations

A Stateflow license is required when using Stateflow charts.

See Also

"Strong Data Typing with Simulink Inputs and Outputs" (Stateflow)

Identify questionable code instrumentation (data I/O)

Check ID: `mathworks.codegen.CodeInstrumentation`

Identify questionable code instrumentation.

Description

- Instrumentation of the generated code can cause nonoptimal results.
- Test points require global memory and are not optimal for production code generation.

Available with Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
Interface parameters are not set to recommended values.	In the Configuration Parameters dialog box, on the **Code Generation > Interface** pane, set the parameters to the recommended values.
Blocks generate assertion code.	In the Configuration Parameters dialog box, set **Model Verification block enabling** to `Disable All` on a block-by-block basis or globally.
Block output signals have one or more test points and, if you have an Embedded Coder license, the **Ignore test point signals** check box is cleared in the Configuration Parameters dialog box.	Remove test points from the specified block output signals. For each signal, in the Signal Properties dialog box, clear the **Test point** check box. Alternatively, if the model is using an ERT-based system target file, select the **Ignore test point signals** check box in the Configuration Parameters dialog box to ignore test points during code generation.

Capabilities and Limitations

If you have a Simulink Check license, you can exclude blocks and charts from this check.

See Also

- “Model Advisor Exclusion Overview” (Simulink Check)

Identify blocks generating inefficient algorithms

Check ID: `mathworks.codegen.UseRowMajorAlgorithm`

Identify blocks generating inefficient algorithms.

Description

This check identifies the blocks that generate inefficient algorithms in the generated code based on the array layout of the model.

Available with Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
The configuration parameter **Array layout** is set to `Column-major` for column-major code generation.	Disable the configuration parameter **Use algorithms optimized for row-major array layout**.
The configuration parameter **Array layout** is set to `Row-major` for row-major code generation.	Select the configuration parameter **Use algorithms optimized for row-major array layout**.

Capabilities and Limitations

- Analyzes content in masked subsystems.

See Also

- “Code Generation of Matrices and Arrays”

Check configuration parameters for MISRA C:2012

Check ID: `mathworks.misra.CodeGenSettings`

Identify configuration parameters that can impact MISRA C:2012 compliant code generation.

Description

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Math and Data Types	
Configuration parameter **Use division for fixed-point net slope computation** is not set to `On` or `Use division for reciprocals of integers only`.	Set **Use division for fixed-point net slope computation** to `On` or `Use division for reciprocals of integers only`.
Configuration parameter **Inf or NaN block output** is set to `None` or `error` and **Support non-finite numbers** is set to `on`. Configuration parameter **Inf or NaN block output** is set to `None` and **Support non-finite numbers** is set to `off`.	When **Support non-finite numbers** is: • `on`, set **Inf or NaN block output** to `warning` • `off`, set **Inf or NaN block output** to `warning` or `error`
Configuration parameter **Model Verification block enabling** is set to `Use local settings` or `Enable All`.	Set **Model Verification block enabling** to `Disable All`.
Configuration parameter **Undirected event broadcasts** is set to `none` or `warning`.	Set **Undirected event broadcasts** to `error`.
Configuration parameter **Wrap on overflow** is set to `None`	Set configuration parameter **Wrap on overflow** to `warning` or `error`.
Hardware Implementation	
Configuration parameter **Production hardware signed integer division rounds to** is set to `Undefined`	Set **Production hardware signed integer division rounds to** to `Zero` or `Floor`.
Configuration parameter **Shift right on a signed integer as arithmetic shift** is selected.	Clear **Shift right on a signed integer as arithmetic shift**.

Condition	Recommended Action
Simulation Target	
Configuration parameter **Compile-time recursion limit for MATLAB functions** is set to a value other than 0.	Set **Compile-time recursion limit for MATLAB functions** to 0.
Configuration parameter **Dynamic memory allocation in MATLAB functions** is selected.	Clear **Dynamic memory allocation in MATLAB functions**.
Configuration parameter **Enable run-time recursion for MATLAB functions** is selected.	Clear **Enable run-time recursion for MATLAB functions**.
Code Generation	
Configuration parameter **Bitfield declarator type specifier** is set to uchar_T when any of these parameters are selected: • **Pack Boolean data into bitfields** • **Use bitsets for storing state configuration** • **Use bitsets for storing Boolean data**	Set **Bitfield declarator type specifier** to uint_T.
Configuration parameter **Casting Modes** is not set to Standards Compliant.	Set **Casting Modes** to Standards Compliant.
Configuration parameter **Code replacement library** is not set to None or AUTOSAR 4.0.	Set **Code replacement library** to None or AUTOSAR 4.0
Configuration parameter **External mode** is selected.	Clear **External mode**.
Configuration parameter **Generate shared constants** is selected.	Clear **Generate shared constants**.
Configuration parameter **Include comments** is cleared.	Select **Include comments**.
Configuration parameter **MAT-file logging** is selected.	Clear **MAT-file logging**
For ERT-based target systems, configuration parameter **MATLAB user comments** is cleared.	Select **MATLAB user comments**.
A value for configuration parameter **Maximum identifier length** is not provided.	Set the value to the implementation-dependent limit. The default is 31.
Configuration parameter **Parenthesis level** is not set to Maximum (Specify precedence with parentheses).	Set **Parentheses level** to Maximum (Specify precedence with parentheses).
For ERT-based target systems, configuration parameter **Preserve static keyword in function declarations** is cleared when **File packaging format** is set to Compact or Compact (with separate data file)	Select **Preserve static keyword in function declarations**.
Configuration parameter **Replace multiplications by powers of two with signed bitwise shifts** is selected.	Clear **Replace multiplications by powers of two with signed bitwise shifts**.

Condition	Recommended Action
Configuration parameter **Shared code placement** is set to `Auto`.	Set **Shared code placement** to `Shared location`
For ERT-based target systems, configuration parameter **Support continuous time** is selected	Clear **Support continuous time**.
For ERT-based target systems, configuration parameter **Support non-inlined S-functions** is selected	Clear **Support non-inlined S-functions**.
Configuration parameter **System-generated identifiers** is set to `Classic`.	Set **System-generated identifiers** to `Shortened`.
Configuration parameter **System target file** is set to a GRT-based target.	Set **System target file** to an ERT-based target.
Configuration parameter **Use dynamic memory allocation for model initialization** is selected when **Code Interface Packaging** is set to `Reusable Function`.	Clear **Use dynamic memory allocation for model initialization**. **Note** Select only when **Code Interface Packaging** is set to `Reusable Function`.

Action Results

Clicking **Modify All** changes the parameter values to the recommended values.

Note When you click **Modify All** for models with a GRT-based target, the Model Advisor does not update the **System target file** configuration parameter to an ERT-based system.

Parameter subchecks depend on the results of the parameter noted with **D** in the results table. When the result is *D-Warning*, the **Current Value** column in the results table states *Prerequisite constraint not met* for the subchecks. After you change the parameter, rerun the check.

Note Some subchecks are specific to configuration parameters for ERT-based systems. These parameters are not updated when you click **Modify All** unless you change the model to an ERT-based system.

Capabilities and Limitations

This check does not review referenced models.

See Also

- hisl_0060: Configuration parameters that improve MISRA C:2012 compliance
- “MISRA C Guidelines”
- “MISRA C:2012 Compliance Considerations”

Check for blocks not recommended for MISRA C:2012

Check ID: `mathworks.misra.BlkSupport`

Identify blocks that are not supported or recommended for MISRA C:2012 compliant code generation.

Description

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Lookup Table blocks using cubic spline interpolation or extrapolation methods were found in the model or subsystem. Specific blocks are: • 1-D Lookup Table • 2-D Lookup Table • n-D Lookup Table	Consider other interpolation and extrapolation methods for the Lookup Table blocks.
Deprecated Lookup Table blocks were found in the model or subsystem. Specific blocks are: • Lookup Table • Lookup Table (2-D)	Consider replacing the deprecated Lookup Table blocks.
S-Function Builder blocks were found in the model or subsystem.	Consider replacing the S-Function Builder blocks with blocks recommended for production.
From Workspace blocks were found in the model or subsystem	Consider replacing the From Workspace blocks with blocks recommended for production.
String blocks were found in the model or subsystem. Specific blocks are: • Compose String • Scan String • String to Single • String to Double • To String	Consider replacing the String blocks with blocks recommended for production.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Analyzes content of library linked blocks.
- Analyzes content in masked subsystems.
- Exclude blocks and charts from this check if you have a Simulink Check license.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- “hisl_0020: Blocks not recommended for MISRA C:2012 compliance”
- “MISRA C Guidelines”
- “MISRA C:2012 Compliance Considerations”
- “Model Advisor Exclusion Overview” (Simulink Check)

Check for unsupported block names

Check ID: `mathworks.misra.BlockNames`

Identify block names containing `/`.

Description

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Block names containing `/` were found in the model or subsystem.	Remove `/` from the block name.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library linked blocks.
- Analyzes content in masked subsystems.
- If you have a Simulink Check license, allows exclusions of blocks and charts.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- MISRA C:2012, Rule 3.1
- “MISRA C Guidelines”
- “MISRA C:2012 Compliance Considerations”

Check usage of Assignment blocks

Check ID: `mathworks.misra.AssignmentBlocks`

Identify Assignment blocks that do not have block parameter **Action if any output element is not assigned** set to **Error** or **Warning**.

Description

This check applies to the Assignment block that is available in the Simulink block library under **Simulink > Math Operations**.

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications, as well as code that complies with the CERT C, CWE, ISO/IEC TS 17961 standards.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The model or subsystem might contain Assignment blocks with incomplete array initialization that do not have block parameter **Action if any output element is not assigned** set to **Error** or **Warning**.	Set block parameter **Action if any output element is not assigned** to one of the recommended values: • **Error**, if Assignment block is not in an Iterator subsystem. • **Warning**, if Assignment block is in an Iterator subsystem.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library linked blocks.
- Analyzes content in masked subsystems.
- If you have a Simulink Check license, allows exclusions of blocks and charts.

Edit-Time Checking

This check is supported by edit-time checking. However, the following check condition is not supported because edit-time checking is unable to determine whether the Assignment block is in an Iterator subsystem.

Set block parameter **Action if any output element is not assigned** to one of the recommended values:

- **Error**, if Assignment block is not in an Iterator subsystem.
- **Warning**, if Assignment block is in an Iterator subsystem.

See Also

- MISRA C:2012, Rule 9.1
- ISO/IEC TS 17961: 2013, uninitref
- CERT C, EXP33-C
- CWE, CWE-908

- “hisl_0029: Usage of Assignment blocks”
- “MISRA C Guidelines”
- “MISRA C:2012 Compliance Considerations”
- “Secure Coding Standards”

Check for switch case expressions without a default case

Check ID: `mathworks.misra.SwitchDefault`

Identify switch case expressions that do not have a default case.

Description

The check flags model objects that have switch case expressions without a default case.

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications, as well as code that complies with the CERT C, CWE, ISO/IEC TS 17961 standards.

The check does not flag blocks without default cases if they are justified with a Polyspace annotation. When you run the check, the **Blocks with justification** table lists blocks without default cases that have a justification.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Model object has a switch case expression without a default case.	For Switch Case blocks, consider selecting block parameter **Show default case** to explicitly specify a default case.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Check license.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- MISRA C:2012, Rule 16.4
- ISO/IEC TS 17961: 2013, swtchdflt
- CERT C, MSC01-C
- CWE, CWE-478
- “Annotate Code and Hide Known or Acceptable Results” (Polyspace Bug Finder)

- “Secure Coding Standards”

Check for missing error ports for AUTOSAR receiver interfaces

Check ID: `mathworks.misra.AutosarReceiverInterface`

Identify AUTOSAR receiver interface inports that do not have matching error ports.

Description

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications. The check flags AUTOSAR receiver interfaces inports that are missing error ports. The following table identifies the AUTOSAR data access mode types for receiver interface ports that are flagged by the check when the corresponding error port is missing.

AUTOSAR Data Access Mode Type	Flagged by Check?
`ImplicitReceive`	Yes
`ExplicitReceive`	Yes
`QueuedExplicitReceive`	No
`ErrorStatus`	No
`ModeReceive`	No
`IsUpdated`	No
`EndToEndRead`	Yes
`ExplicitReceiveByVal`	No
`otherwise`	No

The check does not flag missing error ports when they are justified with a Polyspace annotation. When you run the check, the **Blocks with justification** table lists the missing error ports that have a justification.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
AUTOSAR receiver interface inport does not have a matching error port.	Add missing error port and map to the corresponding AUTOSAR receiver interface inport.
AUTOSAR receiver interface ports do not have a matching error port when data access mode is `ImplicitReceive`, `ExplicitReceive`, or `EndToEndRead`.	Add missing error port and map to the corresponding AUTOSAR receiver interface inport.

Capabilities and Limitations

You can:

- Analyzes top layer/root level models.

- Exclude blocks and charts from this check if you have a Simulink Check license.

See Also

- MISRA C: 2012, Directive 4.7
- “MISRA C Guidelines”
- “Model Advisor Exclusion Overview” (Simulink Check)
- “Annotate Code and Hide Known or Acceptable Results” (Polyspace Bug Finder)
- “Configure AUTOSAR Elements and Properties” (AUTOSAR Blockset)
- “AUTOSAR Component Configuration” (AUTOSAR Blockset)

Check bus object names that are used as bus element names

Check ID: `mathworks.misra.BusElementNames`

Identify bus object names that are used as bus element names.

Description

Using this check increases the likelihood of generating code for embedded applications that is compliant with MISRA C:2012. The check flags instances where a Simulink.Bus object name is used as the Simulink.Bus element name.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
A bus object name is being used as a bus element name.	Change either the flagged bus object name or the bus element name so that they are not identical.

See Also

- MISRA C:2012, Rule 5.6
- MISRA AC AGC, Rule 5.3
- “MISRA C Guidelines”

Check configuration parameters for secure coding standards

Check ID: `mathworks.security.CodeGenSettings`

Identify configuration parameters that might impact compliance with secure coding standards.

Description

Following the recommendations of this check increases the likelihood of generating code that complies with CERT C, CWE, ISO/IEC TS 17961 secure coding standards.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Diagnostics	
Configuration parameter **Inf or NaN block output** is set to `None` or `error` and **Support non-finite numbers** is set to `on`. Configuration parameter **Inf or NaN block output** is set to `None` and **Support non-finite numbers** is set to `off`.	When **Support non-finite numbers** is: • `on`, set **Inf or NaN block output** to `warning`. • `off`, set **Inf or NaN block output** to `warning` or `error`.
Configuration parameter **Model Verification block enabling** is set to `Use local settings` or `Enable All`.	Set **Model Verification block enabling** to `Disable All`.
Configuration parameter **Undirected event broadcasts** is set to `none` or `warning`.	Set **Undirected event broadcasts** to `error`.
Configuration parameter **Wrap on overflow** is set to `none`.	Set **Wrap on overflow** to `warning` or `error`.
Hardware Implementation	
Configuration parameter **Production hardware signed integer division rounds to** is set to `Undefined`.	Set **Production hardware signed integer division rounds to** to `Zero` or `Floor`.
Configuration parameter **Shift right on a signed integer as arithmetic shift** is selected.	Clear **Shift right on a signed integer as arithmetic shift**.
Simulation Target	
Configuration parameter **Compile-time recursion limit for MATLAB functions** is set to a value other than `0`.	Set **Compile-time recursion limit for MATLAB functions** to `0`.
Configuration parameter **Dynamic memory allocation in MATLAB functions** is selected.	Clear **Dynamic memory allocation in MATLAB functions**.
Configuration parameter **Enable run-time recursion for MATLAB functions** is selected.	Clear **Enable run-time recursion for MATLAB functions**.
Code Generation	
Configuration parameter **Code replacement library** is not set to `None` or `AUTOSAR 4.0`.	Set **Code replacement library** to `None` or `AUTOSAR 4.0`.
Configuration parameter **External mode** is selected.	Clear **External mode**.
Configuration parameter **Include comments** is cleared.	Select **Include comments**.
Configuration parameter **MAT-file logging** is selected.	Clear **MAT-file logging**.
For ERT-based target systems, configuration parameter **MATLAB user comments** is cleared.	Select **MATLAB user comments**.

Condition	Recommended Action
Configuration parameter **Replace multiplications by powers of two with signed bitwise shifts** is selected.	Clear **Replace multiplications by powers of two with signed bitwise shifts**.
For ERT-based target systems, configuration parameter **Support continuous time** is selected	Clear **Support continuous time**.
For ERT-based target systems, configuration parameter **Support non-inlined S-functions** is selected	Clear **Support non-inlined S-functions**.
Configuration parameter **System target file** is set to a GRT-based target.	Set **System target file** to an ERT-based target.
Configuration parameter **Use dynamic memory allocation for model initialization** is selected.	Clear **Use dynamic memory allocation for model initialization**. **Note** Select only when **Code Interface Packaging** is set to `Reusable Function`.

Action Results

Clicking **Modify All** changes the parameter values to the recommended values.

Note When you click **Modify All** for models with a GRT-based target, the Model Advisor does not update the **System target file** configuration parameter to an ERT-based system.

Parameter subchecks depend on the results of the parameter noted with **D** in the results table. When the result is *D-Warning*, the **Current Value** column in the results table states *Prerequisite constraint not met* for the subchecks. After you change the parameter, rerun the check.

Note Some subchecks are specific to configuration parameters for ERT-based systems. These parameters are not updated when you click **Modify All** unless you change the model to an ERT-based system.

See Also

"Secure Coding Standards"

Check for blocks not recommended for secure coding standards

Check ID: `mathworks.security.BlockSupport`

Identify blocks not recommended for compliance with secure coding standards.

Description

Following the recommendations of this check increases the likelihood of generating code that complies with CERT C, CWE, ISO/IEC TS 17961 secure coding standards.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Lookup Table blocks using cubic spline interpolation or extrapolation methods were found in the model or subsystem. Specific blocks are: • 1-D Lookup Table • 2-D Lookup Table • n-D Lookup Table	Consider other interpolation and extrapolation methods for the Lookup Table blocks.
Deprecated Lookup Table blocks were found in the model or subsystem. Specific blocks are: • Lookup Table • Lookup Table (2-D)	Consider replacing the deprecated Lookup Table blocks.
S-Function Builder blocks were found in the model or subsystem.	Consider replacing the S-Function Builder blocks with blocks recommended for production.
From Workspace blocks were found in the model or subsystem	Consider replacing the From Workspace blocks with blocks recommended for production.
String blocks were found in the model or subsystem. Specific blocks are: • Compose String • Scan String • String to Single • String to Double • To String	Consider replacing the String blocks with blocks recommended for production.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Check license.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- “Model Advisor Exclusion Overview” (Simulink Check)
- “Secure Coding Standards”

Identify questionable subsystem settings

Check ID: `mathworks.codegen.QuestionableSubsysSetting`

Identify questionable subsystem block settings.

Description

Subsystem blocks implemented as void-void functions in the generated code use global memory to store the subsystem I/O.

Available with Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
Subsystem blocks have the **Subsystem Parameters** > **Function packaging** option set to `Nonreusable function`.	Set the **Subsystem Parameters** > **Function packaging** parameter to `Auto`.
Subsystem blocks have the **Subsystem Parameters** > **Function packaging** option set to `Reusable function`.	Set the **Subsystem Parameters** > **Function packaging** parameter to `Auto`.

Capabilities and Limitations

If you have a Simulink Check license, you can exclude blocks and charts from this check.

See Also

- Subsystem block
- “Model Advisor Exclusion Overview” (Simulink Check)

Check for blocks not supported for row-major code generation

Check ID: `mathworks.codegen.RowMajorCodeGenSupport`

Check for blocks not supported for row-major code generation.

Description

This check identifies the blocks that are not supported for row-major code generation.

Available with Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
The model interfaces with external data that is in row-major array layout.	Set the configuration parameter **Array layout** to `Row-major`.

Capabilities and Limitations

- Analyzes content in masked subsystems.

See Also

- “Code Generation of Matrices and Arrays”

Identify TLC S-Functions with unset array layout

Check ID: `mathworks.codegen.RowMajorUnsetSFunction`

Identify TLC S-Functions with unset array layout.

Description

This check identifies S-functions that have `SSArrayLayout` set to `SS_UNSET`. By default, every S-function has `SSArrayLayout` property set to `SS_UNSET`. This setting disables the S-function for row-major code generation. When the configuration parameter **Array layout** is set to `Row-major`, the Embedded Coder reports an error. You can turn off the error by changing the **External functions compatibility for row-major code generation** to `warning` or `none`.

Available with Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
The configuration parameter **Array layout** is set to `Column-major` for column-major code generation.	Set the `SSArrayLayout` property to `Column-major`.
The configuration parameter **Array layout** is set to `Row-major` for row-major code generation.	Set the `SSArrayLayout` property to `Row-major`.

Capabilities and Limitations

- Analyzes content in all masked subsystems.

See Also

- “Code Generation of Matrices and Arrays”

Identify blocks that generate expensive fixed-point and saturation code

Check ID: `mathworks.codegen.BlockSpecificQuestionableFxptOperations`

Identify fixed-point operations that can lead to nonoptimal results.

Description

Certain block settings can lead to expensive fixed-point and saturation code.

Results and Recommended Actions

Conditions	Recommended Action
Blocks generate expensive saturation code.	Check whether your application requires setting **Function Block Parameters > Signal Attributes > Saturate on integer overflow**. Otherwise, clear the **Saturate on integer overflow** parameter for the most efficient implementation of the block in the generated code.
Product blocks are multiplying signals with mismatched slope adjustment factors. The net slope computation uses multiplication followed by shifts, which is inefficient for some target hardware.	Set the **Optimization > Use division for fixed-point net slope computation** parameter to `On`, or `Use division for reciprocals of integers only` if the net slope can be approximated by a fraction and division is more efficient than multiplication and shifts on the target hardware. **Note** This optimization takes place only if certain simplicity and accuracy conditions are met. For more information, see "Handle Net Slope Computation" (Fixed-Point Designer).
Product blocks are configured with a divide operation for the first input and a multiply operation for the second input.	Reverse the inputs so the multiply operation occurs first and the division operation occurs second.
Product blocks are configured to do multiple division operations.	Multiply all the denominator terms together, and then do a single division using cascading Product blocks.
Product blocks are configured to do many multiplication or division operations.	Split the operations across several blocks, with each block performing one multiplication or one division operation.
Protection code generated as part of the division operation is redundant.	Verify that your model cannot cause exceptions in division operations and then remove redundant protection code by setting the **Optimization > Remove code that protects against division arithmetic exceptions** on page 10-47 parameter in the Configuration Parameters dialog box.
The data type range of the inputs of Sum blocks exceeds the data type range of the output, which can cause overflow or saturation.	Change the output and accumulator data types so the range equals or exceeds all input ranges. For example, if the model has two inputs • `int8` (–128 to 127) • `uint8` (0 to 255) The data type range of the output and accumulator must equal or exceed –128 to 255. A `int16` (–32768 to 32767) data type meets this condition.

Conditions	Recommended Action
A Sum block has an input with a slope adjustment factor that does not equal the slope adjustment factor of the output.	Change the data types so the inputs, outputs, and accumulator have the same slope adjustment factor.
The net sum of the Sum block input biases does not equal the bias of the output.	Change the bias of the output scaling, making the net bias adjustment zero.
The input and output of the MinMax block have different data types.	Change the data type of the input or output.
The input of the MinMax block has a different slope adjustment factor than the output.	Change the scaling of the input or the output.
The initial condition of the Discrete-Time Integrator block is used to initialize both the state and the output.	Set the **Function Block Parameters** > **Initial condition setting** parameter to `State (most efficient)`.
Parameter overflow occurred for the Compare to Zero block. This block uses the input data type to represent zero. The input data type cannot represent zero exactly, so the input value was compared to the closest representable value of zero.	Select an input data type that can represent zero.
Parameter overflow occurred for the following Compare to Constant block. This block uses the input data type to represent its **Constant value** parameter. The **Constant value** parameter is outside the range that the input data type can represent. The input signal was compared to the closest representable value of the **Constant value** parameter.	Choose an input data type that can represent the **Constant value** parameter or change the **Constant value** parameter to match the input data type.

Capabilities and Limitations

- A Fixed-Point Designer license is required to generate fixed-point code.
- If you have a Simulink Check license, you can exclude blocks and charts from this check.

See Also

- “Identify Blocks that Generate Expensive Fixed-Point and Saturation Code” (Fixed-Point Designer)
- “Model Advisor Exclusion Overview” (Simulink Check)

Check for missing const qualifiers in model functions

Check ID: `mathworks.misra.ModelFunctionInterface`

Identify missing const qualifiers in input data pointers.

Description

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications. The check flags input data pointers that do not have a const qualifier.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
A const qualifier is not defined for the input data pointer.	Consider adding a const qualifier to the input data pointer.

See Also

- MISRA C:2012, Rule 8.13
- "MISRA C Guidelines"

Identify questionable fixed-point operations

Check ID: `mathworks.codegen.QuestionableFxptOperations`

Identify fixed-point operations that can lead to nonoptimal results.

Description

Less efficient code can result from blocks that generate cumbersome multiplication and division operations, expensive conversion code, inefficiencies in lookup table blocks, and expensive comparison code.

Results and Recommended Actions

Conditions	Recommended Action
Integer division generated code is large.	In the Configuration Parameters dialog box, on the **Hardware Implementation** pane, set the **Production hardware signed integer division rounds to** parameter to the recommended value.
Lookup Table vector of input values is not evenly spaced.	If breakpoint data is nontunable, adjust the data to have even, power of 2 spacing. See `fixpt_look1_func_approx`.
Lookup Table vector of input values is not evenly spaced when quantized, but it is very close to being evenly spaced.	If breakpoint data is nontunable, adjust the data to have even, power of 2 spacing. See `fixpt_evenspace_cleanup`.
Lookup Table vector of input values is evenly spaced, but the spacing is not a power of 2.	If breakpoint data is nontunable, adjust the data to have even, power of 2 spacing. See `fixpt_look1_func_approx`.
For a Prelookup or n-D Lookup Table block, **Index search method** is `Evenly spaced points`. Breakpoint data does not have power of 2 spacing.	If breakpoint data is nontunable, adjust the data to have even, power of 2 spacing. Otherwise, in the block parameter dialog box, specify a different **Index search method** to avoid the computation-intensive division operation.
n-D Lookup Table breakpoint data is not evenly spaced and **Index search method** is not `Evenly spaced points`.	If breakpoint data is nontunable, adjust the data to have even, power of 2 spacing and then set **Index search method** to `Evenly spaced points`.

Conditions	Recommended Action
n-D Lookup Table breakpoint data is evenly spaced and **Index search method** is `Evenly spaced points`. But the spacing is not a power of 2.	If breakpoint data is nontunable, adjust the data to have even, power of 2 spacing. See `fixpt_look1_func_approx`.
n-D Lookup Table breakpoint data is evenly spaced, but the spacing is not a power of 2. Also, **Index search method** is not `Evenly spaced points`.	Set **Index search method** to `Evenly spaced points`. Also, if the data is nontunable, consider an even, power of 2 spacing.
n-D Lookup Table breakpoint data is evenly spaced, and the spacing is a power of 2. But the **Index search method** is not `Evenly spaced points`.	Set **Index search method** to `Evenly spaced points`.
Blocks require multiword operations in generated code.	Adjust the word lengths of inputs to operations so that they do not exceed the largest word size of your processor. For more information, see “Fixed-Point Multiword Operations In Generated Code” (Fixed-Point Designer).
Blocks require cumbersome multiplication.	Restrict multiplication operations: • So the product integer size is not larger than the target integer size. • To the recommended size.
Product blocks are multiplying signals with mismatched slope adjustment factors.	Change the scaling of the output so that its slope adjustment factor is the product of the input slope adjustment factors.
Blocks multiply signals with nonzero bias.	Insert a Data Type Conversion block before and after the block containing the multiplication operation.
The inputs of the Relational Operator block have different data types.	• Change the data type and scaling of the invariant input to match other inputs. • Insert Data Type Conversion blocks before the Relational Operator block to convert both inputs to a common data type.
The inputs of the Relational Operator block have different slope adjustment factors.	Change the scaling of either input.
The output of the Relational Operator block is constant. This might result in dead code which will be eliminated by Simulink Coder.	Review your model design and either remove the Relational Operator block or replace it with the constant.

Capabilities and Limitations

- A Fixed-Point Designer license is required to generate fixed-point code.
- If you have a Simulink Check license, you can exclude blocks and charts from this check.

See Also

- 1-D Lookup Table

- n-D Lookup Table
- Prelookup
- “Identify Questionable Fixed-Point Operations” (Fixed-Point Designer)
- “Model Advisor Exclusion Overview” (Simulink Check)

Identify blocks that generate expensive rounding code

Check ID: `mathworks.codegen.ExpensiveSaturationRoundingCode`

Check for blocks that generate expensive rounding code.

Description

Generated rounding code is inefficient because of **Integer rounding mode** parameter setting.

Available with Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
Generated code is inefficient.	Set the **Function Block Parameters > Integer rounding mode** parameter to the recommended value.

Capabilities and Limitations

If you have a Simulink Check license, you can exclude blocks and charts from this check.

See Also

- “Identify Blocks that Generate Expensive Rounding Code” (Fixed-Point Designer)
- “Model Advisor Exclusion Overview” (Simulink Check)

Check for bitwise operations on signed integers

Check ID: `mathworks.misra.CompliantCGIRConstructions`

Identify Simulink blocks that contain bitwise operations on signed integers.

Description

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications, as well as code that complies with the CERT C and CWE standards.

Available with Embedded Coder and Simulink Check.

This check requires a Stateflow license when Stateflow is used in the model.

Results and Recommended Actions

Condition	Recommended Action
The model has blocks that contain bitwise operations on signed integers.	Consider using unsigned integers for bitwise operations.

Capabilities and Limitations

You can:

- The check assumes that code is generated for the whole model. When code is generated by a subsystem build or export functions, the check can product incorrect results.
- Exclude blocks and charts from this check if you have a Simulink Check license.

See Also

- MISRA C:2012, Rule 10.1
- CERT C, INT13-C
- CWE, CWE-682
- “hisl_0060: Configuration parameters that improve MISRA C:2012 compliance”
- “MISRA C:2012 Compliance Considerations”
- “Secure Coding Standards”

Check for recursive function calls

Check ID: `mathworks.misra.RecursionCompliance`

Identify recursive function calls in Stateflow charts.

Description

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications. The check flags charts that have recursive function calls.

Available with Embedded Coder and Simulink Check.

This check requires a Stateflow license.

Results and Recommended Actions

Condition	Recommended Action
Chart has a recursive function call.	Remove recursive function call.

See Also

- MISRA C:2012, Rule 17.2
- “Avoid Unwanted Recursion in a Chart” (Stateflow)

Check for equality and inequality operations on floating-point values

Check ID: `mathworks.misra.CompareFloatEquality`

Identify equality and inequality operations on floating-point values.

Description

The check flags sources causing equality or inequality operations on floating-point values.

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications, as well as code that complies with the CERT C and CWE standards.

The check does not flag blocks with equality or inequality operations on floating-point values if they are justified with a Polyspace annotation. When you run the check, the **Blocks with justification** table lists blocks with equality or inequality operations that have a justification.

Available with Embedded Coder and Simulink Check.

This check requires a Stateflow license.

Results and Recommended Actions

Condition	Recommended Action
Model object has an equality or inequality operation on a floating-point value.	Consider using non-floating-point values for equality or inequality operations.

Capabilities and Limitations

You can:

- Exclude blocks and charts from this check if you have a Simulink Check license.

See Also

- MISRA C:2012, Dir 1.1
- CERT C, FLP00-C
- CWE, CWE-697
- “Annotate Code and Hide Known or Acceptable Results” (Polyspace Bug Finder)
- “Secure Coding Standards”

Check integer word length

Check ID: `mathworks.misra.IntegerWordLengths`

Identify integer word lengths that do not comply with hardware implementation settings

Description

The check flags integers whose word lengths exceed the number of bits permitted via the hardware implementation settings.

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications, as well as code that complies with the CERT C and CWE standards.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Model object contains integer word lengths that are not compliant with hardware implementation settings.	Update the integer so its length does not exceed the permitted number of bits. You can view the permitted number of bits in the Configuration Parameters dialog box, on the **Hardware Implementation** > **Device details** pane.

Capabilities and Limitations

You can:

- Exclude blocks and charts from this check if you have a Simulink Check license.

See Also

- MISRA C:2012, Rule 10.1
- CERT C, INT13-C
- CWE, CWE-682
- “MISRA C Guidelines”
- “Model Advisor Exclusion Overview” (Simulink Check)
- “Secure Coding Standards”

Check block names

Check ID: `mathworks.codegen.BlockNames`

Checks whether block names in the **Code Perspective** pane include invalid characters.

Description

This edit-time check evaluates the block names in the **Code Perspective** pane. The check reports invalid characters in block names, except for:

- Blocks that are ignored or not recommended for code generation
- Virtual Subsystem blocks

The check verifies that block names comply with these guidelines:

Form:

name:

- Does not start with a number
- Does not include spaces at the beginning of a block name
- Does not use double byte characters
- Carriage returns are allowed

Allowed Characters:

name:

a b c d e f g h i j k l m n o p q r s t u v w x y z

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

0 1 2 3 4 5 6 7 8 9 _

Results and Recommended Actions

Condition	Recommended Action
The block name in the **Code Perspective** pane does not conform to the guidelines.	Update the block name to comply with the guidelines.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library-linked blocks.
- Analyzes content in masked subsystems.
- Allows exclusions of blocks and charts.

See Also

- "Simulink Built-In Blocks That Support Code Generation"

Embedded Coder Tools

Embedded Coder Dictionary

Create code definitions, which control code generation for model data and functions

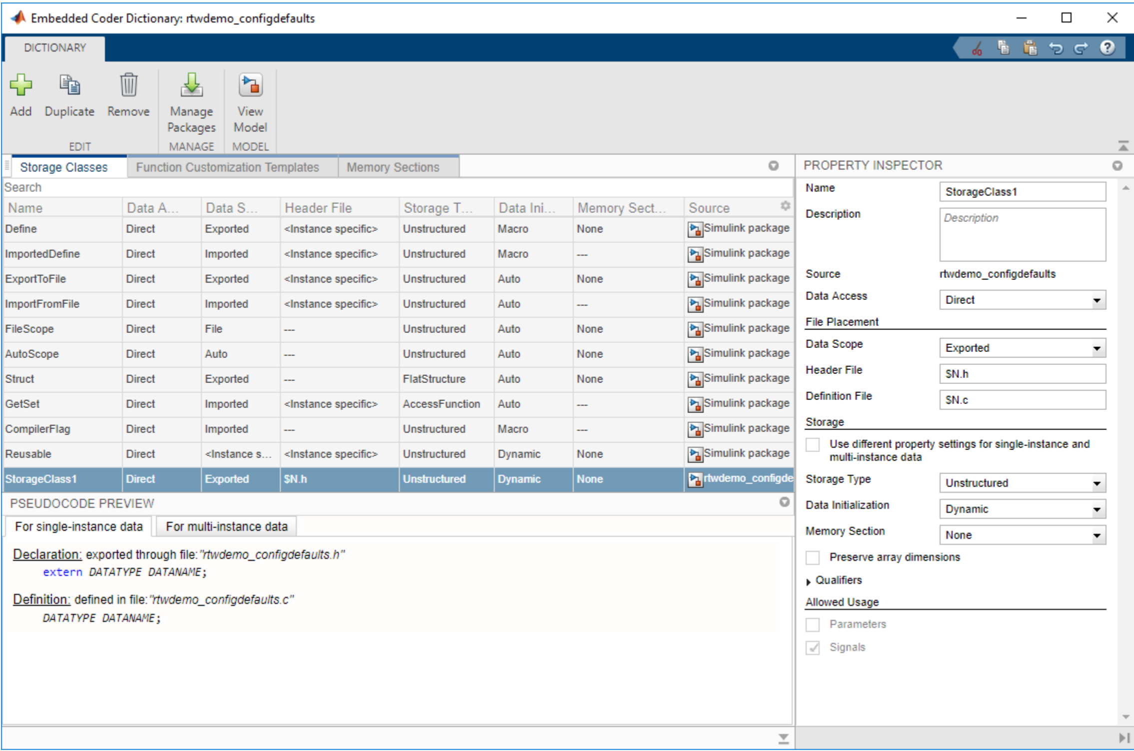
Description

The Embedded Coder Dictionary is a graphical interface for creating custom code definitions. By applying these definitions in models, you and your users can generate code that conforms to a specific software architecture by default. For example, you can create your own storage class, which you and your users can apply by default to a category of model data, such as root-level inputs, or to individual data elements, such as parameters.

You can create these types of code definitions:

- Storage classes, which control the code generated for model data.
- Function customization templates, which control naming of model entry-point functions, such as *model*`_step`. The templates also apply memory sections to the entry-point functions.
- Memory sections, which control the placement of data and functions in memory. The generated code includes custom decorations, such as pragmas, whose syntax you specify.

For general information about creating code generation definitions, see “Define Storage Classes, Memory Sections, and Function Templates for Software Architecture”.



The Embedded Coder Dictionary has a tab for each type of code definition. In each tab, you configure the properties of code definitions. Use the table to configure properties and compare definitions side

by side. To access properties that do not appear in the table, use the Property Inspector. To verify results as you configure properties, use the pseudocode preview.

You can apply the definitions that you create in the dictionary to model elements by using the Code Mappings editor (see “Configure Default C Code Generation for Categories of Data Elements and Functions”). To create storage classes and memory sections that you can use outside of the Code Mappings editor, use the Custom Storage Class Designer (see “Create Code Definitions for External Data Objects”).

Open the Embedded Coder Dictionary

- To open an Embedded Coder Dictionary, in a model window, on the **C Code** tab, select **Code Interface > Embedded Coder Dictionary**.

  The Embedded Coder Dictionary window displays code generation definitions that are stored in the model file. If the model is linked to a data dictionary, the window also displays definitions that are stored in that data dictionary or, if applicable, in a referenced dictionary. The **Source** column indicates where each definition is stored.
- To open the Embedded Coder Dictionary in a Simulink data dictionary, in the Model Explorer **Model Hierarchy** pane:

  1. Under the dictionary node, select the **Embedded Coder** node.

     If you do not see the node, right-click the dictionary node and select **Show Empty Sections**.
  2. In the Dialog pane (the right pane), click **Open Embedded Coder Dictionary**.

Examples

Create and Verify Storage Class

In a model, create a storage class that aggregates internal model data, including block states, into a structure whose characteristics you can control. Then, verify the storage class by generating code from the model.

1. Open the example model `rtwdemo_roll`.

   ```
   rtwdemo_roll
   ```
2. If the model does not open in the Embedded Coder app, open the app and click the **C Code** tab.
3. On the **C Code** tab, select **Code Interface > Embedded Coder Dictionary**. The Embedded Coder Dictionary window displays code generation definitions that are stored in the model file.
4. In the Embedded Coder Dictionary window, click **Add**.
5. Select the new storage class that appears at the bottom of the list, `StorageClass1` . In the Property Inspector pane on the right, set the property values listed in this table.

Property	Value
Name	`InternalStruct`
Header File	`internalData_$R.h`
Definition File	`internalData_$R.c`
Storage Type	`Structured`

Property	Value
Structure Properties > Type Name	internalData_T_$M
Structure Properties > Instance Name	internalData_$M

After making your changes, in the bottom pane, verify that the pseudocode preview reflects what you expect.

6. Return to the model editor. To open the Code Mappings editor, below the canvas, double-click Code Mappings. On the **Data Defaults** tab, expand the **Signals** section. Select the **Signals, states, and internal data** row and set **Storage Class** to `InternalStruct`.
7. In the Configuration Parameters dialog box, on the **Code Generation > Code Placement** pane, set **File packaging format** to `Modular`.
8. Generate code.
9. In the Simulink Editor Code view, open and inspect the file `internalData_rtwdemo_roll.h`. The file defines the structure type `internalData_T_`, whose fields represent block states in the model.

```
/* Storage class 'InternalStruct', for system '<Root>' */
typedef struct {
  real32_T FixPtUnitDelay1_DSTATE;     /* '<S7>/FixPt Unit Delay1' */
  real32_T Integrator_DSTATE;          /* '<S1>/Integrator' */
  int8_T Integrator_PrevResetState;    /* '<S1>/Integrator' */
} internalData_T_;
```

The file also declares a global structure variable named `internalData_`.

```
/* Storage class 'InternalStruct' */
extern internalData_T_ internalData_;
```

10. Open and inspect the file `internalData_rtwdemo_roll.c`. The file allocates memory for `internalData_`.

```
/* Storage class 'InternalStruct' */
internalData_T_ internalData_;
```

Create Function Customization Template

With a function template, you can specify a rule that governs the names of generated entry-point functions. This technique helps save time and maintenance effort in a model with many entry-point functions, such as an export-function model or a multirate, multitasking model.

This example shows how to create a function template that specifies the naming rule `func_$N_$R`. `$N` is the base name of each generated function and `$R` is the name of the Simulink model.

1. Open the example model `rtwdemo_mrmtbb`.
2. Update the block diagram. This multitasking model has two execution rates, so the generated code includes two corresponding entry-point functions.
3. In the model, set model configuration parameter **System target file** to `ert.tlc`. To use a function customization template, you must use an ERT-based system target file.
4. In the Simulink Editor, open the Embedded Coder app and open the Embedded Coder Dictionary.
5. In the Embedded Coder Dictionary, on the **Function Customization Templates** tab, click **Add**.
6. For the new function template, set these properties:

   - **Name** to `myFunctions`.
   - **Function Name** to `func_$N_$R`.

After making your changes, verify that the pseudocode preview reflects what you expect.

7. In the model window, open the Code Mappings editor. On the **Function Defaults** tab, for the **Initialize/Terminate** and **Execution** rows, set **Function Customization Template** to `myFunctions`.
8. Generate code.
9. In the Code view, open and inspect the file `rtwdemo_mrmtbb.c`. The file defines the two execution functions, `func_step0_rtwdemo_mrmtbb` and `func_step1_rtwdemo_mrmtbb`, whose names conform to the rule that you specified in the function template.

Create Memory Section

For an example that shows how to create a memory section, see “Control Data and Function Placement in Memory by Inserting Pragmas”.

Create Storage Class for Use with Statically and Dynamically Initialized Data

This example shows how to create a storage class that places global variable definitions and declarations in files whose names depend on the model name. You create two copies of the storage class so that you can use one copy with parameter data (the data category **Model parameters**) and one copy with other data.

Typically, the generated code initializes parameter data statically, outside any function, and initializes other data dynamically, in the model initialization function. When you create a storage class by using the Custom Storage Class Designer or an Embedded Coder Dictionary, you set the **Data Initialization** property to specify the initialization mechanism.

In an Embedded Coder Dictionary, for each storage class, you must select `Static` or `Dynamic` initialization. Consider creating one copy of the storage class for parameter data (`Static`) and one copy for other data (`Dynamic`).

Create Storage Class

1. Open example model `rtwdemo_roll`.
2. If the **C Code** tab is not open, open the Embedded Coder app and click the **C Code** tab.
3. Select **Code Interface > Embedded Coder Dictionary**
4. In the Embedded Coder Dictionary, click **Add**.
5. For the new storage class, set these properties:

   - **Name** to `SigsStates`
   - **Header File** to `$R_my_data.h`
   - **Definition File** to `$R_my_data.c`
   - **Data Initialization** to `Dynamic`

   After making your changes, verify that the pseudocode preview reflects what you expect.
6. Click **Duplicate**. A new storage class, `SigsStates_copy`, appears.
7. For the new storage class, set these properties:

   - **Name** to `Params`
   - **Data Initialization** to `Static`

   After making your changes, verify that the pseudocode preview reflects what you expect.

Apply Storage Class and Generate Code

1. Return to the model and open the Code Mappings editor. Below the model canvas, double-click **Code Mappings - C**.
2. On the **Data Defaults** tab, for the **Parameters** > **Model Parameters** row, in the **Storage Class** column, select `Params`.
3. For the **Signals** > **Signals, states, and internal data** row, set **Storage Class** to `SigsStates`.
4. Configure some parameter data elements in the model so that optimizations do not eliminate those elements from the generated code. On the **Modeling** tab, click **Design** > **Model Workspace**.
5. In the Model Explorer, on the center pane, select the three rows that correspond to the variables `dispGain`, `intGain`, and `rateGain` in the model workspace.
6. Right-click one of the rows and click `Convert to parameter object`. The Model Data Editor converts the workspace variables to `Simulink.Parameter` objects.
7. In the row for the parameter `dispGain`, in the **Storage Class** column, click **Configure**. The model window highlights the row for the `dispGain` parameter in the Code Mappings editor.
8. For each variable, in the **Storage Class** column, select `Model default: Params`, which means they acquire the default storage class that you specified for **Model parameters**.
9. In the Configuration Parameters dialog box, on the **Code Generation** > **Code Placement** pane, set **File packaging format** to `Modular`.
10. Generate code.
11. In the Code view, open and inspect the files `rtwdemo_roll_my_data.c` and `rtwdemo_roll_my_data.h`. These files define and declare global variables that correspond to the parameter objects and some block states, such as the state of the Integrator block in the `BasicRollMode` subsystem.

```
/* Storage class 'SigsStates' */
real32_T rtFixPtUnitDelay1_DSTATE;
real32_T rtIntegrator_DSTATE;
int8_T rtIntegrator_PrevResetState;

/* Storage class 'Params' */
real32_T dispGain = 0.75F;
real32_T intGain = 0.5F;
real32_T rateGain = 2.0F;
```

Refer to Code Generation Definitions in a Package

You can configure an Embedded Coder Dictionary to refer to code generation definitions that you store in a package (see “Create Code Definitions for External Data Objects”). Those definitions then appear available for selection in the Code Mappings editor. In this example, you configure the Embedded Coder Dictionary in `rtwdemo_roll` to refer to definitions stored in the built-in example package `ECoderDemos`.

1. Open the Embedded Coder Dictionary for `rtwdemo_roll`. For instructions, see “Create and Verify Storage Class” on page 23-3.
2. In the Embedded Coder Dictionary window, click **Manage Packages**.
3. In the Manage Packages dialog box, click **Refresh**. Wait until more options appear in the **Select package** drop-down list.
4. Set **Select package** to `ECoderDemos` and click **Load**.

   In the Embedded Coder Dictionary window, on the **Storage Classes** tab, the table shows the storage classes defined in the `ECoderDemos` package. Now, in `rtwdemo_roll`, you can select these storage classes in the Code Mappings editor on the **Data Defaults** tab.

5 To unload the package, in the Manage Packages dialog box, select the package in the **Select package** drop-down list and click **Unload**.

Share Code Generation Definitions Between Models by Using Simulink Data Dictionary

For an example that shows how to share code generation definitions between models by using data dictionaries, see “Share Embedded Coder Dictionary Definition Between Models”.

Configure Default Code Mappings in a Shared Coder Dictionary

For an example that shows how to configure default code mappings in a shared Embedded Coder Dictionary, see “Configure Default Code Mapping in a Shared Dictionary”.

Parameters

These properties appear in the Property Inspector pane of the Embedded Coder Dictionary window. In the table, some properties appear as columns to facilitate batch editing.

Storage Classes

`Name` — Name of storage class

`StorageClass1` (default) | text

Name of the storage class. The name must be unique among the storage classes in the dictionary.

For lists of built-in and example storage classes that Simulink provides, see “Choose Storage Class for Controlling Data Representation in Generated Code”.

`Description` — Purpose and functionality of storage class

text

Custom text that you can use to describe the purpose and functionality of the storage class.

`Source` — Location of storage class definition

text

This property is read-only.

The location of the storage class definition.

- `Built-in` — Provided by Simulink.
- Model name — Defined in a Simulink model.
- Dictionary name — Defined in a Simulink data dictionary (see “What Is a Data Dictionary?”).
- Package name — Defined in the Simulink package or in a custom package (see “Create Storage Classes by Using the Custom Storage Class Designer”).

`Data Access` — Specification to access the data

`Direct` (default) | `Function`

Specification to access data associated with the model. Access the data directly (`Direct`) or through customizable `get` and `set` functions (`Function`). For more information, see “Access Data Through Functions by Using Storage Classes in Embedded Coder Dictionary”.

Dependencies

- Setting this property to `Function`:
  - Sets **Data Scope** to `Imported`.
  - Means that you cannot specify multi-instance properties.
  - Enables these properties:
    - **Access Mode**
    - **Allowed Access**
    - **Name of Getter**
    - **Name of Setter**
  - Disables the **Preserve array dimensions** property. To preserve dimensions of multidimensional arrays in the generated code, set **Data Access** to `Direct`.

`Data Scope` — Specification to generate data definition

`Exported` (default) | `Imported`

Specification that the generated code define the data (`Exported`) or import (`Imported`) the data definition from external code. Built-in storage classes and storage classes in packages such as Simulink can use other scope options, such as `File`.

Dependencies

- Setting this property to `Imported`:
  - Disables **Definition File**. To include your external source code file in the build process, use model configuration parameters. For an example, see “Configure Data Interface”.
  - Means that you cannot set **Header File** to `$N.h`, though you can use the `$N` token.
- To set this property to `Exported`, you must use one of the tokens `$N` or `$R` in the value of **Header File**.

`Header File` — Name of header file that declares data

`$N.h` (default) | text

Name of the header file that declares the data, specified as a name or naming rule. A naming rule includes a combination of text and tokens. Valid tokens are listed in this table.

Token	Description
`$R`	Name of root model
`$N`	Name of associated data element
`$G`	Name of storage class
`$U`	User token text, which you specify for a model as described in “Identifier Format Control”

Dependencies

- If you set **Data Scope** to `Exported`, you must use one of the tokens `$R` or `$N` in the value of this property.
- If you set **Data Scope** to `Imported`, you cannot set the value of this property to `$N.h`, but you can use the `$N` token.

`Definition File` — Name of source file that defines data

`$N.c` (default) | text

Name of the source file that defines the data, specified as a name or naming rule. A naming rule includes a combination of text and tokens. Valid tokens are listed in this table.

Token	Description
`$R`	Name of root model
`$N`	Name of associated data element
`$G`	Name of storage class
`$U`	User token text, which you specify for a model as described in "Identifier Format Control"

Dependencies

Setting **Data Scope** to `Imported` disables **Definition File**. To include your external source code file in the build process, use model configuration parameters. For an example, see "Configure Data Interface".

`Access Mode` — Specification to access data through functions

`Value` (default) | `Pointer`

Specification for the storage class to access data associated with the model through functions by using `Value` or `Pointer`. For more information, see "Access Data Through Functions by Using Storage Classes in Embedded Coder Dictionary".

Dependencies

This property is enabled only when you set **Data Access** to `Function`.

`Allowed Access` — Specification to allow access to data through functions

`Read/Write` (default) | `Read Only` | `Write Only`

Specification for the storage class to allow read and write (`Read/Write`), read-only (`Read Only`), or write-only (`Write Only`) access to the data.

Dependencies

This property is enabled only when you set **Data Access** to `Function`.

`Name of Getter` — Name of the get function that fetches the associated data

`get_$N$M` (default) | text

Name of the `get` function that fetches the associated data, specified as a name or naming rule. A naming rule includes a combination of text and tokens. Valid tokens are listed in this table.

Token	Description
`$N`	Name of associated data element (required)
`$R`	Name of root model
`$M`	Mangle text that ensures uniqueness
`$U`	User token text. See "Identifier Format Control".

Dependencies

This property is enabled only when you set **Data Access** to `Function`.

`Name of Setter` — Name of the `set` function that modifies the associated data

`set_$N$M` (default) | text

Name of the `set` function that fetches the modifies data, specified as a name or naming rule. A naming rule includes a combination of text and tokens. Valid tokens are listed in this table.

Token	Description
`$N`	Name of associated data element (required)
`$R`	Name of root model
`$M`	Mangle text that ensures uniqueness
`$U`	User token text. See “Identifier Format Control”.

Dependencies

This property is enabled only when you set **Data Access** to `Function`.

`Use different property settings for single-instance and multi-instance data` — Specification to assign separate storage settings

`off` (default) | `on`

Specification for the storage class to use either the storage settings that you specify in the **Single-instance storage** section or the storage settings that you specify in the **Multi-instance storage** section. When you apply the storage class to a data item, the Embedded Coder Dictionary determines if it is a single-instance storage class or a multi-instance storage class by the type of data and by the context of the model within the model reference hierarchy.

Dependencies

Selecting this property enables the sections **Single-instance storage** and **Multi-instance storage**. The properties **Storage Type**, **Type Name**, and **Instance Name** appear in both the **Single-instance storage** and **Multi-instance storage** sections.

`Storage Type` — Specification to aggregate data into a structure

`Unstructured` (default) | `Structured`

Specification to aggregate the data that uses the storage class into a structure in the generated code. Each data element appears in the code as a field of the structure. To create a structure, use `Structured`.

Dependencies

Setting this property to `Structured` enables **Type Name** and **Instance Name**.

`Type Name` — Name of structure type

`$R$N$G$M` (default) | text

Name of the structure type in the generated code, specified as a name or a naming rule. A naming rule includes a combination of text and tokens. Valid tokens are listed in this table.

Token	Description
`$R`	Name of root model
`$N`	Base name of associated function, such as `step`
`$G`	Name of storage class
`$U`	User token text, which you specify for a model as described in "Identifier Format Control"
`$M`	Name-mangling text inserted, if necessary, to avoid name collisions

Dependencies

Setting **Storage Type** to `Structured` enables this property.

`Instance Name` — Name of structure variable

`$N$G$M` (default) | text

Name of the structure variable in the generated code, specified as a name or a naming rule. A naming rule includes a combination of text and tokens. Valid tokens are listed in this table.

Token	Description
`$R`	Name of root model
`$N`	Base name of associated function, such as `step`
`$G`	Name of storage class
`$U`	User token text, which you specify for a model as described in "Identifier Format Control"
`$M`	Name-mangling text inserted, if necessary, to avoid name collisions

Dependencies

Setting **Storage Type** to `Structured` enables this property.

`Data Initialization` — How to initialize data

`Auto` (default) | `Dynamic` | `Static` | `None`

Specification that the generated code initialize the data.

- `Auto` — The generated code statically initializes parameter data and dynamically initializes signal and state data.
- `Dynamic` — The generated code initializes the data as part of the model initialization entry-point function.
- `Static` — The generated code initializes the data in the same statement that defines and allocates memory for the data. The assignment statement appears at the top of a `.c` or `.cpp` source file, outside of a function.
- `None` — The generated code does not initialize the data.

Dependencies

- If you select **Const**, you cannot set this property to `Dynamic`.
- Setting this property to `Dynamic` disables **Const**.

`Memory Section` — Location in memory to allocate data

`None` (default) | existing memory section

Location in memory to allocate data, specified as a memory section that exists in the Embedded Coder Dictionary on the **Memory Sections** tab. For information about memory sections, see “Control Data and Function Placement in Memory by Inserting Pragmas”.

`Preserve array dimensions` — Specification to preserve dimensions of multidimensional arrays

`off` (default) | `on`

Specification for the storage class to preserve dimensions of multidimensional arrays in the generated code. For more information, see “Preserve Dimensions of Multidimensional Arrays in Generated Code”.

`Const` — Specification to apply `const` qualifier

`off` (default) | `on`

Specification to apply the `const` qualifier to the data.

Dependencies

- If you select this property, you cannot set **Data Initialization** to `Dynamic`.
- Setting **Data Initialization** to `Dynamic` disables this property.

`Volatile` — Specification to apply `volatile` qualifier

`off` (default) | `on`

Specification to apply the `volatile` qualifier to the data.

`Other Qualifier` — Specification to apply a custom qualifier

text

Specification to apply a custom qualifier to the data. For example, some memory architectures support qualifiers `far` and `huge`.

Do not use this property to apply the keyword `static`. Instead, use the built-in storage class `FileScope`, which you cannot apply with the Code Mappings editor. See “Choose Storage Class for Controlling Data Representation in Generated Code”.

`Parameters` — Whether to allow usage with model parameters

`off` (default) | `on`

Specification indicating whether to allow usage of the storage class with model parameters.

Dependencies

- Setting **Data Initialization** to `Static` enables this property.
- Setting **Data Initialization** to `Dynamic` disables this property.
- To set the value of this property, set **Data Initialization** to `None`.

`Signals` — Whether to allow usage with model signals

`on` (default) | `off`

Specification indicating whether to allow usage of the storage class with model signals.

Dependencies

- Setting **Data Initialization** to `Dynamic` enables this property.
- Setting **Data Initialization** to `Static` disables this property.
- To set the value of this property, set **Data Initialization** to `None`.

Function Customization Templates

`Name` — Name of function template

`FunctionTemplate1` (default) | text

Name of the template. The name must be unique among the function templates in the dictionary. Embedded Coder provides the built-in templates listed in this table.

Template	Description
`ModelFunction`	In the Code Mappings editor, use for entry-point functions for initialization, execution, termination, and reset (see "Configure Default Code Generation for Functions")
`UtilityFunction`	In the Code Mappings editor, use for shared utility functions (see "Configure Default Code Generation for Functions")

`Description` — Purpose and functionality of function template

text

Custom text that you can use to describe the purpose and functionality of the function template.

`Source` — Location of function template definition

text

This property is read-only.

The location of the function template definition.

- Model name — Defined in a Simulink model.
- Dictionary name — Defined in a Simulink data dictionary (see "What Is a Data Dictionary?").

`Function Name` — Names of generated functions

`$R$N` (default) | text

Names of the functions in the generated code, specified as a naming rule. A naming rule includes a combination of text and tokens. Valid tokens are listed in this table.

Token	Description
`$R`	Name of root model
`$N`	Base name of associated function, such as `step`
`$U`	User token text, which you specify for a model as described in "Identifier Format Control"
`$C`	For shared utility functions, a checksum inserted to avoid name collisions
`$M`	Name-mangling text inserted, if necessary, to avoid name collisions

`Memory Section` — Location in memory to allocate function

`None` (default) | existing memory section

Location in memory to allocate function, specified as a memory section that exists in the Embedded Coder Dictionary on the **Memory Sections** tab. For information about memory sections, see “Control Data and Function Placement in Memory by Inserting Pragmas”.

Memory Sections

`Name` — Name of memory section

text

Name of the memory section. The name must be unique among the memory sections in the dictionary. Embedded Coder provides the built-in memory sections listed in this table.

Memory Section	Description
`MemConst`	Apply the storage type qualifier `const` to the data.
`MemVolatile`	Apply the storage type qualifier `volatile` to the data.
`MemConstVolatile`	Apply the storage type qualifiers `const` and `volatile` to the data.

`Description` — Purpose and functionality of memory section

text

Custom text that you can use to describe the purpose and functionality of the memory section.

`Source` — Location of memory section definition

text

This property is read-only.

The location of the memory section definition.

- Model name — Defined in a Simulink model.
- Dictionary name — Defined in a Simulink data dictionary (see “What Is a Data Dictionary?”).
- Package name — Defined in the Simulink package or in a custom package (see “Create Code Definitions for External Data Objects”).

`Comment` — Comment to insert in the generated code

text

Code comment that the code generator includes with the pragmas or other decorations that you specify with **Pre Statement** and **Post Statement**.

`Pre Statement` — Code to insert before data or function code

text

Code, such as pragmas, to insert before the definitions and declarations of the data or functions that are in the memory section.

You can use the token `$R` to represent the name of the model that uses the memory section.

When you set **Statements Surround** to `Each variable`, you can use the token `$N` to represent the name of each variable or function that uses the memory section.

`Post Statement` — Code to insert after data or function code
text

Code, such as pragmas, to insert after the definitions and declarations of the data or functions that are in the memory section.

You can use the token `$R` to represent the name of the model that uses the memory section.

When you set **Statements Surround** to `Each variable`, you can use the token `$N` to represent the name of each variable or function that uses the memory section.

`Statements Surround` — Specification to wrap data and functions separately or in a group
`Each variable` (default) | `Group of variables`

Specification to insert code statements (**Pre Statement** and **Post Statement**):

- Around each variable and function that uses the memory section. Select `Each variable`.
- Once, around the entire memory section. The generated code aggregates the variable and function definitions into a contiguous code block and surrounds the block with the statements. Select `Group of variables`.

Limitations

- A storage class or function customization template that you create in an Embedded Coder Dictionary cannot use a memory section that you load from a package (as described in “Refer to Code Generation Definitions in a Package” on page 23-6). Use a memory section defined in the Embedded Coder Dictionary.
- You cannot create code generation definitions in a `.mdl` model file.
- For additional limitations for code generation definitions in the Embedded Coder Dictionary of a data dictionary (`.sldd` file), see “Deploy Code Generation Definitions”.

See Also

Code Mappings editor

Topics

“Define Storage Classes, Memory Sections, and Function Templates for Software Architecture”
“Deploy Code Generation Definitions”
“Generate Code to Conform to Software Architecture by Sharing and Copying Default Settings Between Models”
“Flexible Storage Class for Different Model Hierarchy Contexts”

Introduced in R2018a

Code Mappings Editor – C

Associate model elements with code definitions

Description

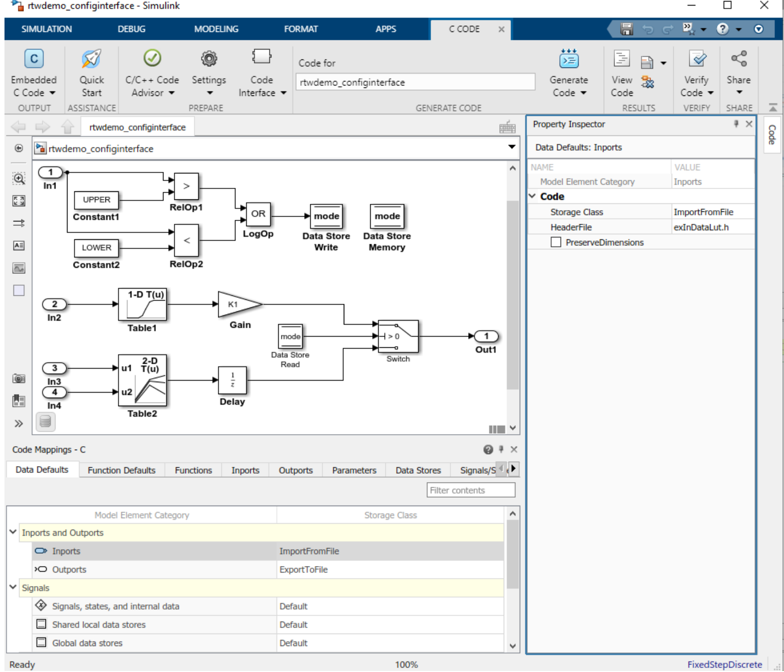
The Code Mappings editor is a graphical interface where you can configure data elements and entry-point functions in a model, excluding referenced models, for code generation. Each model in a model reference hierarchy has its own code mappings. Associate each category of model data element with a specific storage class and each category of model entry-point function with a specific function customization template throughout a model. Then, override those settings, as needed, for specific data elements and functions.

A storage class defines properties such as appearance and location, which the code generator uses when producing code for associated data. Function customization templates define how the code generator produces code for associated functions. If you leave the storage class or function customization template set to `Default`, you can configure a memory section for that data or function category.

To configure data elements and functions for code generation, use the tables in the Code Mappings editor display:

- **Data Defaults**
- **Function Defaults**
- **Functions**
- **Inports**
- **Outports**
- **Parameters**
- **Data Stores**
- **Signals/States**

When you select a row in the active table, the **Code** section of the Property Inspector displays storage class or function customization template property settings for the selected data element or function.



Before you can configure a signal for code generation, add the signal to the model code mappings. Add and remove signals from the code mappings by pausing on the ellipsis that appears above or below a signal line to open the action bar. Click the **Add Signal** or **Remove Signal** button. These buttons are also available in the Code Mappings editor on the **Signals/States** tab.

Open the Code Mappings Editor – C

Do one of the following:

- Open the Embedded Coder app. On the **C Code** tab, select **Code Interface** > **Default Code Mappings** or **Code Interface** > **Individual Element Code Mappings**.
- Open the Embedded Coder app. On the **C Code** tab, in the bottom left corner of the Simulink Editor window, click the **Code Mappings - C** tab.
- In the model canvas of the Simulink Editor window, click the perspective control in the lower-right corner and select **Code**. Then, click the **Code Mappings - C** tab.

Examples

Configure Code Generation for Root Inport and Outport Blocks

Configure code generation for the root Inport and Outport blocks throughout a model. Applying default configurations can save time, especially for large-scale models that use a significant amount of data. After applying default mappings, you can adjust mappings for individual data elements.

Set Up Example Environment

1 Copy external code files into a writable folder.

```
copyfile(fullfile(matlabroot,'toolbox','rtw','rtwdemos','roll_input_data.c'));
copyfile(fullfile(matlabroot,'toolbox','rtw','rtwdemos','roll_input_data.h'));
```

```
copyfile(fullfile(matlabroot,'toolbox','rtw','rtwdemos','roll_heading_mode.c'));
copyfile(fullfile(matlabroot,'toolbox','rtw','rtwdemos','roll_heading_mode.h'));
```

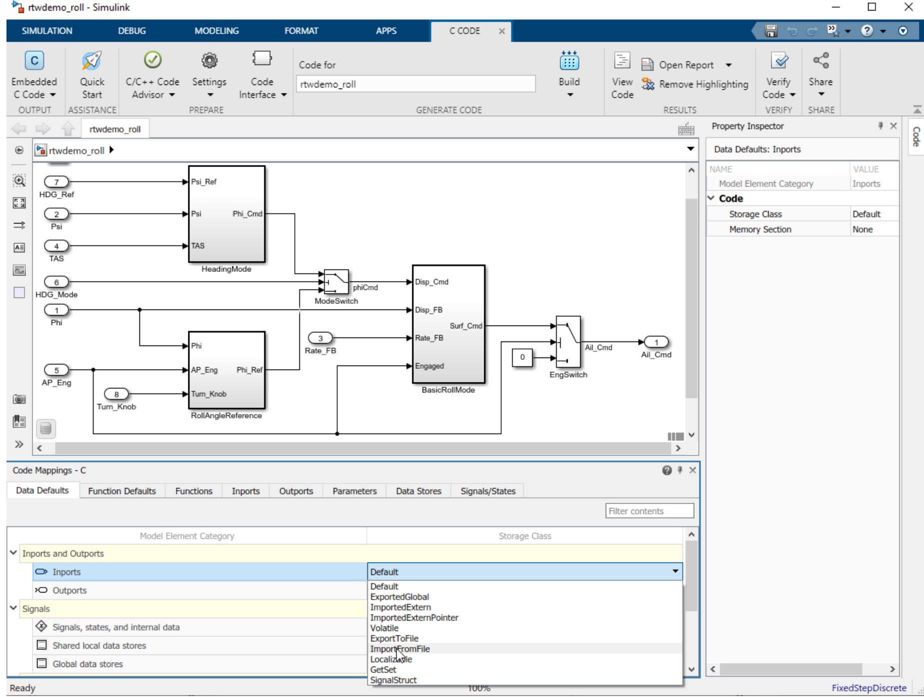
2 Open model `rtwdemo_roll`. Save a copy of the model in the folder where you copied the external code files.
3 Open the **Embedded Coder** app.

Configure Default Mappings

Configure the code generator to:

- Use header file `roll_input_data.h` to declare the variables representing model Inport blocks.
- Represent variables for model Outport blocks as separate global variables.
- Define output variables in `roll_output_data.c` and declare them in `roll_output_data.h`.
- Configure names that the code generator uses for variables it produces in the code for Inport blocks.

1 In the **C Code** tab, select **Code Interface > Default Code Mappings**.
2 In the **Data Defaults** tab, under **Inports and Outports**, select the row for **Inports**. Then, set the storage class to `ImportFromFile`.



3 In the Property Inspector, set **Header File** to `roll_input_data.h`.
4 Set the storage class for model element category **Outports** to `ExportToFile`.
5 Set **Header File** to `roll_output_data.h` and **Definition File** to `roll_output_data.c`.

Configure Individual Inports for Default Configuration

1 In the Code Mappings editor, click the **Inports** tab. The storage class for each inport is set to `Auto`, which means that the code generator might eliminate or change the representation of relevant code for optimization purposes. If optimizations are not possible, the code generator applies the default configuration for inports.
2 Force the code generator to use the default configuration for inports, storage class `ImportFromFile` with external header file `roll_input_data.h`. Press the **Ctrl** key and select

the inports. For one of the selected inports, set the storage class to `Model default: ImportFromFile`. The editor updates the storage class setting for the selected inports.

Override Default Mappings

Override the default source location for inport variable `HDG_Mode`. That variable is declared in the external file `roll_heading_mode.h`.

1. In the Code Mappings editor, click the **Inports** tab.
2. Select the `HDG_Mode` row.
3. Set **Storage Class** to `ImportFromFile`.
4. In the Property Inspector, under the **Code** section, set **Header File** to `roll_heading_mode.h`.
5. Configure the code generator to produce variable names in the code for the Inport blocks that match the variable names in external files `roll_input_data.h` and `roll_heading_mode.h`. On the **Inports** tab, select each Inport block and in the Property Inspector set **Identifier** to the block name. When the storage class is set to a value other than `Auto`, you must specify a value for the **Identifier** storage class property.

Include External Source Files In Code Generation and Build Process

Include external source files `roll_input_data.c` and `roll_heading_mode.c` in the code generation and build process. Set the model configuration parameter **Source files** to `roll_input_data.c roll_heading_mode.c`.

Save the model.

Generate and Verify Code

Generate code and verify that the code generated for the Inport and Output blocks appears as you expect.

- `rtwdemo_roll.h` includes these header files associated with storage classes:

  ```
  #include "roll_output_data.h"
  #include "roll_input_data.h"
  #include "roll_heading_mode.h"
  ```
- `roll_heading_mode.c` includes `roll_heading_mode.h` and defines variable `HDG_Mode`.

  ```
  #include "roll_heading_mode.h"

  boolean_T HDG_Mode;
  ```
- `roll_input_data.c` defines the variables declared in `roll_input_data.h`.

  ```
  #include "roll_input_data.h"

  boolean_T AP_Eng;
  real32_T HDG_Ref;
  real32_T Rate_FB;
  real32_T Phi;
  real32_T Psi;
  real32_T TAS;
  real32_T Turn_Knob;
  ```
- `roll_output_data.c` includes this exported data definition:

  ```
  real32_T Ail_Cmd;
  ```

- `roll_output_data.h` includes this exported data declaration:

```
extern real32_T Ail_Cmd;
```

Configure Default Function Names for Entry-Point Functions

By default, the code generator uses the identifier naming rule `$R$N` to name entry-point functions. `$R` is the name of the root model. `$N` is the name of the function, for example, `initialize`, `step`, and `terminate`. To integrate generated code with existing external code or to comply with naming standards or guidelines, you can adjust the default naming rule. This example shows how to add the text string `myproj_` as a prefix to `$R$`. Adjusting the default naming rule can save time, especially for multirate models for which the code generator produces a unique `step` function for each rate.

Set Up Example Environment

1. Open model `rtwdemo_multirate_multitasking`. Save a copy to a writable folder.
2. Open the Embedded Coder app. The **C Code** tab opens, which includes the Code Mappings editor.

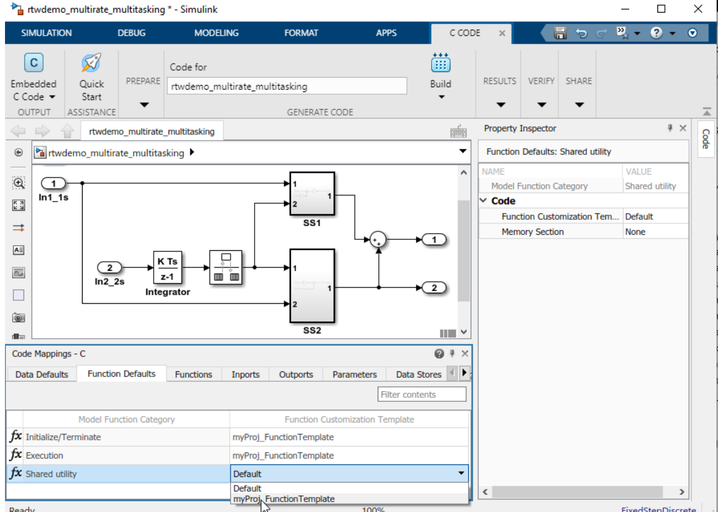
Define Function Naming Rule

Create a function customization template that defines the naming rule `myproj_$R$N`.

1. Open the Embedded Coder Dictionary. In the **C Code** tab, select **Code Interface** > **Embedded Coder Dictionary**.
2. Click the **Function Customization Templates** tab.
3. Click **Add**.
4. In the **Name** column of the new table row, name the new template `myproj_FunctionTemplate`.
5. In the **Function Name** column, enter the naming rule `myproj_$R$N`.
6. Close the Embedded Coder Dictionary.

Configure Default Mappings

1. In the **C Code** tab, select **Code Interface** > **Default Code Mappings**.
2. Click the **Function Defaults** tab.
3. For the **Initialize/Terminate** and **Execution** function categories, change the default function customization template from `Default` to `myproj_FunctionTemplate`.



4 Save the model.

Generate and Review Code

Generate code and verify the entry-point function names.

```
void myproj_rtwdemo_multirate_multitasking_step0(void) /* Sample time: [1.0s, 0.0s] */
{
  (rtM->Timing.RateInteraction.TID0_1)++;
  if ((rtM->Timing.RateInteraction.TID0_1) > 1) {
       rtM->Timing.RateInteraction.TID0_1 = 0;
  }

  if (rtM->Timing.RateInteraction.TID0_1 == 1) {
      rtDW.RateTransition = rtDW.RateTransition_Buffer0;
  }
      rtY.Out2 = 2.0 * rtDW.RateTransition + rtU.In1_1s;
      rtY.Out1 = (3.0 * rtDW.RateTransition + rtU.In1_1s) * 5.0 + rtY.Out2;
}

/* Model step function for TID1 */
void myproj_rtwdemo_multirate_multitasking_step1(void) /* Sample time: [2.0s, 0.0s] */
{
  rtDW.RateTransition_Buffer0 = rtDW.Integrator_DSTATE;
  rtDW.Integrator_DSTATE += 2.0 * rtU.In2_2s;
}

void myproj_rtwdemo_multirate_multitasking_initialize(void)
{
  /* (no initialization code required) */
}

void myproj_rtwdemo_multirate_multitasking_terminate(void)
{
  /* (no terminate code required) */
}
```

Customize Individual Entry-Point Functions

For your model, you can customize the names of most entry-point functions and the arguments of execution functions, such as step functions and Simulink functions. This example shows how to customize the entry-point functions for the model `rtwdemo_roll`.

Set Up the Environment

1 Copy external code files into a writable folder.

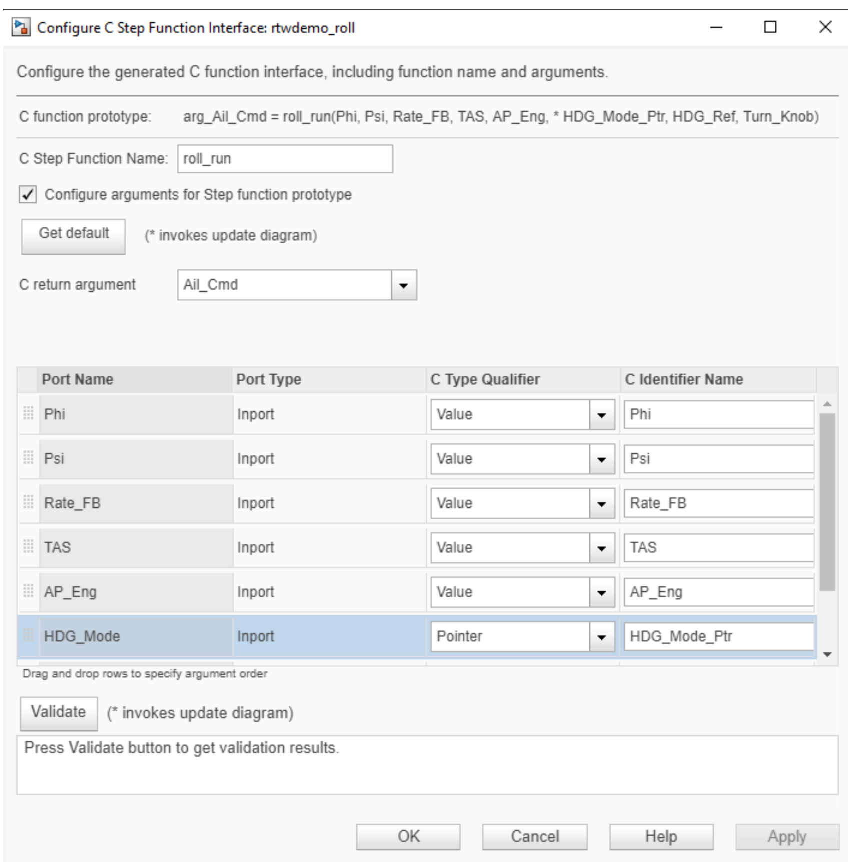
   ```
   copyfile(fullfile(matlabroot,'toolbox','rtw','rtwdemos','roll_input_data.c'));
   copyfile(fullfile(matlabroot,'toolbox','rtw','rtwdemos','roll_input_data.h'));
   copyfile(fullfile(matlabroot,'toolbox','rtw','rtwdemos','roll_heading_mode.c'));
   copyfile(fullfile(matlabroot,'toolbox','rtw','rtwdemos','roll_heading_mode.h'));
   ```
2 Open model `rtwdemo_roll`. Save a copy of the model in the folder where you copied the external code files.
3 Open the **Embedded Coder** app. The **C Code** tab opens, which includes the Code Mappings editor.

Customize Entry-Point Functions

1 In the **C Code** tab, select **Code Interface > Individual Element Code Mappings**.
2 Click the **Functions** tab.
3 Customize the name of the step (execution) function. In the **Function Name** column, enter the name `roll_run`.
4 Customize arguments of the `step` function. Open the configuration dialog box for the `step` function by clicking the prototype hyperlink in the **Function Preview** column.
5 Select **Configure arguments for Step function prototype**.
6 To open a table that displays the default configurations for the arguments, click **Get Default**.
7 Customize the arguments:

- From the **C return argument** drop-down list, select `Ail_Cmd`.
- For each port, in the **C Identifier Name** field, remove the `arg_` prefix from their default names.
- For the `HDG_Mode` Inport, from the **C Type Qualifier** drop-down list, select `Pointer`. In the **C Identifier Name** field change the name to `HDG_Mode_Ptr`

8 Click **Apply** and verify that the function prototype reflects the changes.



9 Validate the changes by clicking **Validate**.

10 Click **OK**.

Generate and Verify Code

1 Generate code.

2 Verify the updates in the generated C file `rtwdemo_roll.c`. To find the updated `step` function (`roll_run`), use the **Search** field.

3 Select the `step` function to verify its prototype.

```
real32_T roll_run(real32_T Phi, real32_T Psi, real32_T Rate_FB, real32_T TAS,
                  boolean_T AP_Eng, boolean_T *HDG_Mode_Ptr, real32_T HDG_Ref,
                  real32_T Turn_Knob)
```

Parameters

Data Defaults

`Model Element Category` — Category of model data elements

character vector

Names a category of Simulink model data elements. The storage class that you set for a category applies to elements in that category throughout the model.

Model Element Category	Description
Inports	Root-level input ports of a model, such as Inport and In Bus Element blocks.
Outports	Root-level output ports of a model, such as Outport and Out Bus Element blocks.
Signals, states, and internal data	Data elements that are internal to the model, such as block output signals, discrete block states, data stores, and zero-crossing signals.
Shared local data stores	Data Store Memory blocks that have the block parameter **Share across model instances** set. These data stores are accessible only in the model where they are defined. The data store value is shared across instances of the model.
Global data stores	Data stores that are defined by a signal object in the base workspace or in a data dictionary. Multiple models in an application can use these data stores. To view and configure these data stores in the Code Mappings editor, click the `Refresh` link to the right of the category name. Clicking this link updates the model diagram.
Model parameter arguments	Parameters in the model workspace that you configure as model arguments. These parameters are exposed at the model block to enable each model instance to provide its own value. To specify a parameter as a model argument, select the **Model Data Editor** > **Parameters** > **Argument** check box.
Model parameters	Parameters that are defined within a model, such as parameters in the model workspace. Excludes model arguments.
External parameters	Parameters that you define as objects in the base workspace or in a data dictionary. Multiple models in an application can use these parameters. To view and configure these parameters in the Code Mappings editor, click the `Refresh` link to the right of the category name. Clicking this link updates the model diagram.
Constants	Constant-value block output and parameters that could not be inlined. These values are stored in variables for one of the following reasons. • The value is an array larger than the loop unrolling threshold. • The value address is needed in the code.

The Code Mappings editor presents valid storage class options for a given category. The options can include:

- Unspecified storage class (`Default`). The code generator places the code for the category of data elements in standard structures, such as `B_`, `ExtY_`, `ExtU_`, `DW_`, and `P_`. See “Data Structures in the Generated Code”.
- Relevant predefined storage classes, such as `ExportedGlobal`.
- Relevant storage classes in an available package, such as `ImportFromFile` .
- Storage class defined in an Embedded Coder Dictionary .

`Storage Class` — Code definition for model data elements

character vector

Definition (specification) that the code generator uses to determine properties, such as appearance and location, for code that it produces for model data elements. See “Choose Storage Class for Controlling Data Representation in Generated Code”.

Function Defaults

`Model Function Category` — Category of model functions

character vector

Names a category of Simulink model functions. The function customization template that you set for a category applies to functions in that category throughout the model.

Model Function Category	Description
Initialize/Terminate	Entry-point functions for initialization and termination
Execution	Entry-point functions for initiating execution and resets
Shared utility	Shared utility functions

`Function Customization Template` — Code definition for functions

character vector

Definition (specification) that the code generator uses to determine properties, such as appearance and location, for code that it produces for model functions. Templates are not available by default. You might need to define a function customization template in the Embedded Coder Dictionary.

Functions

`Source` — Type of entry-point function

character vector

Identifies the type of entry-point function. For rate-based models, this property provides the sample rate of `step` functions.

`Function Customization Template` — Code definition for function

character vector

Definition (specification) that the code generator uses to determine properties, such as appearance and location, for code that it produces for a model function.

`Function Name` — Name for a function

character vector

Name that the code generator gives a model function.

`Function Preview` — Preview of function prototype

character vector

Preview of the entry-point function prototype. To verify a prototype, review the prototype preview. To open a dialog box where you can customize the prototype, click the preview hyperlink. For more information, see “Configure Default Settings for Functions”.

Inports

`Source` — Name of root-level Inport block or bus element

character vector

Identifies a root Inport block or an element of an In Bus Element block (for example, `InBus1.signal1`) in the model. If the element resolves to a data object, the Code Mappings editor displays a resolve-to-signal-object icon to the right of the source name and resolves the configuration based on whether the storage class setting for the element is `Auto`. If the storage class is `Auto`, the data element assumes the code configuration that the data object specifies. The editor changes the display text in the **Storage Class** column to `From signal object:` followed by the name of the storage class of the data object. If the storage class is not `Auto`, the data element assumes the configuration that you specify in the Code Mappings editor.

`Storage Class` — Code definition for root inport

character vector

Definition that the code generator uses to determine properties, such as appearance and location, for code that it produces for the root inport. See “Choose Storage Class for Controlling Data Representation in Generated Code”.

Outports

`Source` — Name of root Outport block or bus element

character vector

Identifies a root-level Outport block or an element of an Out Bus Element block (for example, `OutBus1.signal1`) in the model. If the element resolves to a data object, the Code Mappings editor displays a resolve-to-signal-object icon to the right of the source name and resolves the configuration based on whether the storage class setting for the element is `Auto`. If the storage class is `Auto`, the data element assumes the code configuration that the data object specifies. The editor changes the display text in the **Storage Class** column to `From signal object:` followed by the name of the storage class of the data object. If the storage class is not `Auto`, the data element assumes the configuration that you specify in the Code Mappings editor.

`Storage Class` — Code definition for root outport

character vector

Definition that the code generator uses to determine properties, such as appearance and location, for code that it produces for the root outport. See “Choose Storage Class for Controlling Data Representation in Generated Code”.

Parameters

`Source` — Name of model parameter argument, model parameter, or external parameter

character vector

Identifies a parameter in the model. If the element resolves to a data object, the Code Mappings editor displays a resolve-to-parameter-object icon to the right of the source name and resolves the

configuration based on whether the storage class setting for the element is `Auto`. If the storage class is `Auto`, the data element assumes the code configuration that the data object specifies. The editor changes the display text in the **Storage Class** column to `From parameter object:` followed by the name of the storage class of the data object. If the storage class is not `Auto`, the data element assumes the configuration that you specify in the code mappings.

Types of parameter elements are listed in this table.

Type of Parameter Element	Description
Model parameter argument	Block parameter in the model workspace that you configure as a model argument. The parameter is exposed at the model block to enable each model instance to provide its own value. To specify a parameter as a model argument, select the **Model Data Editor** > **Parameters** > **Argument** check box.
Model parameter	Parameter that is defined within a model, such as a parameter in the model workspace. Excludes model arguments.
External parameter	Parameter that you define as an object in the base workspace or in a data dictionary. Multiple models in an application can use these parameters. This grouping of parameters appears in the editor only if the model uses such an element. To view and configure these parameters in the Code Mappings editor, click the `Refresh` link to the right of the category name. Clicking this link updates the model diagram.

`Storage Class` — Code definition for parameter
character vector

Definition that the code generator uses to determine properties, such as appearance and location, for code that it produces for the parameter. For external parameters, after you click the `Refresh` link to the right of the category name, the compiled storage class (for example, the storage class configured for an external parameter) appears on the right side of the **Storage Class** column. See “Choose Storage Class for Controlling Data Representation in Generated Code”.

Data Stores

`Source` — Name of local data store, shared local data store, or global data store
character vector

Identifies a data store in the model. If the element resolves to a data object, the Code Mappings editor displays a resolve-to-signal-object icon to the right of the source name and resolves the configuration based on whether the storage class setting for the element is `Auto`. If the storage class is `Auto`, the data element assumes the code configuration that the data object specifies. The editor changes the display text in the **Storage Class** column to `From signal object:` followed by the name of the storage class of the data object. If the storage class is not `Auto`, the data element assumes the configuration that you specify in the code mappings.

Types of data store elements are listed in this table.

Type of Data Store Element	Description
Local data store	Data store that is accessible from anywhere in the model hierarchy that is at or below the level at which you define the data store. You can define a local data store graphically in a model by including a Data Store Memory block or by creating a signal object (synthesized data store) in the model workspace.
Shared local data store	Data Store Memory block that has the block parameter **Share across model instances** set. These data stores are accessible only in the model where they are defined. The data store value is shared across instances of the model. This grouping of data stores appears in the editor only if such an element exists in the model.
Global data store	Data store that is defined by a signal object in the base workspace or in a data dictionary. Multiple models in an application can use these data stores. These data stores are not configurable in the code mappings. After you click the refresh button, they appear in the Code Mappings editor in a read-only state for viewing or accounting purposes. This grouping of data stores appears in the editor only if the model uses such an element. To view and configure these data stores in the Code Mappings editor, click the `Refresh` link to the right of the category name. Clicking this link updates the model diagram.

Names of local and shared local data stores appear in the format *`block-name`*`:` *`data-store-name`*.

Depending on how the data store element is represented and configured in the model, local and shared local data stores can resolve to a signal object in the model workspace, based workspace, or a data dictionary. Global data stores resolve to a signal object in the base workspace or a data dictionary.

`Storage Class` — Code definition for data store

character vector

Definition that the code generator uses to determine properties, such as appearance and location, for code that it produces for the data store. For global data stores, after you click the `Refresh` link to the right of the category name, the compiled storage class (for example, the storage class configured for a global data store) appears on the right side of the **Storage Class** column. See “Choose Storage Class for Controlling Data Representation in Generated Code”.

`Path` — Path to data store in model

character vector

Link that you can click to highlight the data store in the model diagram.

Signals/States

`Source` — Name of signal or state

character vector

Identifies a signal line or state in the model. If the element resolves to a data object, the Code Mappings editor displays a resolve -to-signal-object icon to the right of the source name and resolves the configuration based on whether the storage class setting for the element is `Auto`. If the storage

class is `Auto`, the data element assumes the code configuration that the data object specifies. The editor changes the display text in the **Storage Class** column to `From signal object:` followed by the name of the storage class of the data object. If the storage class is not `Auto`, the data element assumes the configuration that you specify in the Code Mappings editor.

The Code Mappings editor lists:

- Named signals and states by using the data element name
- Unnamed signals by using the format *`source-block`*`:` *`port-number`*
- States used in multiple blocks by using the format *`block-name`*`:` *`state-name`*

To configure an individual signal line in the Code Mappings editor for a model, first you must add the signal to the mappings. See “Configure Signal Data for C Code Generation”.

`Storage Class` — Code definition for signal
character vector

Definition that the code generator uses to determine properties, such as appearance and location, for code that it produces for the signal line or state. See “Choose Storage Class for Controlling Data Representation in Generated Code”.

`Path` — Path to signal line or state in model
character vector

Link that you can click to highlight the signal line or block that uses the state in the model diagram.

See Also

Embedded Coder Dictionary | `coder.mapping.api.CodeMapping`

Topics
“C Code Generation Configuration for Model Interface Elements”
“Choose Data Configuration Approach”
“Choose Storage Class for Controlling Data Representation in Generated Code”
“Configure Default C Code Generation for Categories of Data Elements and Functions”
“Configure C Code Generation for Model Entry-Point Functions”

Introduced in R2018a

Code Mappings – C++ Editor

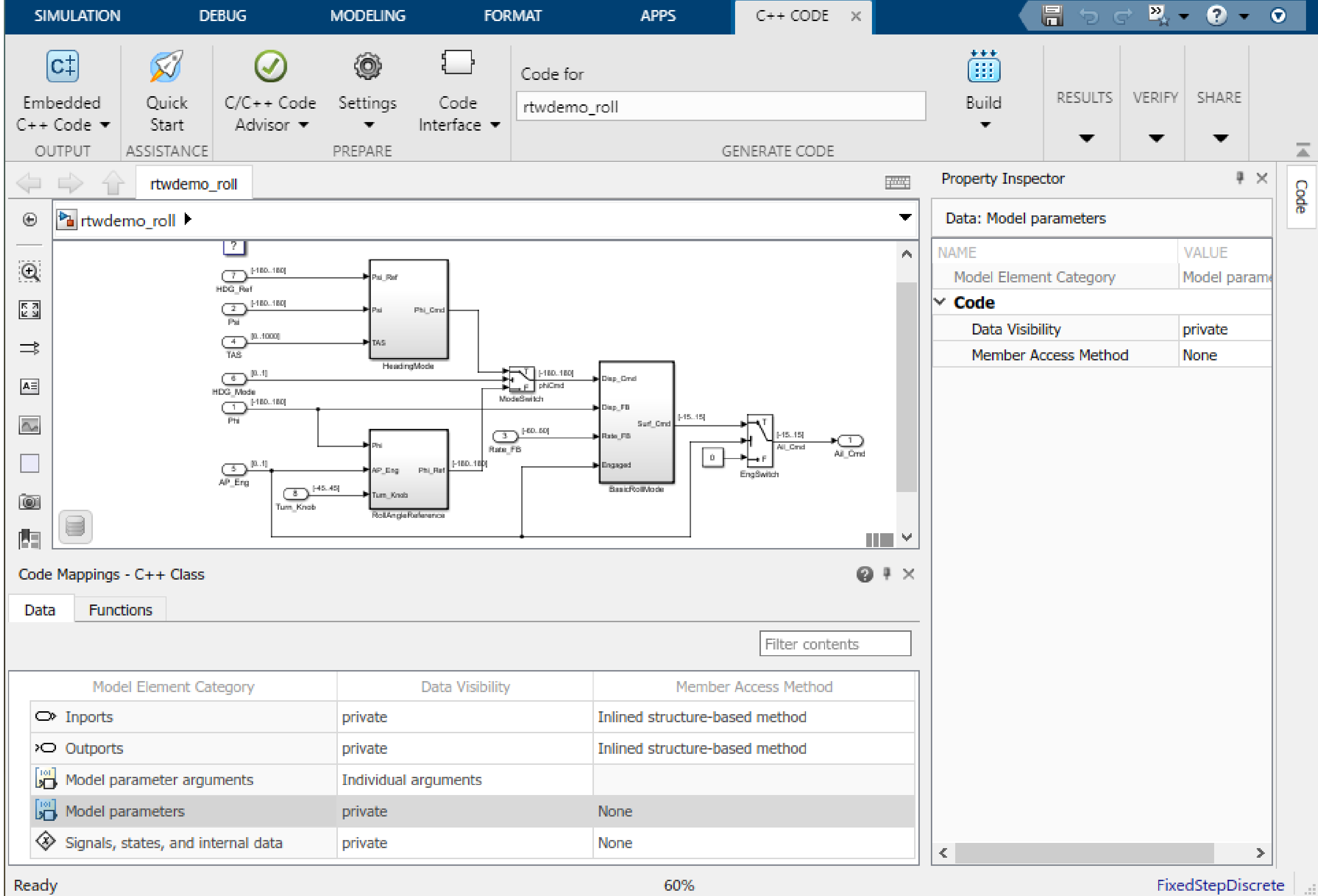
Configure how model elements and functions appear in generated C++ code

Description

The Code Mappings editor is a graphical interface used to configure how Simulink model data elements and functions appear in generated C++ code.

To configure Simulink model data elements and functions for C++ code generation, use the tabs in the Code Mappings editor:

- **Data**
- **Functions**



Open the Code Mappings – C++ Editor

Open the Embedded Coder app. Verify the **Output** is set to **Embedded C++ Code**. On the **C++ Code** tab, click **Code Interface** and select **Code Mappings**.

Examples

Configure Data Elements

You can use the Code Mappings editor to customize the data visibility and method access of model elements in the generated C++ class interface. This example uses the model `rtwdemo_roll` to show how to configure the data visibility and method access for the inports in this model.

Set Up the Environment

1. Open model `rtwdemo_roll`.
2. Open the Embedded Coder app. In the Apps gallery, click **Embedded Coder**.
3. Set the language to C++. On the **C++ Code** tab, click **Output** and select **Embedded C++ Code**.
4. Open the Code Mappings editor. From the tab, click **Code Interface** and select **Code Mappings**.

Customize the Data Visibility and Method Access of Data Elements

1. In the Code Mappings editor, click the **Data** tab.
2. Customize the data visibility. For the **Inports** category, in the **Data Visibility** column, select `public`.
3. Customize the method access. For the **Inports** category, in the **Member Access Method** column, select `method`.

Generate and Verify Code

1. Generate code.
2. Verify the generated C++ code for the inports. In the **Code** view, open `rtwdemo_roll.cpp` file and search for the inports in the model.

Configure Functions

You can use the Code Mappings editor to customize the names of entry-point functions and the names and arguments of base-rate periodic or Simulink Functions. This examples uses the model `rtwdemo_roll` to show how to configure the name and arguments of a base-rate periodic function.

Set Up the Environment

1. Open model `rtwdemo_roll`.
2. Open the Embedded Coder app. In the Apps gallery, click **Embedded Coder**.
3. Set the language to C++. On the **C++ Code** tab, click **Output** and select **Embedded C++ Code**.
4. Open the Code Mappings editor. From the tab, click **Code Interface** and select **Code Mappings**.

Customize Function Name and Arguments

1. In the Code Mappings editor, click the **Functions** tab.
2. Customize the name of the periodic function. In the **Function Name** column, enter the name `roll_run`.
3. Customize the arguments of the periodic function. In the **Function Preview** column, click the prototype hyperlink. A configuration dialog box opens.
4. In the dialog box, select **Configure arguments for Step function prototype** and click **Get Default**. A table that displays the arguments opens.

5 Customize the arguments:

   - From the **C++ return argument** drop-down list, select `Ail_Cmd`.
   - For each port, in the **C++ Identifier Name** field, remove the `arg_` prefix from their default names.
   - For the `HDG_Mode` inport, from the **C++ Type Qualifier** drop-down list, select `Pointer`. In the **C++ Identifier Name** field change the name to `HDG_Mode_Ptr`

6 Click **Apply**. Visually verify that the function prototype reflects the changes. Click **OK** to exit.

Generate and Verify Code

1 Generate code.
2 Verify the updated method names and arguments in the generated C++ code. In the **Code** view, in the open `rtwdemo_roll.cpp` file, search for the base-rate periodic function, `roll_run`.

Parameters

Data

`Model Element Category` — Category of model data elements

character vector

Each category describes a type of Simulink model data element. The data visibility and method access set for a category applies to the data elements in that category for the model.

Model Element Category	Description
Inports	Root-level data input ports of a model, such as Inport and In Bus Element blocks.
Outports	Root-level data output ports of a model, such as Outport and Out Bus Element blocks.
Model parameter arguments	Workspace variables that may appear as per instance (nonstatic) class data members.
Model parameters	Workspace variables that are shared across instances of the model class and are generated as static class data members.
Signals, states, and internal data	Data elements that are internal to the model, such as block output signals, discrete block states, data stores, and zero-crossing signals.

`Data Visibility` — Determines the access specifier for the generated class members

character vector

The data visibility determines if data elements appear in generated code as public or private. For model parameter arguments, this parameter can also specify that data elements are generated outside the class and passed as individual arguments.

`Member Access Method` — Determines the generated get and set methods for class members

character vector

The access determines how the generated code provides access to the class member data.

Functions

`Source` — Functions in a model
character vector

The functions in a model that generate entry-point methods in a C++ class interface. These functions include:

- Initialize Functions
- Terminate Functions
- Periodic Functions
- Partition Functions
- Exported Functions
- Reset Functions
- Simulink Functions

`Method Name` — Name of generated class method
character vector

Name for the generated class method.

`Method Preview` — Preview of method prototype
character vector

Preview of the entry-point method prototype. To customize the prototype, click the preview hyperlink and configure the method in the opened dialog box.

See Also

Introduced in R2021a

Code Replacement Tool

Create, modify, and validate content of code replacement libraries

Description

The Code Replacement Tool is a graphical interface that you can use to create and manage custom code replacement libraries. You can create, import, manipulate, and validate the code replacement tables in a library. The tool also generates the customization file to register a code replacement library with the code generator. If you specify a table name when you open the tool, the tool displays only the contents of that table.

The tool display consists of three panes that show table and table entry information:

- Left pane lists code replacement tables.
- Middle pane lists available tables or, if you select a table in the left pane, the table entries that are in that table.
- Right pane lists table or table entry details. If you select a table, the right pane shows table properties: the table name, which you can modify, the table version, and the total number of entries in the table. If select a table entry, the right pane shows mapping and build information for that entry.

Open the Code Replacement Tool

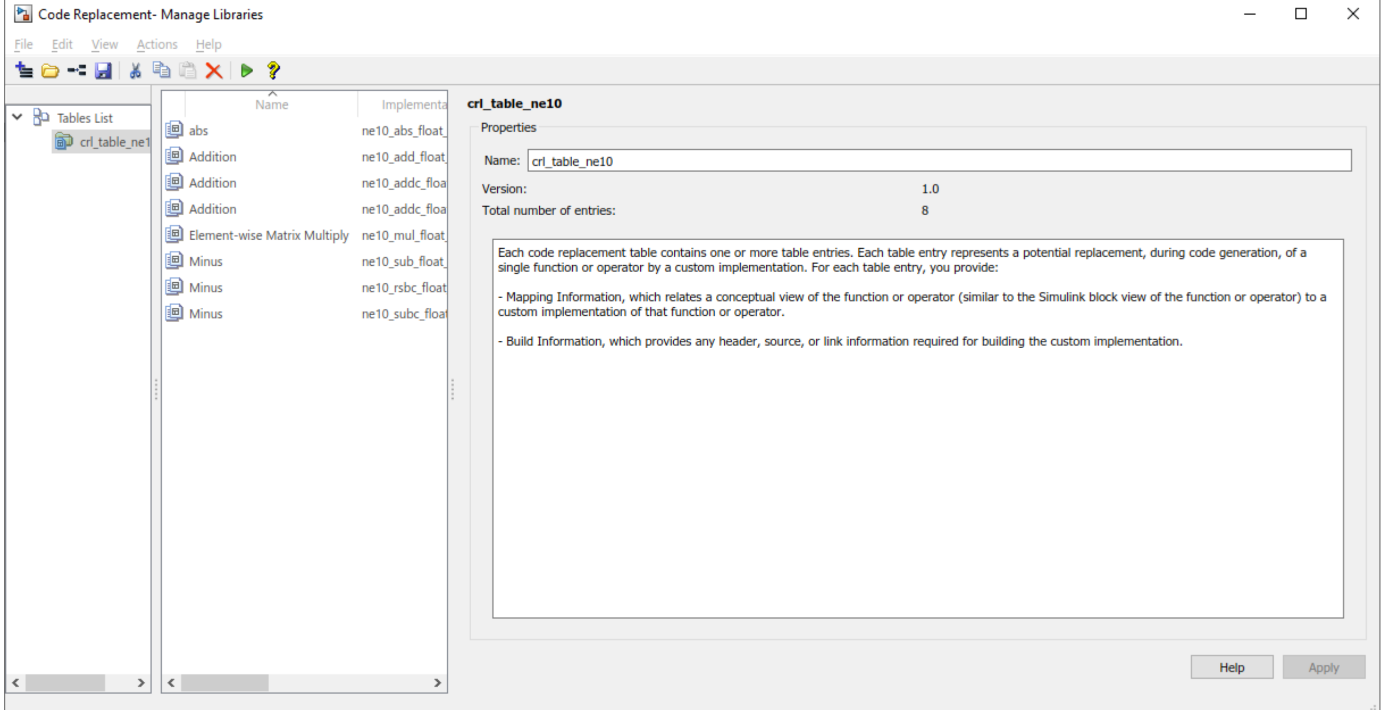
At the command prompt, type `crtool`.

Examples

Open an Existing Table in the Tool

This example shows how to open a code replacement table, `crl_table_ne10`, in the Code Replacement Tool.

```
crtool('crl_table_ne10')
```



- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Quick Start Code Replacement Library Development - Simulink®”

Parameters

Entry Summary Information (Center Pane)

Name — Name of table entry (read-only)
character vector

Conceptual name of the function or operation being replaced. Can name a math operation, function, BLAS operation, CBLAS operation, net slope fixed-point operation, semaphore or mutex entry, or customization entry.

`Implementation` — Name of replacement function
character vector

Name of the implementation (replacement) function.

`NumIn` — Number of input arguments (read-only)
scalar integer

Number of input arguments.

In*n*Type — Data type of conceptual input argument
character vector

Data type of a conceptual input argument.

`OutnType` — Data type of conceptual output argument
character vector

Data type of a conceptual output argument.

`Priority` — Entry match priority
100 (default) | integer, ranging from 0 to 100

The entry match priority, relative to other entries of the same name and to the conceptual argument list within the selected code replacement library. The priority can range from 0 to 100, with 0 being the highest priority. The default is 100. If the library provides two implementations for a function or operator, the implementation with the higher priority shadows the one with the lower priority.

Entry Mapping Information (Right Pane)

`Function/Operation` — Name of table entry
character vector

Conceptual name of the function or operation being replaced. Can name a math operation, function, BLAS operation, CBLAS operation, net slope fixed-point operation, semaphore or mutex entry, or customization entry.

`Algorithm` — Computation or approximation algorithm
unspecified (default) | options vary depending on function or operation

Computation or approximation algorithm configured for a function or operation being replaced. For example, you can configure:

- The Reciprocal Sqrt block to use the `Newton-Raphson` computation method.
- The Trigonometric Function block, with **Function** set to `sin`, `cos`, or `sincos`, to use the approximation method `CORDIC`.
- An addition or subtraction operation, to use the cast-before-operation or cast-after-operation algorithm.

`Conceptual arguments` — Conceptual argument names
yn | un

Names of input and output arguments of function or operation being replaced. Conceptual arguments observe naming conventions (`y1`, `u1`, `u2`, ...) and data types familiar to the code generator.

`Data type (conceptual)` — Conceptual argument data type
double (default) | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | boolean | void | logical | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0)

Data type of a selected input or output argument of the function or operation being replaced. Conceptual arguments observe data types familiar to the code generator.

`Complex (conceptual)` — Conceptual argument complexity
cleared (default) | selected

Whether the selected input or output argument of the function or operation being replaced is real or complex.

`Argument type` — Conceptual argument type
scalar (default) | matrix

Whether the selected input or output argument of the function or operation being replaced is a scalar value or a matrix. If you select `Matrix`, parameters for specifying range dimensions, and for replacement of MATLAB code, array layout appear.

`Lower range` — Lower range of matrix dimensions
[2 2] (default)

Vector that specifies the lower range of the matrix dimensions.

`Upper range` — Upper range of matrix dimensions
[2 2] (default)

Vector that specifies the upper range of the matrix dimensions.

`Array layout supported by entry` — Layout for array storage
Column-major (default) | Row-major | Column-and-Row

Order in which array elements are stored in memory. Row-major layout can improve performance for certain algorithms and ease integration with external code or data that uses the row-major layout.

`Make conceptual and implementation argument types the same` — Data type consistency
selected (default) | cleared

Whether you want the data types for your implementation arguments to be the same as the conceptual argument types. For example, most ANSI-C functions operate on and return `double` data. Clear the check box if want to map the conceptual representation of a function or operation to an implementation representation that specifies an argument and return value. For example, clear the check box to map the conceptual representation of the function `sin` to an implementation representation that specifies an argument and return value of type `single` (`single sin`(`single`)).

`Name` — Name of replacement function
character vector

Name of the replacement function.

`C++ namespace` — Namespace of replacement function
character vector

Namespace of the replacement function.

`Function returns void` — Function returns void
selected (default) | cleared

Whether your implementation function returns `void`.

`Function arguments` — Replacement argument names
yn | un

Names of input and output arguments of your replacement function.

`Data type (replacement)` — Replacement argument data type
double (default) | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | boolean | void | integer | size_t | long | ulong | long long | ulong long | char

Data type of a selected input or output argument of your replacement function.

`I/O type` — Replacement argument I/O type
OUTPUT | INPUT

Whether a selected argument of your replacement function is an input or output argument.

`Const` — Const replacement argument
cleared (default) | selected

Whether to apply the `const` type qualifier to a selected argument of your replacement function.

`Pointer` — Pointer replacement argument
cleared (default) | selected

Whether a selected argument of your replacement function is a pointer.

`Complex (replacement)` — Replacement argument complexity
cleared (default) | selected

Whether the selected input or output argument of the replacement function is real or complex.

`Integer saturation mode` — Saturation mode
unspecified Saturation (default) | wrap on overflow | saturate on overflow

Saturation mode supported by the replacement function.

`Rounding modes` — Rounding modes
unspecified rounding (default) | floor | ceil | zero | nearest | MATLAB nearest | simplest | conv

Rounding modes supported by the replacement function.

`Allow expressions as inputs` — Expressions as inputs
selected (default) | cleared

Whether your replacement function accepts expression inputs. If you select the parameter, the code generator integrates an expression input into the generated code rather than inserting a temporary variable in place of the expression input.

`Function modifies internal or global state` — State modification
cleared (default) | selected

Whether your replacement function modifies variables representing internal or global state.

Entry Build Information (Right Pane)

`Implementation header file` — Header file for replacement function
character vector

Header file for the replacement function (for example, `my_rep_func.h`).

`Implementation source file` — Source file for replacement function
character vector

Source file for the replacement function (for example, `my_rep_func.c`).

`Additional header files/include paths` — Names and paths of additional header files
character vector

Names and paths of additional header files to include for the replacement function (for example, `support_files.h` and `matlab\customization\mylib\include`).

`Additional source files/ paths` — Names and paths of additional source files
character vector

Names and paths of additional source files to include for the replacement function (for example, `support_files.c` and `matlab\customization\mylib\src`).

`Additional object files/ paths` — Names and paths of link object files
character vector

Names and paths of link object files to use for the replacement function (for example, `support_files.o` and `matlab\customization\mylib\bin`).

`Additional link flags` — Link flags to use
character vector

Link flags to use for the replacement function (for example, `-MD -Gy`).

`Additional compile flags` — Compile flags to use
character vector

Compile flags to use for the replacement function (for example, `-Zi -Wall`).

`Copy files to build directory` — Copy files to build folder
cleared (default) | selected

Whether the code generator copies files from external folders to the build folder before starting the build process.

Programmatic Use

`crtool(table)` opens the Code Replacement Tool and displays the contents of `table`, where `table` is a character vector that names a MATLAB file that defines code replacement tables. The file must be in the current folder or on the MATLAB path.

See Also

Topics
"What Is Code Replacement?"
"What Is Code Replacement Customization?"
"Quick Start Code Replacement Library Development - Simulink®"

Introduced in R2014b

Code Replacement Viewer

Explore content of code replacement libraries

Description

The Code Replacement Viewer displays the content of code replacement libraries and tables. You can use this tool to explore and choose a code replacement library or to view a predefined code replacement table. If you develop a custom code replacement library, you can use this viewer to verify table entries for the following properties:

- Argument order is correct.
- Conceptual argument names match code generator naming conventions.
- Implementation argument names are correct.
- Header or source file specification is not missing.
- I/O types are correct.
- Relative priority of entries is correct (highest priority is 0, and lowest priority is 100).
- Saturation or rounding mode specifications are not missing.

If you specify a library name when you open the viewer, the viewer displays the code replacement tables for that library. If you specify a table name when you open the viewer, the viewer displays the function and operator code replacement entries for that table. The viewer can only display code replacement tables that are defined. For more information on creating code replacement tables, see "Define Code Replacement Library Optimizations".

Abbreviated Entry Information

In the middle pane, the viewer displays entries that are in the selected code replacement table, along with abbreviated information for each entry.

Field	Description
Name	Name or identifier of the function or operator being replaced (for example, `cos` or `RTW_OP_ADD`).
Implementation	Name of the implementation function, which can match or differ from **Name**.
NumIn	Number of input arguments.
In1Type	Data type of the first conceptual input argument.
In2Type	Data type of the second conceptual input argument.
OutType	Data type of the conceptual output argument.
Priority	The entry's match priority, relative to other entries of the same name and to the conceptual argument list within the selected code replacement library. The priority can range from 0 to 100, with 0 being the highest priority. The default is 100. If the library provides two implementations for a function or operator, the implementation with the higher priority shadows the one with the lower priority.

Field	Description
UsageCount	Not used.

Detailed Entry Information

In the middle pane, when you select an entry, the viewer displays entry details.

Field	Description
Description	Text description of the table entry (can be empty).
Key	Name or identifier of the function or operator being replaced (for example, `cos` or `RTW_OP_ADD`), and the number of conceptual input arguments.
Implementation	Name of the implementation function, and the number of implementation input arguments.
Implementation type	Type of implementation: `FCN_IMPL_FUNCT` for function or `FCN_IMPL_MACRO` for macro.

Saturation mode	Saturation mode that the implementation function supports. One of:
	`RTW_SATURATE_ON_OVERFLOW`
	`RTW_WRAP_ON_OVERFLOW`
	`RTW_SATURATE_UNSPECIFIED`

Rounding modes	Rounding modes that the implementation function supports. One or more of:
	`RTW_ROUND_FLOOR`
	`RTW_ROUND_CEILING`
	`RTW_ROUND_ZERO`
	`RTW_ROUND_NEAREST`
	`RTW_ROUND_NEAREST_ML`
	`RTW_ROUND_SIMPLEST`
	`RTW_ROUND_CONV`
	`RTW_ROUND_UNSPECIFIED`

GenCallback file	Not used.
Implementation header	Name of the header file that declares the implementation function.
Implementation source	Name of the implementation source file.
Priority	The entry's match priority, relative to other entries of the same name and to the conceptual argument list within the selected code replacement library. The priority can range from 0 to 100, with 0 being the highest priority. The default is 100. If the library provides two implementations for a function or operator, the implementation with the higher priority shadows the one with the lower priority.
Total Usage Count	Not used.
Entry class	Class from which the current table entry is instantiated.
Conceptual arguments	Name, I/O type (`RTW_IO_OUTPUT` or `RTW_IO_INPUT`), and data type for each conceptual argument.
Implementation	Name, I/O type (`RTW_IO_OUTPUT` or `RTW_IO_INPUT`), data type, and alignment requirement for each implementation argument.

Fixed-Point Entry Information

When you select an operator entry that specifies net slope fixed-point parameters, the viewer displays fixed-point information.

Field	Description
Net slope adjustment factor F	Slope adjustment factor (F) part of the net slope factor, $F2^E$, for net slope table entries. You use this factor with fixed-point multiplication and division replacement to map a range of slope and bias values to a replacement function.
Net fixed exponent E	Fixed exponent (E) part of the net slope factor, $F2^E$, for net slope table entries. You use this fixed exponent with fixed-point multiplication and division replacement to map a range of slope and bias values to a replacement function.
Slopes must be the same	Indicates whether code replacement request processing must check that the slopes on arguments (input and output) are equal. You use this information with fixed-point addition and subtraction replacement to disregard specific slope and bias values, and map relative slope and bias values to a replacement function.
Must have zero net bias	Indicates whether code replacement request processing must check that the net bias on arguments is zero. You use this information with fixed-point addition and subtraction replacement to disregard specific slope and bias values, and map relative slope and bias values to a replacement function.

Open the Code Replacement Viewer

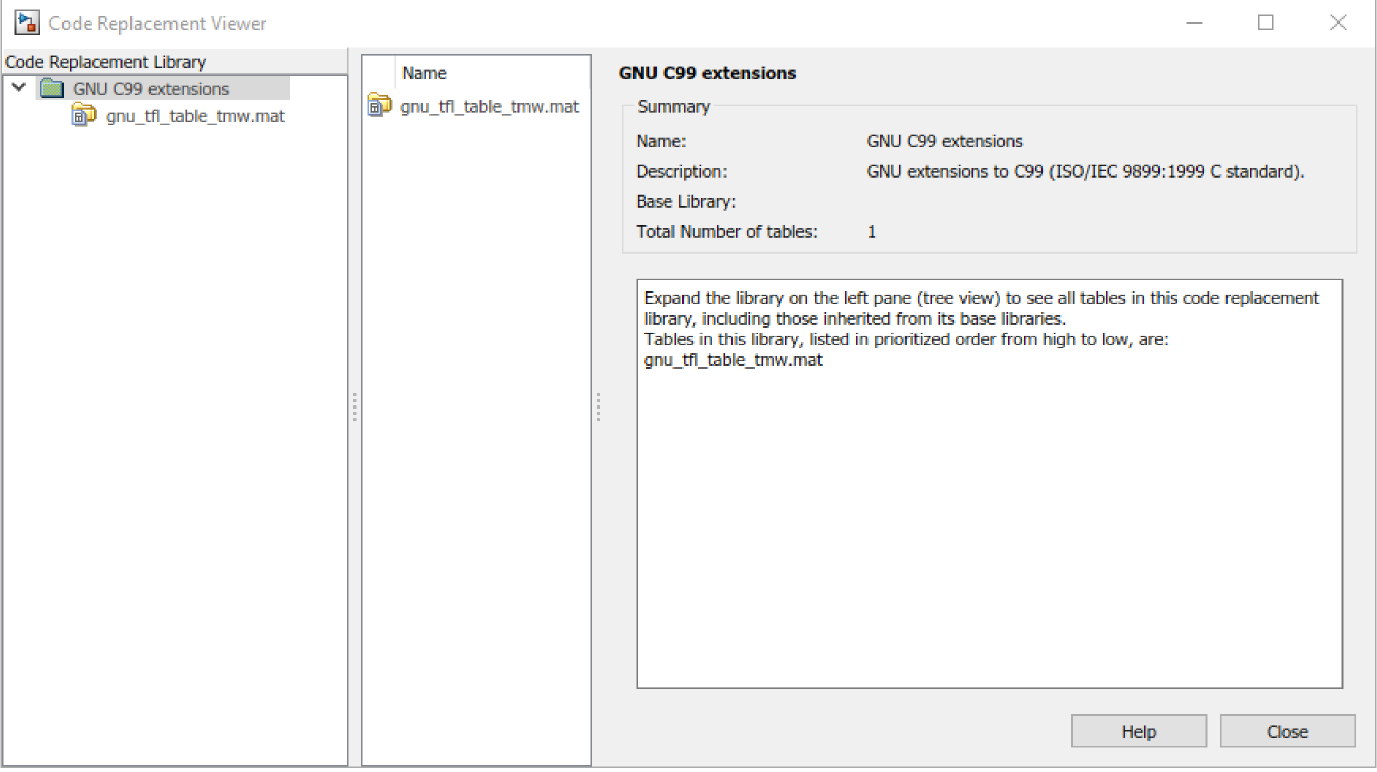
Open from the MATLAB command prompt using `crviewer`.

Examples

Display Contents of Code Replacement Library

This example opens the registered code replacement library `GNU C99 extensions`.

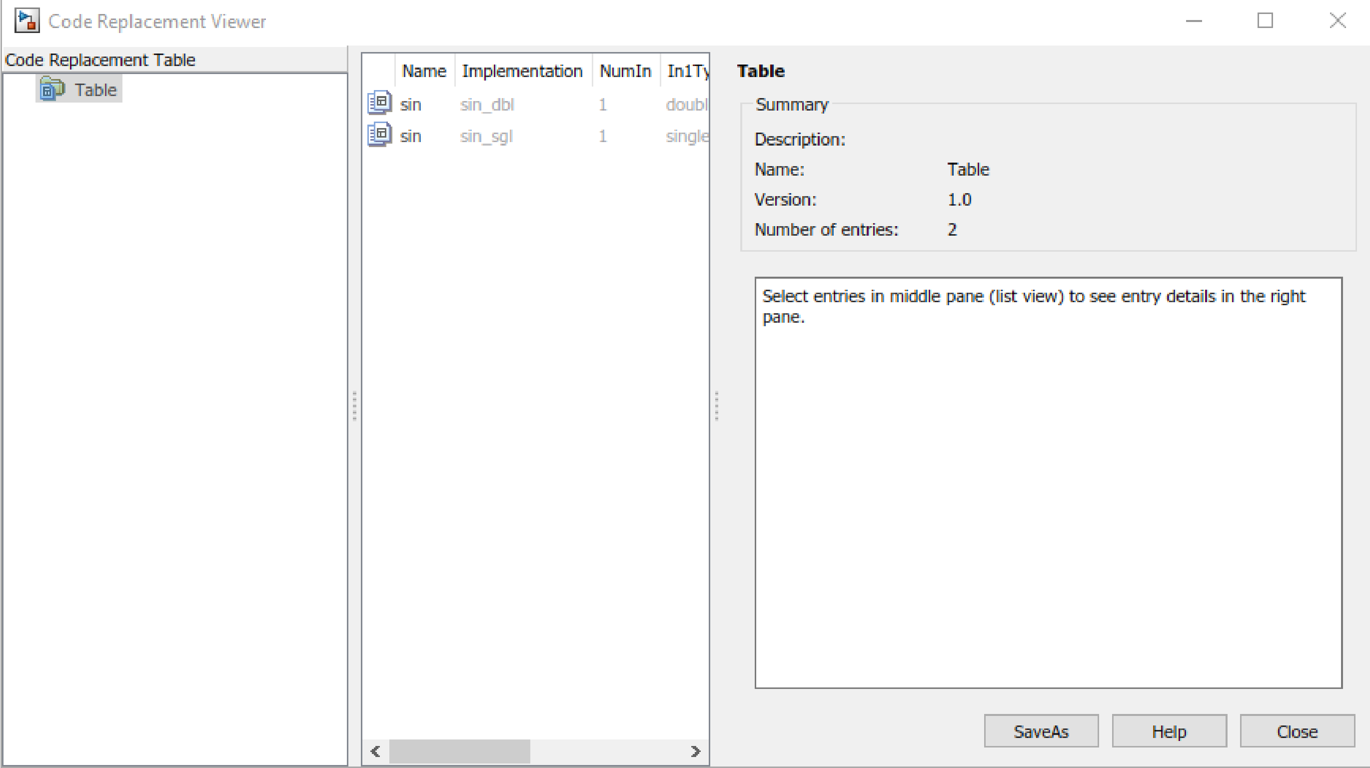
```
crviewer('GNU C99 extensions')
```



Display Contents of Code Replacement Table

This example opens a predefined code replacement table `crl_table_sinfcn`. To learn how to create this example table, see “Define Code Replacement Library Optimizations”.

```
crviewer(crl_table_sinfcn)
```



- “Choose a Code Replacement Library”
- “Verify Code Replacement Library”

Programmatic Use

`crviewer('library')` opens the Code Replacement Viewer and displays the contents of `library`, where `library` is a character vector that names a registered code replacement library.

`crviewer(table)` opens the Code Replacement Viewer and displays the contents of a predefined `table`, where `table` is a MATLAB file that defines code replacement tables. The table must be user predefined and the file must be in the current folder or on the MATLAB path.

See Also

Topics
“Choose a Code Replacement Library”
“Verify Code Replacement Library”
“What Is Code Replacement?”
“What Is Code Replacement Customization?”
“Code Replacement Libraries”
“Code Replacement Terminology”

Introduced in R2014b

C/C++ Functions That Support Symbolic Dimensions for Simulink Function Blocks

ssSetSymbolicDimsSupport

Specify whether an S-function supports symbolic dimensions

Languages

C, C++

Syntax

```
void ssSetSymbolicDimsSupport(SimStruct *S, const boolean_T val)
```

Arguments

`S`

SimStruct representing an S-Function block.

`val`

Boolean value corresponding to whether the S-Function block supports symbolic dimensions.

Returns

This function does not return a value.

Example

Call this function from inside the `mdlInitializeSizes` function. For an example of how to use this function to configure an S-function that supports forward and backward propagation of symbolic dimensions, see “Configure Dimension Variants for S-Function Blocks”

Introduced in R2016a

mdlSetInputPortSymbolicDimensions

Specify symbolic dimensions of an input port and how those dimension propagate forward

Languages

C, C++

Syntax

```
#if defined(MATLAB_MEX_FILE)
#define MDL_SET_INPUT_PORT_SYMBOLIC_DIMENSIONS

static void mdlSetInputPortSymbolicDimensions(SimStruct *S, int_T portIndex,
 SymbDimsId symbDimsId)

{
}
#endif
```

Arguments

`S`

SimStruct representing an S-Function block.

`portIndex`

Index of an input port.

`symbDimsId`

Unique integer value corresponding to a symbolic dimension specification.

Returns

This function does not return a value.

Example

For an example of how to use this function to configure an S-function that supports forward and backward propagation of symbolic dimensions, see “Configure Dimension Variants for S-Function Blocks”.

Introduced in R2016a

mdlSetOutputPortSymbolicDimensions

Specify symbolic dimensions of an output port and how those dimension propagate backward

Languages

C, C++

Syntax

```
#if defined(MATLAB_MEX_FILE)
#define MDL_SET_OUTPUT_PORT_SYMBOLIC_DIMENSIONS

static void mdlSetOutputPortSymbolicDimensions(SimStruct *S, int_T portIndex,
 SymbDimsId symbDimsId)

{
}
#endif
```

Arguments

`S`

SimStruct representing an S-Function block.

`portIndex`

Index of an output port.

`symbDimsId`

Unique integer value corresponding to a symbolic dimension specification.

Returns

This function does not return a value.

Example

Call this function from inside the `mdlInitializeSizes` function. For an example of how to use this function to configure an S-function that supports forward and backward propagation of symbolic dimensions, see “Configure Dimension Variants for S-Function Blocks”.

Introduced in R2016a

ssRegisterSymbolicDimsExpr

Create SymbDimsId from expression string (aExpr)

Languages

C, C++

Syntax

```
SymbDimsId ssRegisterSymbolicDimsExpr(SimStruct *S, const char_T* aExpr)
```

Arguments

S

SimStruct representing an S-Function block.

aExpr

Expression string that forms a valid syntax in C.

Returns

A unique SymbDimsId (integer value) that represents a symbolic dimension specification.

Example

This example creates a SymbDimsId for the expression string [F / C , D * (B-3)].

```
const SymbDimsId outputDimsId =
       ssRegisterSymbolicDimsExpr(S, "[ F / C , D * (B-3)]");
```

Introduced in R2016a

ssRegisterSymbolicDims

Create `SymbDimsId` from array of `SymDimsId`s

Languages

C, C++

Syntax

```
SymbDimsId ssRegisterSymbolicDims(SimStruct *S, const SymbDimsId* aDimsVec,
 const size_t aNumDims)
```

Arguments

`S`

SimStruct representing an S-Function block.

`aDimsVec`

Array of `SymDimsId`s

`aNumDims`

Size of `SymDimsId` array

Returns

A unique `SymbDimsId` (integer value) that represents a symbolic dimension specification.

Example

For an example of how to use this function to configure an S-function that supports forward and backward propagation of symbolic dimensions, see “Configure Dimension Variants for S-Function Blocks”.

Introduced in R2016a

ssRegisterSymbolicDimsString

Create `SymbDimsId` from identifier string

Languages

C, C++

Syntax

```
SymbDimsId ssRegisterSymbolicDimsString(SimStruct *S, const char_T* aString)
```

Arguments

`S`

SimStruct representing an S-Function block.

`aString`

Identifier string

Returns

A unique `SymbDimsId` (integer value) that represents a symbolic dimension specification.

Example

This example creates a `SymbDimsId` for the string `"B"`.

```
const SymbDimsId symbolId = ssRegisterSymbolicDimsString(S, "B");
```

Introduced in R2016a

ssRegisterSymbolicDimsIntValue

Create `SymbDimsId` from integer value

Languages

C, C++

Syntax

```
SymbDimsId ssRegisterSymbolicDimsIntValue(SimStruct *S, const int_T aIntValue)
```

Arguments

`S`

SimStruct representing an S-Function block.

`aIntValue`

Dimensions value

Returns

A unique `SymbDimsId` (integer value) that represents a symbolic dimension specification.

Example

This example creates a `SymbDimsId` for the integer `2`.

```
const SymbDimsId symbolId = ssRegisterSymbolicDimsIntValue(S, 2);
```

Introduced in R2016a

ssRegisterSymbolicDimsPlus

Create `SymbDimsId` by adding two symbolic dimensions

Languages

C, C++

Syntax

```
SymbDimsId ssRegisterSymbolicDimsPlus(SimStruct *S, const SymbDimsId aLHS,
 const SymbDimsId aRHS)
```

Arguments

`S`

SimStruct representing an S-Function block.

`aLHS`

Unique integer value corresponding to a symbolic dimension specification.

aRHS

Unique integer value corresponding to a symbolic dimension specification.

Returns

A unique `SymbDimsId` (integer value) that represents a symbolic dimension specification.

Example

This example shows how to add the `SymbDimsId` `symbDims` to `symbolId`, and then sets the result equal to a new `SymbDimsId` called `outputDimsId`.

```
const SymbDimsId outputDimsId =
        ssRegisterSymbolicDimsPlus(S, symbDimsId, symbolId);
```

Introduced in R2016a

ssRegisterSymbolicDimsMinus

Create `SymbDimsId` by subtracting two symbolic dimensions

Languages

C, C++

Syntax

```
SymbDimsId ssRegisterSymbolicDimsMinus(SimStruct *S, const SymbDimsId aLHS,
 const SymbDimsId aRHS)
```

Arguments

`S`

SimStruct representing an S-Function block.

`aLHS`

Unique integer value corresponding to a symbolic dimension specification.

aRHS

Unique integer value corresponding to a symbolic dimension specification.

Returns

A unique `SymbDimsId` (integer value) that represents a symbolic dimension specification.

Example

For an example of how to use this function to configure an S-function that supports forward propagation of symbolic dimensions, see “Configure Dimension Variants for S-Function Blocks”.

Introduced in R2016a

ssRegisterSymbolicDimsMultiply

Create `SymbDimsId` by multiplying two symbolic dimensions

Languages

C, C++

Syntax

```
SymbDimsId ssRegisterSymbolicDimsMultiply(SimStruct *S, const SymbDimsId aLHS,
 const SymbDimsId aRHS)
```

Arguments

`S`

SimStruct representing an S-Function block.

`aLHS`

Unique integer value corresponding to a symbolic dimension specification.

aRHS

Unique integer value corresponding to a symbolic dimension specification.

Returns

A unique `SymbDimsId` (integer value) that represents a symbolic dimension specification.

Example

This example shows how to multiply the `SymbDimsId`s `symbDimsId` and `symbolId`. It sets the result equal to a new `SymbDimsId` called `outputDimsId`.

```
const SymbDimsId outputDimsId =
       ssRegisterSymbolicDimsMultiply(S, symbDimsId, symbolId);
```

Introduced in R2016a

ssRegisterSymbolicDimsDivide

Create SymbDimsId by dividing two symbolic dimensions

Languages

C, C++

Syntax

```
SymbDimsId ssRegisterSymbolicDimsDivide(SimStruct *S, const SymbDimsId aLHS,
  const SymbDimsId aRHS)
```

Arguments

S

SimStruct representing an S-Function block.

aLHS

Unique integer value corresponding to a symbolic dimension specification.

aRHS

Unique integer value corresponding to a symbolic dimension specification.

Returns

A unique SymbDimsId (integer value) that represents a symbolic dimension specification.

Example

This example shows how to divide the SymbDimsId symbDimsId by symbolId. It sets the result equal to a new SymbDimsId called outputDimsId.

```
const SymbDimsId outputDimsId =
        ssRegisterSymbolicDimsDivide(S, symbDimsId, symbolId);
```

Introduced in R2016a

ssGetNumSymbolicDims

Get the number of dimensions for `SymbDimsId`

Languages

C, C++

Syntax

```
size_t  ssGetNumSymbolicDims(SimStruct *S, const SymbDimsId aSymbDimsId)
```

Arguments

`S`

SimStruct representing an S-Function block.

`aSymbDimsId`

Unique integer value corresponding to a symbolic dimension specification.

Returns

The number of dimensions for a `SymbDimsId`.

Example

For an example of how to use this function to configure an S-function that supports forward and backward propagation of symbolic dimensions, see “Configure Dimension Variants for S-Function Blocks”.

Introduced in R2016a

ssGetSymbolicDim

Get `SymbDimsId` from array of `SymbDimsId`s

Languages

C, C++

Syntax

```
SymbDimsId ssGetSymbolicDim(SimStruct *S, const SymbDimsId aSymbDimsId,
 const int_T aDimsIdx)
```

Arguments

`S`

SimStruct representing an S-Function block.

`aSymbDimsId`

Unique integer corresponding to a symbolic dimension specification.

`aDimsIdx`

Array index

Returns

A unique `SymbDimsId`.

Example

For an example of how to use this function to configure an S-function that supports forward and backward propagation of symbolic dimensions, see “Configure Dimension Variants for S-Function Blocks”.

Introduced in R2016a

ssSetInputPortSymbolicDimsId

Set precompiled `SymbDimsId` of input port

Languages

C, C++

Syntax

```
void ssSetInputPortSymbolicDimsId(SimStruct *S, const int_T aPortIdx,
 const SymbDimsId aSymbDimsId)
```

Arguments

`S`

SimStruct representing an S-Function block.

`aPortIdx`

Array index

`aSymbDimsId`

Unique integer value corresponding to a symbolic dimension specification.

Returns

This function does not return a value.

Example

You can call this function from inside the `mdlInitializeSizes` function. For an input port with an index of `0`, this example shows how to set the precompiled `SymbDimsId` equal to `inputDimsId`.

```
const SymbDimsId inputDimsId = ssRegisterSymbolicDimsExpr(S, "[A+3, B-2]");
    ssSetInputPortSymbolicDimsId(S, 0, inputDimsId);
```

Introduced in R2016a

ssGetCompInputPortSymbolicDimsId

Get compiled `SymbDimsId` of input port

Languages

C, C++

Syntax

```
SymbDimsId  ssGetCompInputPortSymbolicDimsId(SimStruct *S, const int_T aPortIdx)
```

Arguments

`S`

SimStruct representing an S-Function block.

`aPortIdx`

Index of an input port

Returns

`SymbDimsId` corresponding to symbolic dimensions of an input port.

Example

For an example of how to use this function to configure an S-function that supports forward and backward propagation of symbolic dimensions, see “Configure Dimension Variants for S-Function Blocks”.

Introduced in R2016a

ssSetCompInputPortSymbolicDimsId

Set compiled `SymbDimsId` of an input port

Languages

C, C++

Syntax

```
void ssSetCompInputPortSymbolicDimsId(SimStruct *S, const int_T aPortIdx,
  const SymbDimsId aSymbDimsId)
```

Arguments

`S`

SimStruct representing an S-Function block.

`aPortIdx`

Index of an input port.

`aSymbDimsId`

Unique integer value corresponding to a symbolic dimension specification.

Returns

This function does not return a value.

Example

For examples of how to use this function to configure S-functions that support forward propagation of symbolic dimensions and forward and backward propagation of symbolic dimensions, see “Configure Dimension Variants for S-Function Blocks”.

Introduced in R2016a

ssSetOutputPortSymbolicDimsId

Set precompiled SymbDimsId of an output port.

Languages

C, C++

Syntax

```
void ssSetOutputPortSymbolicDimsId(SimStruct *S, const int_T aPortIdx,
 const SymbDimsId aSymbDimsId)
```

Arguments

S

SimStruct representing an S-Function block.

aPortIdx

Index of an input port.

aSymbDimsId

Unique integer value corresponding to a symbolic dimension specification.

Returns

This function does not return a value.

Example

You can call this function from inside the mdlInitializeSizes function. For an output port with an index of 0, this example shows how to set the precompiled SymbDimsId equal to outputDimsId.

```
const SymbDimsId outputDimsId =
        ssRegisterSymbolicDimsExpr(S, "[ F / C , D * (B-3)]");
    ssSetOutputPortSymbolicDimsId(S, 0, outputDimsId);
```

Introduced in R2016a

ssGetCompOutputPortSymbolicDimsId

Get compiled `SymbDimsId` of output port

Languages

C, C++

Syntax

```
SymbDimsId ssGetCompOutputPortSymbolicDimsId(SimStruct *S, const int_T aPortIdx)
```

Arguments

`S`

SimStruct representing an S-Function block.

`aPortIdx`

Index of an output port.

Returns

`SymbDimsId` corresponding to symbolic dimensions of an output port.

Example

For an example of how to use this function to configure an S-function that supports forward and backward propagation of symbolic dimensions, see “Configure Dimension Variants for S-Function Blocks”.

Introduced in R2016a

ssSetCompOutputPortSymbolicDimsId

Set compiled `SymbDimsId` of output port

Languages

C, C++

Syntax

```
void ssSetCompOutputPortSymbolicDimsId(SimStruct *S, const int_T aPortIdx,
 const SymbDimsId aSymbDimsId)
```

Arguments

`S`

SimStruct representing an S-function block.

`aPortIdx`

Index of an output port.

`aSymbDimsId`

Unique integer value corresponding to a symbolic dimension specification.

Returns

This function does not return a value.

Example

For an example of how to use this function to configure an S-function that supports forward and backward propagation of symbolic dimensions, see "Configure Dimension Variants for S-Function Blocks".

Introduced in R2016a

ssSetCompDWorkSymbolicDimsId

Set compiled `SymbDimsId` of an index of a block's data type work (`DWork`) vector

Languages

C, C++

Syntax

```
void ssSetCompOutputPortSymbolicDimsId(SimStruct *S, const int_T aPortIdx,
 const SymbDimsId aSymbDimsId)
```

Arguments

`S`

SimStruct representing an S-Function block.

`aPortIdx`

Index of an input port

`aSymbDimsId`

Unique integer value corresponding to a symbolic dimension specification.

Returns

This function does not return a value.

Introduced in R2016a

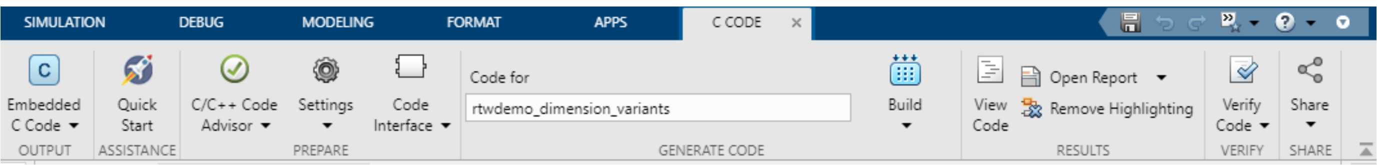
Apps

Embedded Coder

Generate readable, compact, and fast C and C++ code for embedded processors used in mass production

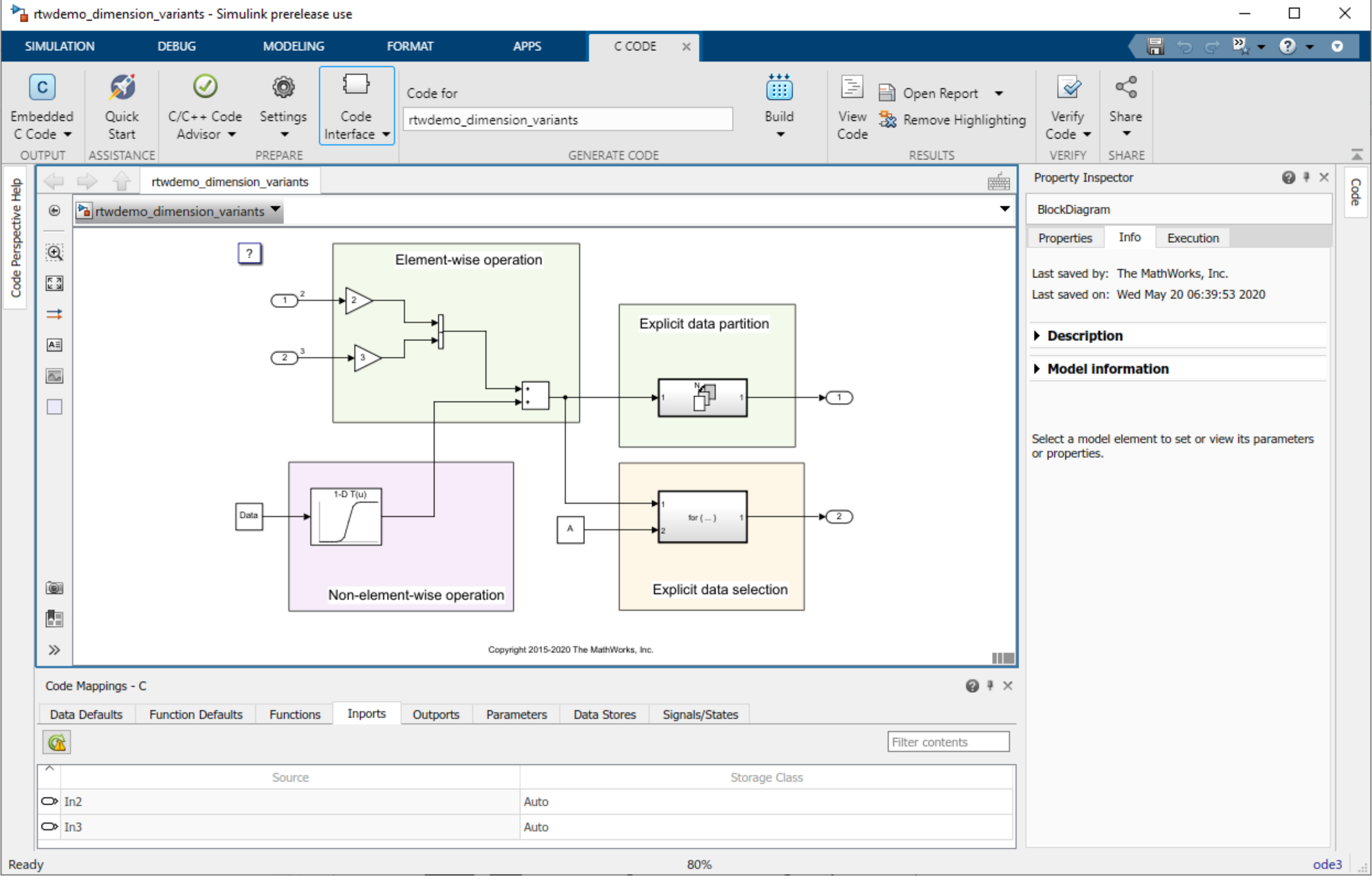
Description

Use the **Embedded Coder** app to generate C or C++ code from a model that represents a discrete-time system. The app extends the **Simulink Coder** app capabilities with advanced optimizations for precise control of the generated functions, files, and data. When you open the app, a **C Code** tab is added to the toolstrip. The **C Code** tab represents groups of tasks in the Embedded Coder workflow.



Use the app to perform these tasks:

- If you are new to Embedded Coder, use the Embedded Coder Quick Start to prepare your model for code generation. The Embedded Coder Quick Start chooses fundamental code generation settings based on your goals and application. Open the Embedded Coder Quick Start by clicking **Quick Start**.
- Set code generation objectives and prepare your model for code generation by clicking the **C/C++ Code Advisor**.
- To set model configuration parameters, select **Settings > C/C++ Code generation settings** or **Settings > Hardware Implementation**.
- Opening the **Embedded Coder** app opens the Code perspective. The Code perspective contains an integrated help pane, the Code Mappings editor, and the Property Inspector or Code view. Use the Code Mappings editor and the Property Inspector to configure data elements and entry-point functions in a model. Select **Code Interface > Individual Element Code Mappings**.
- To create custom code definitions, open the Embedded Coder Dictionary by selecting **Code Interface > Embedded Coder Dictionary**.
- Generate code only by selecting **Build > Generate Code**. Build the model and generate code by selecting **Build > Build**.
- To view the generated code alongside your model, use the Code view. You can trace between model elements and the code by clicking hyperlinked lines of code in the Code view. Open the latest code generation report by selecting **Open Report**.
- Verify the equivalence of the simulation and the code execution results by opening the **SIL/PIL** app. Select **Verify Code > SIL/PIL Manager**.
- Create a protected model for simulation and code generation to share with a third party by selecting **Share > Generate Protected Model**.
- Package the code and artifacts by selecting **Share > Generate Code and Package**.



Open the Embedded Coder App

In the **Apps** gallery, under **Code generation**, click **Embedded Coder**. The **C Code** tab opens.

Examples

- “Generate Code by Using the Quick Start Tool”
- “Generate C Code from Simulink Models”
- Code Mappings Editor – C
- Embedded Coder Dictionary
- “Design Models for Generated Embedded Code Deployment”

Tips

- If you are working with a model hierarchy, open the **Embedded Coder** app in the Simulink Editor window for the top model of the hierarchy that you are generating code for. On the **C Code** tab, the functionalities apply to the top model of the hierarchy that is open in the editor.
- To configure and view code for a referenced model, navigate to the model in the hierarchy and use the Code Mappings editor, Model Data Editor, Property Inspector, and Code view. These views apply to the active model, which can be the top model or a referenced model.

See Also

Functions
`coder.Dictionary`

Topics
"Generate Code by Using the Quick Start Tool"
"Generate C Code from Simulink Models"
Code Mappings Editor – C
Embedded Coder Dictionary
"Design Models for Generated Embedded Code Deployment"

Introduced in R2019b

Run on Custom Hardware

Run external mode simulations

Description

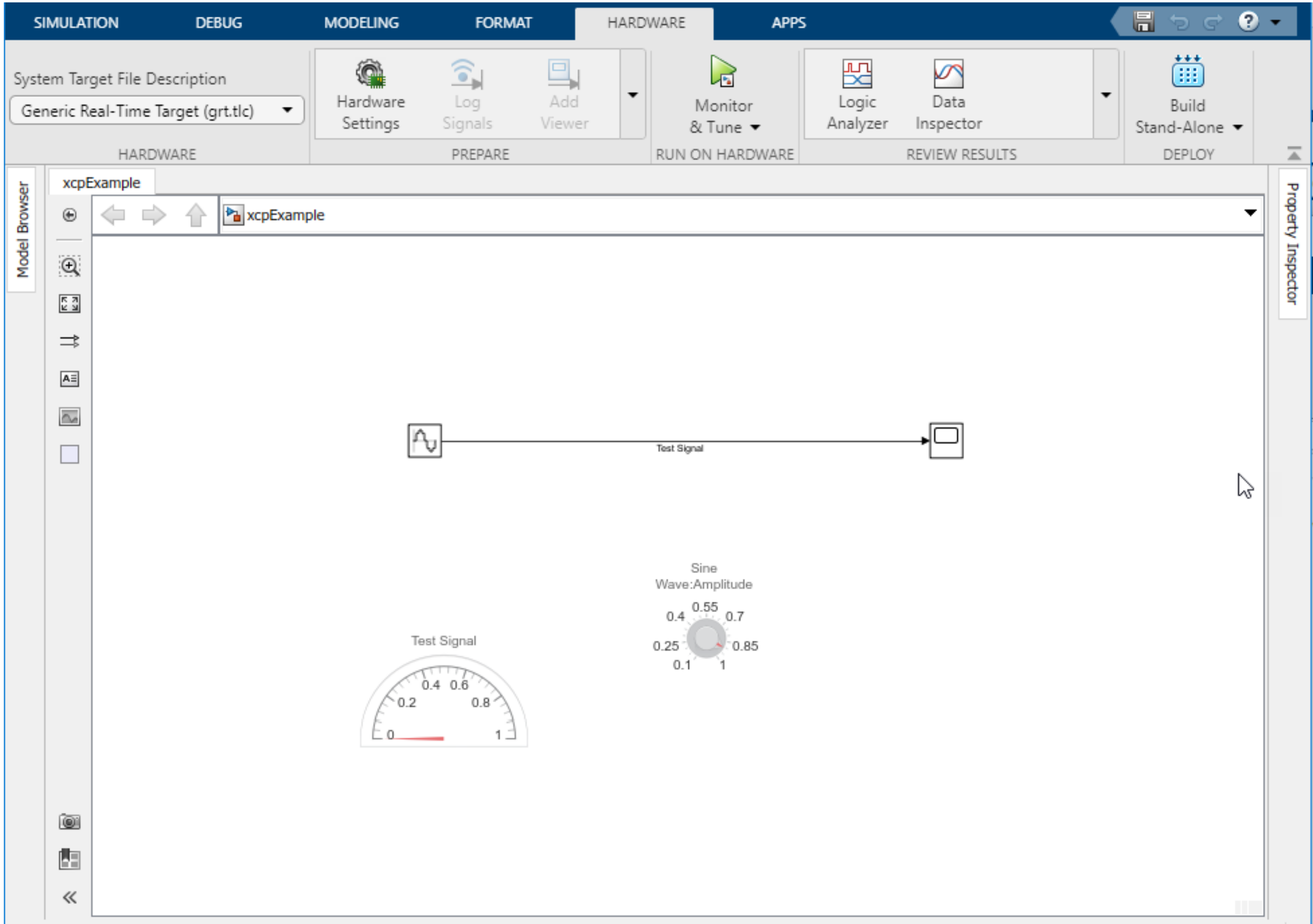
The **Run on Custom Hardware** app enables you to run external mode simulations on your development computer or other target hardware. In an external mode simulation, you can tune parameters in real time and monitor target application signals.

To run an external mode simulation, you:

1. Build the target application on your development computer.
2. Deploy the target application to the target hardware.
3. Connect Simulink to the target application that runs on the target hardware.
4. Start execution of generated code on the target hardware.

Using the app, you can:

- Perform the steps separately or with one click.
- Register custom launchers that deploy the target application.



Open the Run on Custom Hardware App

On the **Apps** tab, click **Run on Custom Hardware**. Or, on the **C Code** tab, select **Verify Code > Run on Custom Hardware**.

Examples

Run XCP External Mode Simulation

For an example that uses the **Run on Custom Hardware** app, see “Run XCP External Mode Simulation on Development Computer”.

See Also

Topics
“External Mode Simulation by Using XCP Communication”
“External Mode Simulation with TCP/IP or Serial Communication”

Introduced in R2019b

SIL/PIL Manager

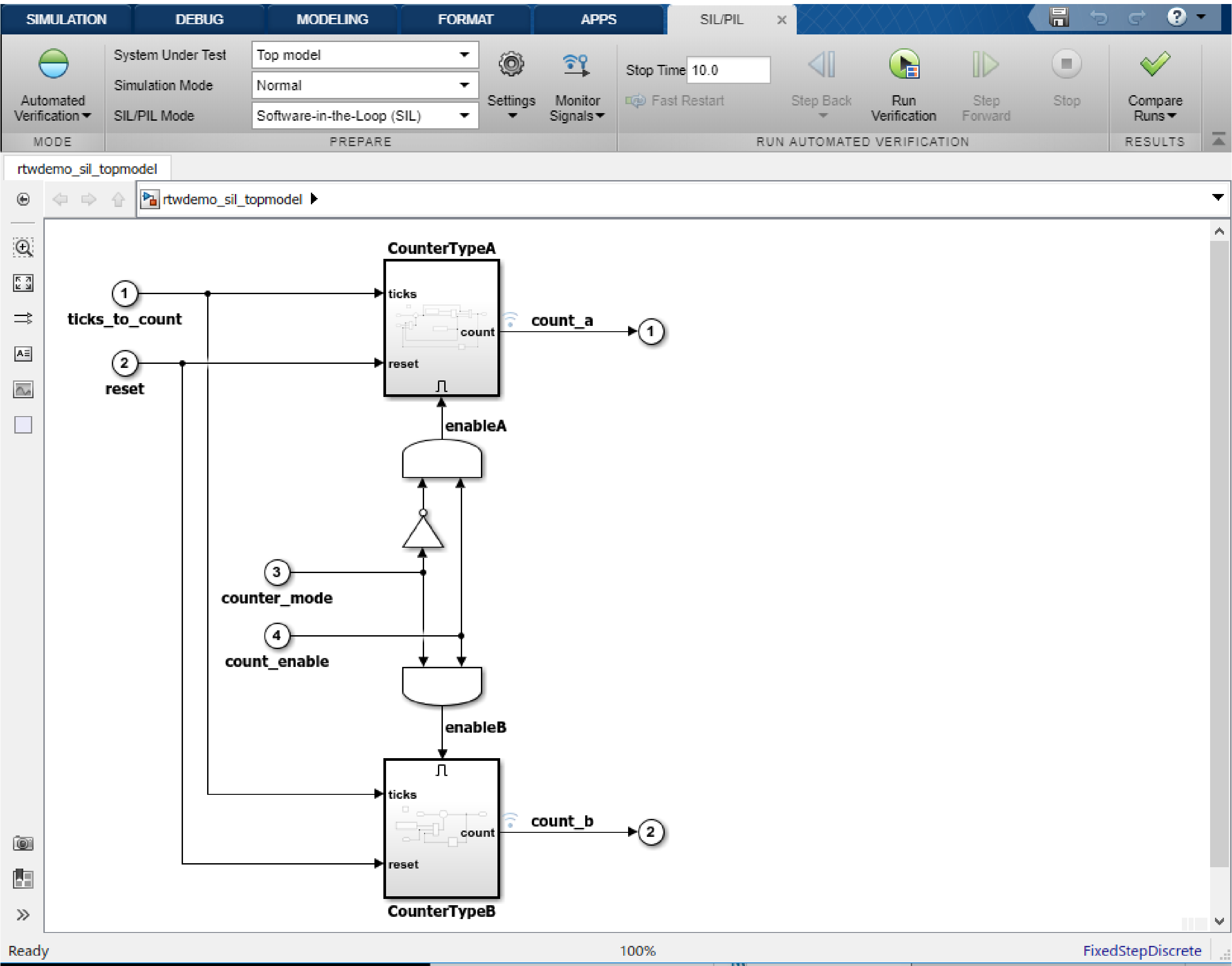
Verify generated code

Description

The SIL/PIL Manager simplifies verification of code that you generate from a model.

You can:

- With one click, test numeric equivalence between the model and generated code by running back-to-back model simulations and software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations.
- Configure SIL or PIL simulations to produce code coverage and execution-time profiling metrics.
- Enable your debugger for SIL simulations.
- Export automatically generated test cases for Simulink Test.



Open the SIL/PIL Manager App

On the Simulink toolstrip **Apps** tab, click **SIL/PIL Manager**. Or, on the Embedded Coder app **C Code** tab, click **Verify Code** > **SIL/PIL Manager**.

Examples

Verify Top-Model Code with a Single Click

1. In the Command Window, enter `rtwdemo_sil_topmodel`.
2. To open the SIL/PIL Manager, on the **Apps** tab, click **SIL/PIL Manager**.
3. On the **SIL/PIL** tab, use the supplied settings.
4. In the **Run Automated Verification** section, click **Run Verification**.

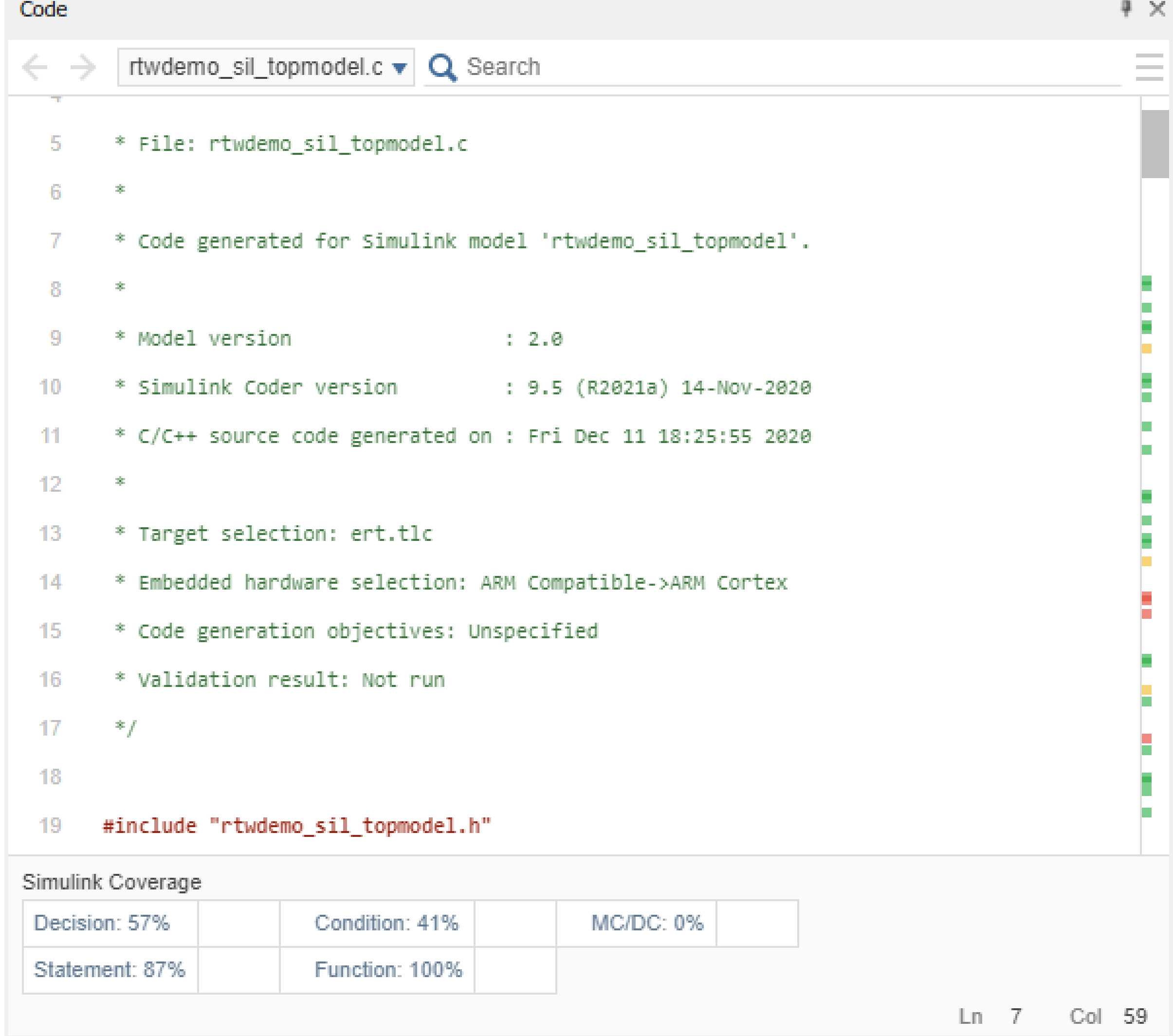
   The SIL/PIL Manager runs these simulations back-to-back:

- `rtwdemo_sil_topmodel` in normal mode
- `rtwdemo_sil_topmodel` in SIL mode. As the **Coverage Collection** and **Profile Code** controls are enabled, the SIL simulation also performs code coverage analysis and code execution profiling. For code coverage, you require Simulink Coverage.

You can monitor simulation progress through the Diagnostic Viewer.

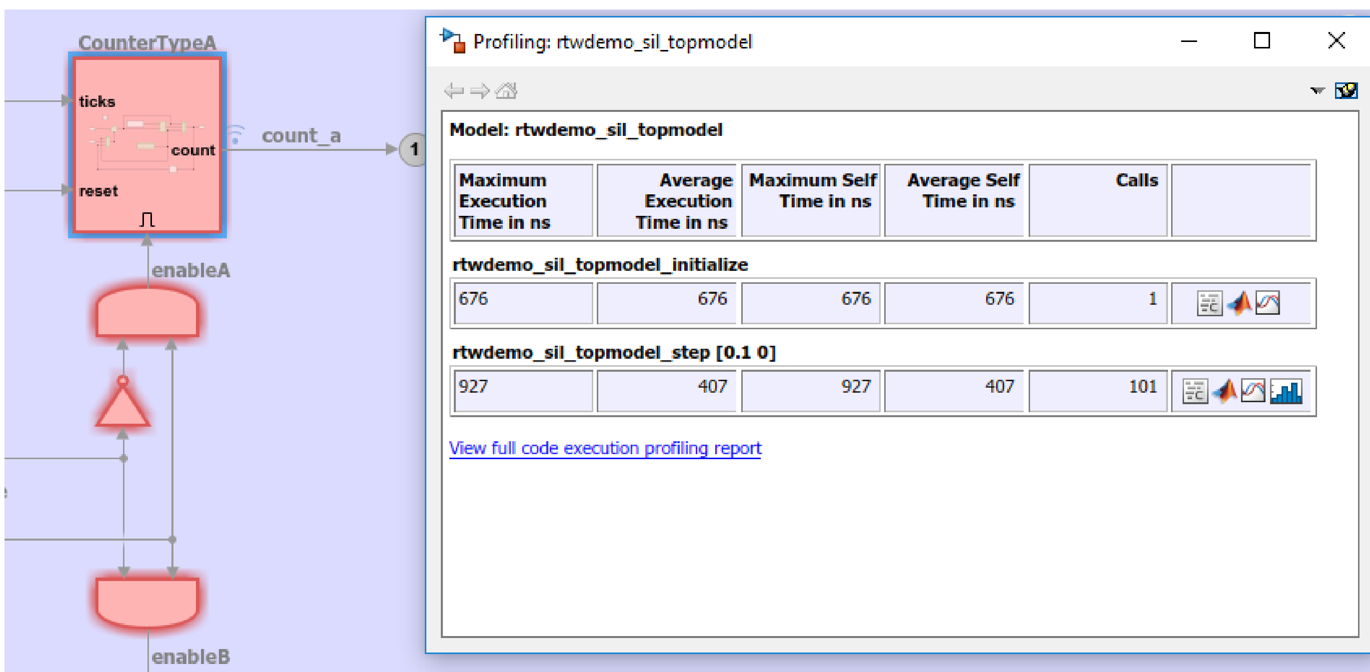
At the end of the second simulation:

- The SIL/PIL Manager displays generated code in the Code view, which enables you to analyze generated code, see code metrics, and trace between model elements and generated code.

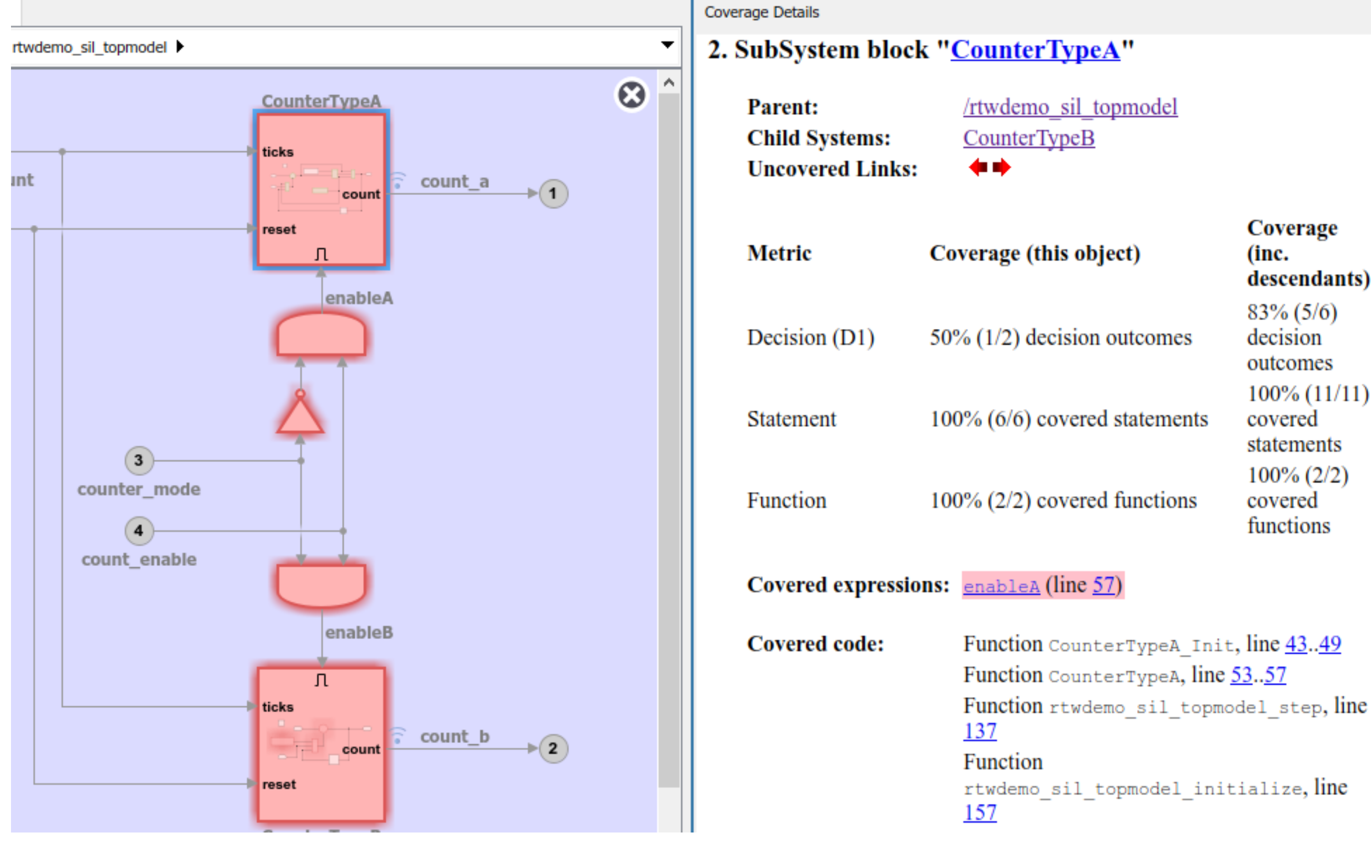


- The Simulation Data Inspector compares results from the model against results from generated code.

- To view execution-time metrics, in the model view, click the blue region.



- The Simulink Editor displays the **Coverage** tab. To display code coverage analysis results, in the **Review Results** section, click **Coverage Details**. To view coverage metrics for a specific block, in the model view, click the block, which is colored red.



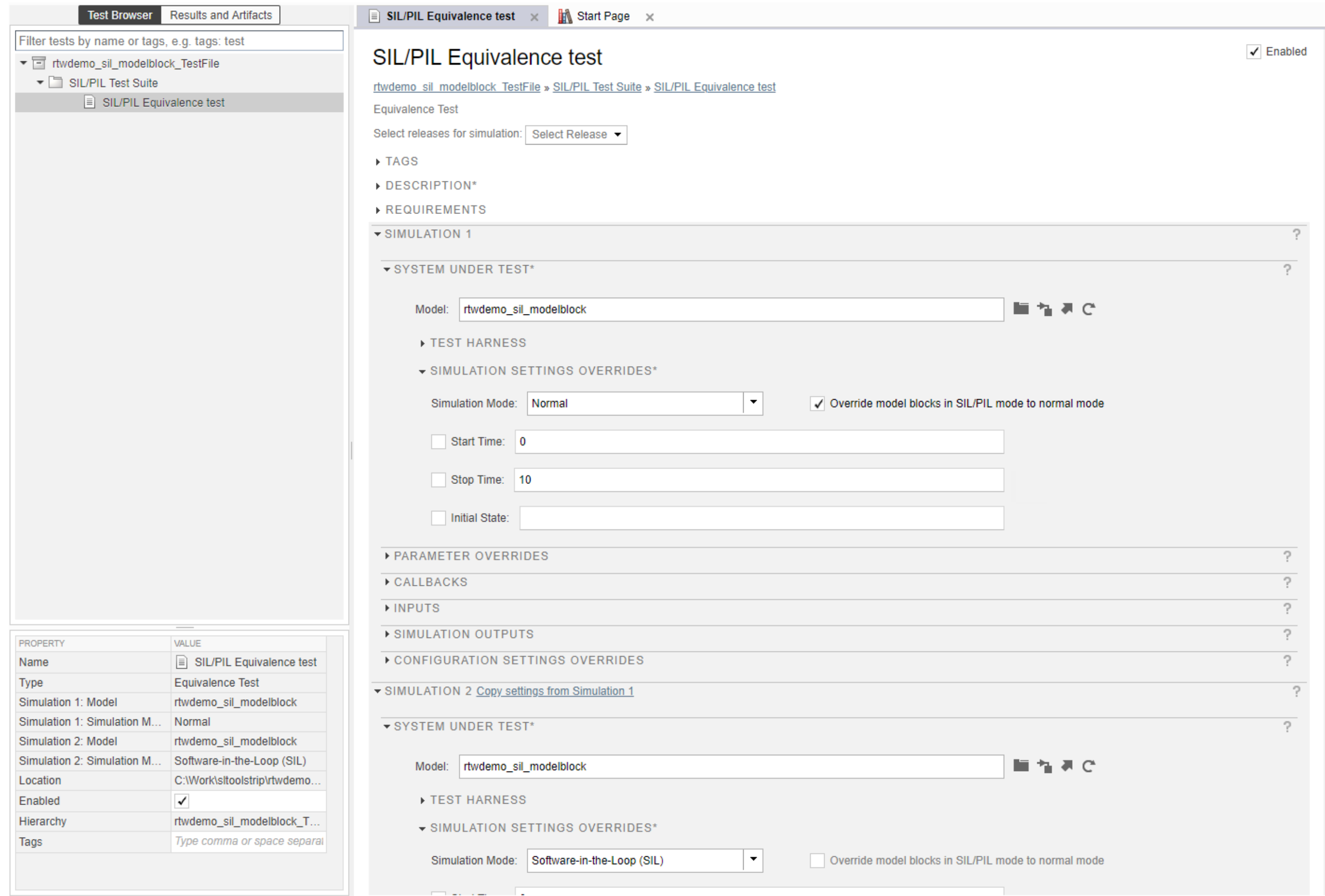
Export Numeric Equivalence Test Case for Simulink Test

If you have Simulink Test, you can export a numeric equivalence test case:

1. In the Command Window, enter `rtwdemo_sil_modelblock`.
2. To open the SIL/PIL Manager, on the **Apps** tab, click **SIL/PIL Manager**.
3. On the **SIL/PIL** tab, use the supplied settings.
4. Open the **Results** gallery. In the **Results** section, click either **Compare Runs** or **Data Inspector**.
5. Under **Simulink Test**, click **Export to Test Manager**.
6. In the Export SIL/PIL Test Cases dialog box, use the default settings, and click **OK**.

The SIL/PIL Manager:

- Creates the test case in `rtwdemo_sil_topmodel_TestFile.mldatx`, which you can find in the current working folder.
- Opens the test file that contains the test case.



For information about running test cases in Simulink Test, see “Import Test Cases for Equivalence Testing” (Simulink Test).

See Also

Topics

“SIL/PIL Manager Verification Workflow”
“Choose a SIL or PIL Approach”

Introduced in R2019b