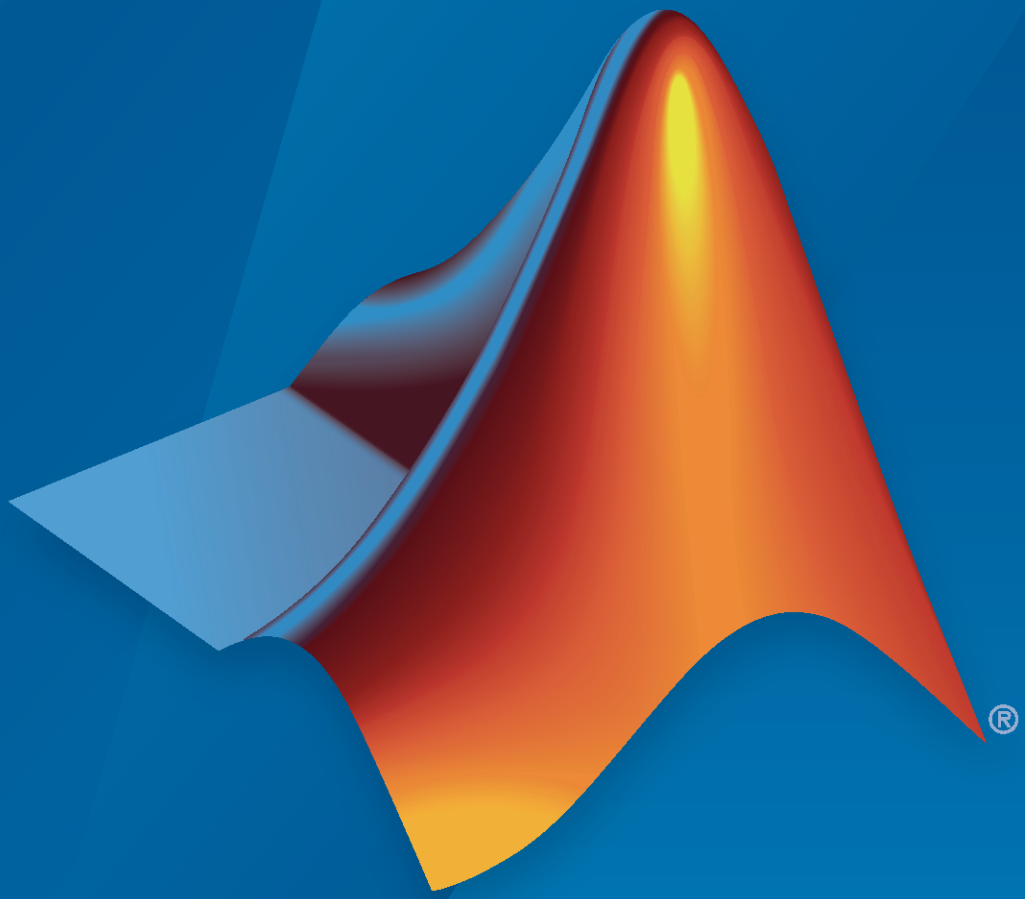
Simulink® Coder™

Reference



MATLAB® & SIMULINK®

R2021a

MathWorks®

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

Simulink® 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 8.0 (Release 2011a)
September 2011	Online only	Revised for Version 8.1 (Release 2011b)
March 2012	Online only	Revised for Version 8.2 (Release 2012a)
September 2012	Online only	Revised for Version 8.3 (Release 2012b)
March 2013	Online only	Revised for Version 8.4 (Release 2013a)
September 2013	Online only	Revised for Version 8.5 (Release 2013b)
March 2014	Online only	Revised for Version 8.6 (Release 2014a)
October 2014	Online only	Revised for Version 8.7 (Release 2014b)
March 2015	Online only	Revised for Version 8.8 (Release 2015a)
September 2015	Online only	Revised for Version 8.9 (Release 2015b)
October 2015	Online only	Rereleased for Version 8.8.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 8.10 (Release 2016a)
September 2016	Online only	Revised for Version 8.11 (Release 2016b)
March 2017	Online only	Revised for Version 8.12 (Release 2017a)
September 2017	Online only	Revised for Version 8.13 (Release 2017b)
March 2018	Online only	Revised for Version 8.14 (Release 2018a)
September 2018	Online only	Revised for Version 9.0 (Release 2018b)
March 2019	Online only	Revised for Version 9.1 (Release 2019a)
September 2019	Online only	Revised for Version 9.2 (Release 2019b)
March 2020	Online only	Revised for Version 9.3 (Release 2020a)
September 2020	Online only	Revised for Version 9.4 (Release 2020b)
March 2021	Online only	Revised for Version 9.5 (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 Simulink Code Generation Limitations

Simulink Code Generation Limitations **1-2**

2 Apps

3 Simulink Coder Functions

4 Simulink Coder Blocks

5 Simulation Target Parameters

Target library **5-2**
Description **5-2**
Settings **5-2**
Dependencies **5-2**
Command-Line Information **5-2**

Auto tuning **5-3**
Description **5-3**
Settings **5-3**
Dependencies **5-3**
Command-Line Information **5-3**

6 GPU Acceleration Parameters

Model Configuration Parameters: GPU Acceleration **6-2**

Simulation Target: GPU Acceleration Tab Overview **6-3**

Custom compute capability **6-4**
Description **6-4**
Settings **6-4**
Dependencies **6-4**
Command-Line Information **6-4**

Dynamic memory allocation threshold **6-5**
Description **6-5**
Settings **6-5**
Dependencies **6-5**
Command-Line Information **6-5**

Stack size per GPU thread **6-6**
Description **6-6**
Settings **6-6**
Dependencies **6-6**
Command-Line Information **6-6**

Include error checks in generated code **6-7**
Description **6-7**
Settings **6-7**
Dependencies **6-7**
Command-Line Information **6-7**

Additional compiler flags **6-8**
Description **6-8**
Settings **6-8**
Dependencies **6-8**
Command-Line Information **6-8**

7 Code Generation Parameters: Code Generation

Model Configuration Parameters: Code Generation **7-2**

Code Generation: General Tab Overview **7-6**
To get help on an option **7-6**

System target file **7-7**
Description **7-7**
Settings **7-7**
Tips **7-7**
Command-Line Information **7-7**
Recommended Settings **7-7**

Browse **7-9**
Description **7-9**
Tips **7-9**

Language **7-10**
Description **7-10**
Settings **7-10**
Dependencies **7-10**
Command-Line Information **7-10**
Recommended Settings **7-10**

Generate GPU code **7-12**
Description **7-12**
Settings **7-12**
Dependencies **7-12**
Command-Line Information **7-12**

Description **7-13**
Description **7-13**

Toolchain **7-14**
Description **7-14**
Settings **7-14**
Tip **7-14**
Command-Line Information **7-14**
Recommended Settings **7-14**

Build configuration **7-16**
Description **7-16**
Settings **7-16**
Tip **7-16**
Dependencies **7-17**
Command-Line Information **7-17**
Recommended Settings **7-17**

Tool/Options **7-18**
Description **7-18**
Settings **7-18**
Dependencies **7-18**
Command-Line Information **7-18**

Compiler optimization level **7-19**
Description **7-19**
Settings **7-19**
Tips **7-19**
Dependencies **7-19**
Command-Line Information **7-19**
Recommended Settings **7-20**

Custom compiler optimization flags **7-21**
Description **7-21**
Settings **7-21**
Dependency **7-21**
Command-Line Information **7-21**
Recommended Settings **7-21**

Generate makefile **7-22**
Description **7-22**
Settings **7-22**
Dependencies **7-22**
Command-Line Information **7-22**
Recommended Settings **7-22**

Make command **7-24**
Description **7-24**
Settings **7-24**
Tip **7-24**
Dependency **7-24**
Command-Line Information **7-24**
Recommended Settings **7-25**

Template makefile **7-26**
Description **7-26**
Settings **7-26**
Tips **7-26**
Dependency **7-26**
Command-Line Information **7-26**
Recommended Settings **7-26**

Select objective **7-28**
Description **7-28**
Settings **7-28**
Dependency **7-28**
Command-Line Information **7-28**
Recommended Settings **7-28**

Prioritized objectives **7-30**
Description **7-30**
Settings **7-30**
Dependencies **7-31**
Command-Line Information **7-31**
Recommended Settings **7-31**

Set Objectives **7-32**
Description **7-32**
Dependency **7-32**

Check Model **7-33**
Description **7-33**
Settings **7-33**
Dependency **7-33**

Check model before generating code **7-34**
Description **7-34**
Settings **7-34**
Command-Line Information **7-34**
Recommended Settings **7-34**

Generate code only **7-36**
Description **7-36**
Settings **7-36**

Tip . . . 7-36
Command-Line Information . . . 7-36
Recommended Settings . . . 7-36

Package code and artifacts . . . 7-37
Description . . . 7-37
Settings . . . 7-37
Dependency . . . 7-37
Command-Line Information . . . 7-37
Recommended Settings . . . 7-37

Zip file name . . . 7-39
Description . . . 7-39
Settings . . . 7-39
Dependency . . . 7-39
Command-Line Information . . . 7-39
Recommended Settings . . . 7-39

Custom FFT library callback . . . 7-40
Description . . . 7-40
Settings . . . 7-40
Limitation . . . 7-40
Tip . . . 7-40
Command-Line Information . . . 7-40
Recommended Settings . . . 7-40

Custom BLAS library callback . . . 7-42
Description . . . 7-42
Settings . . . 7-42
Limitation . . . 7-42
Tip . . . 7-42
Command-Line Information . . . 7-42
Recommended Settings . . . 7-42

Custom LAPACK library callback . . . 7-44
Description . . . 7-44
Settings . . . 7-44
Limitation . . . 7-44
Tip . . . 7-44
Command-Line Information . . . 7-44
Recommended Settings . . . 7-44

Verbose build . . . 7-46
Description . . . 7-46
Settings . . . 7-46
Command-Line Information . . . 7-46
Recommended Settings . . . 7-46

Retain .rtw file . . . 7-47
Description . . . 7-47
Settings . . . 7-47
Command-Line Information . . . 7-47
Recommended Settings . . . 7-47

Profile TLC **7-48**
Description **7-48**
Settings **7-48**
Command-Line Information **7-48**
Recommended Settings **7-48**

Start TLC debugger when generating code **7-49**
Description **7-49**
Settings **7-49**
Tips **7-49**
Command-Line Information **7-49**
Recommended Settings **7-49**

Start TLC coverage when generating code **7-50**
Description **7-50**
Settings **7-50**
Tip **7-50**
Command-Line Information **7-50**
Recommended Settings **7-50**

Enable TLC assertion **7-51**
Description **7-51**
Settings **7-51**
Command-Line Information **7-51**
Recommended Settings **7-51**

Show Custom Hardware App in Simulink Toolstrip **7-52**
Description **7-52**
Settings **7-52**
Command-Line Information **7-52**
Recommended Settings **7-52**

Show Embedded Hardware App in Simulink Toolstrip **7-53**
Description **7-53**
Settings **7-53**
Command-Line Information **7-53**
Recommended Settings **7-53**

8 Code Generation Parameters: Report

Model Configuration Parameters: Code Generation Report **8-2**

Code Generation: Report Tab Overview **8-4**
Configuration **8-4**

Create code generation report **8-5**
Description **8-5**
Settings **8-5**
Dependency **8-6**
Command-Line Information **8-6**
Recommended Settings **8-6**

Open report automatically **8-7**
Description **8-7**
Settings **8-7**
Dependency **8-7**
Command-Line Information **8-7**
Recommended Settings **8-7**

9 Code Generation Parameters: Comments

Model Configuration Parameters: Comments **9-2**
Auto generated comments **9-2**
Custom comments **9-2**

Code Generation: Comments Tab Overview **9-5**

Include comments **9-6**
Description **9-6**
Settings **9-6**
Dependencies **9-6**
Command-Line Information **9-6**
Recommended Settings **9-6**

Simulink block comments **9-8**
Description **9-8**
Settings **9-8**
Dependency **9-8**
Command-Line Information **9-8**
Recommended Settings **9-8**

Stateflow object comments **9-9**
Description **9-9**
Settings **9-9**
Dependency **9-9**
Command-Line Information **9-9**
Recommended Settings **9-9**

MATLAB source code as comments **9-11**
Description **9-11**
Settings **9-11**
Dependency **9-11**
Command-Line Information **9-11**
Recommended Settings **9-11**

Show eliminated blocks **9-13**
Description **9-13**
Settings **9-13**
Dependency **9-13**
Command-Line Information **9-13**
Recommended Settings **9-13**

Verbose comments for 'Model default' storage class **9-14**
Description **9-14**
Settings **9-14**
Dependency **9-14**
Command-Line Information **9-14**
Recommended Settings **9-15**

10 Code Generation Parameters: Identifiers

Model Configuration Parameters: Code Generation Identifiers **10-2**

Code Generation: Identifiers Tab Overview **10-4**

Maximum identifier length **10-5**
Description **10-5**
Settings **10-5**
Tips **10-5**
Command-Line Information **10-5**
Recommended Settings **10-5**

Use the same reserved names as Simulation Target **10-7**
Description **10-7**
Settings **10-7**
Command-Line Information **10-7**
Recommended Settings **10-7**

Reserved names **10-8**
Description **10-8**
Settings **10-8**
Tips **10-8**
Command-Line Information **10-8**
Recommended Settings **10-8**

Duplicate enumeration member names **10-10**
Description **10-10**
Settings **10-10**
Command-Line Information **10-10**
Example **10-10**
Recommended Settings **10-11**

11 Code Generation Parameters: Custom Code

Model Configuration Parameters: Code Generation Custom Code **11-2**

Code Generation: Custom Code Tab Overview **11-3**
Configuration **11-3**

Use the same custom code settings as Simulation Target **11-4**
Description **11-4**
Settings **11-4**
Command-Line Information **11-4**
Recommended Settings **11-4**

Source file **11-5**
Description **11-5**
Settings **11-5**
Command-Line Information **11-5**
Recommended Settings **11-5**

Header file **11-6**
Description **11-6**
Settings **11-6**
Command-Line Information **11-6**
Recommended Settings **11-6**

Initialize function **11-7**
Description **11-7**
Settings **11-7**
Command-Line Information **11-7**
Recommended Settings **11-7**

Terminate function **11-8**
Description **11-8**
Settings **11-8**
Dependency **11-8**
Command-Line Information **11-8**
Recommended Settings **11-8**

Include directories **11-9**
Description **11-9**
Settings **11-9**
Command-Line Information **11-9**
Recommended Settings **11-9**

Source files **11-11**
Description **11-11**
Settings **11-11**
Limitation **11-11**
Tip **11-11**
Command-Line Information **11-11**
Recommended Settings **11-11**

Libraries **11-12**
Description **11-12**
Settings **11-12**
Limitation **11-12**
Tip **11-12**
Command-Line Information **11-12**
Recommended Settings **11-12**

Defines **11-13**
Description **11-13**

Settings 11-13
Command-Line Information 11-13
Recommended Settings 11-13

12 Code Generation Parameters: Interface

Model Configuration Parameters: Code Generation Interface 12-2

Code Generation: Interface Tab Overview 12-8

Code replacement library 12-9
Description 12-9
Settings 12-9
Tips 12-9
Dependencies 12-9
Command-Line Information 12-9
Recommended Settings 12-10

Code replacement libraries 12-11
Description 12-11
Settings 12-11
Tips 12-11
Dependencies 12-12
Command-Line Information 12-12
Recommended Settings 12-12

Shared code placement 12-13
Description 12-13
Settings 12-13
Command-Line Information 12-13
Recommended Settings 12-13

Support: non-finite numbers 12-15
Description 12-15
Settings 12-15
Dependencies 12-15
Command-Line Information 12-15
Recommended Settings 12-15

Code interface packaging 12-17
Description 12-17
Settings 12-17
Tips 12-17
Dependencies 12-18
Command-Line Information 12-18
Recommended Settings 12-18

Multi-instance code error diagnostic 12-20
Description 12-20
Settings 12-20
Dependencies 12-20

Command-Line Information 12-20
Recommended Settings 12-20

Generate C API for: signals 12-22
Description 12-22
Settings 12-22
Command-Line Information 12-22
Recommended Settings 12-22

Generate C API for: parameters 12-23
Description 12-23
Settings 12-23
Command-Line Information 12-23
Recommended Settings 12-23

Generate C API for: states 12-24
Description 12-24
Settings 12-24
Command-Line Information 12-24
Recommended Settings 12-24

Generate C API for: root-level I/O 12-25
Description 12-25
Settings 12-25
Command-Line Information 12-25
Recommended Settings 12-25

ASAP2 interface 12-26
Description 12-26
Settings 12-26
Command-Line Information 12-26
Recommended Settings 12-26

External mode 12-27
Description 12-27
Settings 12-27
Command-Line Information 12-27
Recommended Settings 12-27

Transport layer 12-28
Description 12-28
Settings 12-28
Tips 12-28
Dependency 12-29
Command-Line Information 12-29
Recommended Settings 12-29

MEX-file arguments 12-30
Description 12-30
Settings 12-30
Dependency 12-31
Command-Line Information 12-31
Recommended Settings 12-31

Static memory allocation **12-32**
Description **12-32**
Settings **12-32**
Tips **12-32**
Dependencies **12-32**
Command-Line Information **12-32**
Recommended Settings **12-32**

Target library **12-34**
Description **12-34**
Settings **12-34**
Dependencies **12-34**
Command-Line Information **12-34**

ARM Compute Library version **12-36**
Description **12-36**
Settings **12-36**
Dependencies **12-36**
Command-Line Information **12-36**

ARM Compute Library architecture **12-37**
Description **12-37**
Settings **12-37**
Dependencies **12-37**
Command-Line Information **12-37**

Auto tuning **12-38**
Description **12-38**
Settings **12-38**
Dependencies **12-38**
Command-Line Information **12-38**

Static memory buffer size **12-39**
Description **12-39**
Settings **12-39**
Tips **12-39**
Dependency **12-39**
Command-Line Information **12-39**
Recommended Settings **12-39**

LUT object struct order for even spacing specification **12-41**
Description **12-41**
Settings **12-41**
Command-Line Information **12-41**
Recommended Settings **12-41**

LUT object struct order for explicit value specification **12-42**
Description **12-42**
Settings **12-42**
Command-Line Information **12-42**
Recommended Settings **12-42**

Buffer size of dynamically-sized string (bytes) **12-43**
Description **12-43**
Settings **12-43**

Command-Line Information 12-43
Recommended Settings 12-43

Array layout 12-44
Description 12-44
Settings 12-44
Command-Line Information 12-44
Recommended Settings 12-44

External functions compatibility for row-major code generation 12-46
Description 12-46
Settings 12-46
Dependencies 12-46
Command-Line Information 12-46
Recommended Settings 12-46

Standard math library 12-48
Description 12-48
Settings 12-48
Tips 12-48
Dependencies 12-49
Command-Line Information 12-49
Recommended Settings 12-49

Maximum word length 12-50
Description 12-50
Settings 12-50
Tips 12-50
Dependencies 12-50
Command-Line Information 12-50
Recommended Settings 12-51

Classic call interface 12-52
Description 12-52
Settings 12-52
Tips 12-52
Dependencies 12-52
Command-Line Information 12-52
Recommended Settings 12-53

Single output/update function 12-54
Description 12-54
Settings 12-54
Tips 12-54
Dependencies 12-54
Command-Line Information 12-55
Recommended Settings 12-55

MAT-file logging 12-56
Description 12-56
Settings 12-56
Dependencies 12-56
Limitations 12-57
Command-Line Information 12-57
Recommended Settings 12-57

13 Code Generation Parameters: GPU Code

Model Configuration Parameters: GPU Code **13-2**

Code Generation: GPU Code Tab Overview **13-3**

GPU index **13-4**
Description **13-4**
Settings **13-4**
Dependencies **13-4**
Command-Line Information **13-4**

Compute capability **13-5**
Description **13-5**
Settings **13-5**
Dependencies **13-5**
Command-Line Information **13-5**

Custom compute capability **13-6**
Description **13-6**
Settings **13-6**
Dependencies **13-6**
Command-Line Information **13-6**

Memory mode **13-7**
Description **13-7**
Settings **13-7**
Dependencies **13-7**
Command-Line Information **13-7**
Compatibility Considerations **13-7**

Maximum blocks per kernel **13-9**
Description **13-9**
Settings **13-9**
Dependencies **13-9**
Command-Line Information **13-9**

Dynamic memory allocation threshold **13-10**
Description **13-10**
Settings **13-10**
Dependencies **13-10**
Command-Line Information **13-10**

Stack size per GPU thread **13-11**
Description **13-11**
Settings **13-11**
Dependencies **13-11**
Command-Line Information **13-11**

Include error checks in generated code **13-12**
Description **13-12**
Settings **13-12**
Dependencies **13-12**

Command-Line Information 13-12

Kernel name prefix 13-13
Description 13-13
Settings 13-13
Dependencies 13-13
Command-Line Information 13-13

Additional compiler flags 13-14
Description 13-14
Settings 13-14
Dependencies 13-14
Command-Line Information 13-14

cuBLAS 13-15
Description 13-15
Settings 13-15
Dependencies 13-15
Command-Line Information 13-15

cuSOLVER 13-16
Description 13-16
Settings 13-16
Dependencies 13-16
Command-Line Information 13-16

cuFFT 13-17
Description 13-17
Settings 13-17
Dependencies 13-17
Command-Line Information 13-17

14 Simulink Coder Parameters: Advanced Parameters

Use Simulink Coder Features 14-2
Description 14-2
Settings 14-2
Dependencies 14-2
Command-Line Information 14-2

15 Configuration Parameters for Simulink Models

Code Generation Pane: RSim Target 15-2
Code Generation: RSim Target Tab Overview 15-2
Enable RSim executable to load parameters from a MAT-file 15-2
Solver selection 15-3
Force storage classes to AUTO 15-4

Code Generation Pane: S-Function Target **15-5**
Code Generation S-Function Target Tab Overview **15-5**
Create new model **15-5**
Use value for tunable parameters **15-6**
Include custom source code **15-6**

Code Generation Pane: Tornado Target **15-8**
Code Generation: Tornado Target Tab Overview **15-9**
Standard math library **15-9**
Code replacement library **15-10**
Shared code placement **15-11**
MAT-file logging **15-12**
MAT-file variable name modifier **15-13**
Code Format **15-14**
StethoScope **15-14**
Download to VxWorks target **15-15**
Base task priority **15-16**
Task stack size **15-17**
External mode **15-17**
Transport layer **15-18**
MEX-file arguments **15-19**
Static memory allocation **15-20**
Static memory buffer size **15-20**

Recommended Settings Summary for Model Configuration Parameters **15-22**

16 Model Advisor Checks

Simulink Coder Checks **16-2**
Simulink Coder Checks Overview **16-2**
Check reuse of subsystem code **16-2**
Identify blocks using one-based indexing **16-3**
Check solver for code generation **16-3**
Check for blocks not supported by code generation **16-4**
Check and update model to use toolchain approach to build generated code **16-5**
Check and update embedded target model to use ert.tlc system target file **16-6**
Check and update models that are using targets that have changed significantly across different releases of MATLAB **16-8**
Check for blocks that have constraints on tunable parameters **16-8**
Check for model reference configuration mismatch **16-9**
Check sample times and tasking mode **16-10**
Check for code generation identifier formats used for model reference **16-11**
Check for relative execution order change for Data Store Read and Data Store Write blocks **16-11**
Available Checks for Code Generation Objectives **16-12**
Identify questionable blocks within the specified system **16-18**

Check model configuration settings against code generation objectives 16-19

Code Generation Advisor Checks 16-21
Available Checks for Code Generation Objectives 16-21
Identify questionable blocks within the specified system 16-27
Check model configuration settings against code generation objectives 16-28

17 Parameters for Creating Protected Models

Create Protected Model 17-2
Create Protected Model: Overview 17-2
Open read-only view of model 17-3
Simulate 17-3
Use generated code 17-4
Code interface 17-4
Content type 17-5
Use generated HDL code 17-6
Destination folder 17-6
Contents 17-7
Create harness model for protected model 17-7
Name of project archive (.mlproj) 17-8

18 Simulink Coder Tools

19 Optimization Parameters

Model Configuration Parameters: Code Generation Optimization 19-2

Optimization Pane: Tab Overview 19-5
Tips 19-5
To get help on an option 19-5

Remove code from floating-point to integer conversions that wraps out-of-range values 19-6
Description 19-6
Settings 19-6
Tips 19-6
Dependency 19-6
Command-Line Information 19-6
Recommended Settings 19-7

Buffer for reusable subsystems **19-8**
Description **19-8**
Settings **19-8**
Command-Line Information **19-8**
Recommended Settings **19-8**

Default parameter behavior **19-9**
Description **19-9**
Settings **19-9**
Tips **19-9**
Dependencies **19-9**
Command-Line Information **19-10**
Recommended Settings **19-10**

Inline invariant signals **19-11**
Description **19-11**
Settings **19-11**
Dependencies **19-11**
Command-Line Information **19-11**
Recommended Settings **19-11**

Use memcpy for vector assignment **19-13**
Description **19-13**
Settings **19-13**
Dependencies **19-13**
Command-Line Information **19-13**
Recommended Settings **19-13**

Memcpy threshold (bytes) **19-15**
Description **19-15**
Settings **19-15**
Dependencies **19-15**
Command-Line Information **19-15**
Recommended Settings **19-15**

Loop unrolling threshold **19-16**
Description **19-16**
Settings **19-16**
Dependency **19-16**
Command-Line Information **19-16**
Recommended Settings **19-16**

Maximum stack size (bytes) **19-17**
Description **19-17**
Settings **19-17**
Tips **19-17**
Command-Line Information **19-17**
Recommended Settings **19-18**

Use bitsets for storing state configuration **19-19**
Description **19-19**
Settings **19-19**
Tips **19-19**
Dependency **19-19**
Command-Line Information **19-19**

Recommended Settings 19-19

Use bitsets for storing Boolean data 19-21
Description 19-21
Settings 19-21
Tips 19-21
Dependency 19-21
Command-Line Information 19-21
Recommended Settings 19-21

Base storage type for automatically created enumerations 19-23
Description 19-23
Settings 19-23
Tips 19-23
Dependency 19-23
Command-Line Information 19-23

Enable local block outputs 19-25
Description 19-25
Settings 19-25
Tips 19-25
Dependencies 19-25
Command-Line Information 19-25
Recommended Settings 19-25

Reuse local block outputs 19-27
Description 19-27
Settings 19-27
Dependencies 19-27
Command-Line Information 19-27
Recommended Settings 19-27

Eliminate superfluous local variables (Expression folding) 19-29
Description 19-29
Settings 19-29
Dependencies 19-29
Command-Line Information 19-29
Recommended Settings 19-29

Remove code from floating-point to integer conversions with saturation that maps NaN to zero 19-31
Description 19-31
Settings 19-31
Tips 19-31
Dependencies 19-31
Command-Line Information 19-31
Recommended Settings 19-32

Use memset to initialize floats and doubles to 0.0 19-33
Description 19-33
Settings 19-33
Dependency 19-33
Command-Line Information 19-33
Recommended Settings 19-33

Signal storage reuse **19-35**
Description **19-35**
Settings **19-35**
Tips **19-35**
Dependencies **19-35**
Command-Line Information **19-36**
Recommended Settings **19-36**

Simulink Code Generation Limitations

Simulink Code Generation Limitations

The following topics identify Simulink code generation limitations:

- “C++ Language Support Limitations”
- “Limitations”
- “Tunable Expression Limitations”
- “Generate Reentrant Code from Subsystems”
- “Code Generation Limitations for Model Reference”
- “TCP/IP and Serial External Mode Limitations”
- “Noninlined S-Function Parameter Type Limitations”
- “S-Function Target Limitations”
- “Rapid Simulation Target Limitations”
- “Asynchronous Support Limitations”
- “C API Limitations”
- “Blocks and Products Supported for Code Generation”

Apps

Simulink Coder

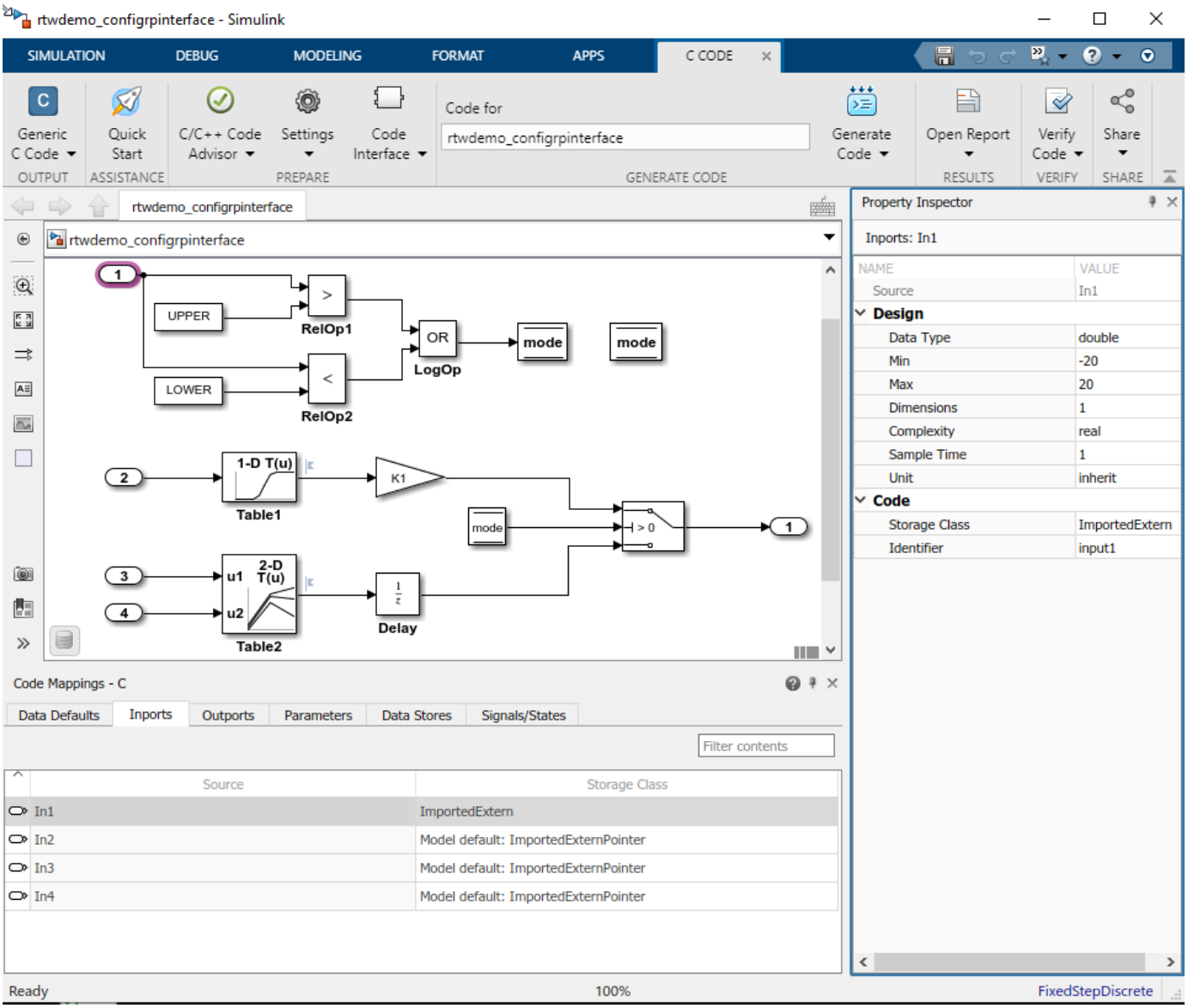
Generate and execute C and C++ code from Simulink models, Stateflow charts, and MATLAB functions for use in applications such as simulation acceleration, rapid prototyping, and hardware-in-the-loop (HIL) simulations

Description

To generate C or C++ code from a model that represents a discrete-time system, use the **Simulink Coder** app. When you open the app, a **C Code** tab is added to the Simulink Editor toolstrip. The **C Code** tab contains sections that represent steps of the Simulink Coder workflow. You can iterate through model preparation for code generation, code generation and builds, and code verification by using the editor window panels.

Use the app to:

- Learn product basics while quickly configuring a model for code generation. If you are new to Simulink Coder, use the Simulink Coder Quick Start to prepare a model for code generation. The Simulink Coder Quick Start chooses fundamental code generation settings based on your goals and application. To open the Simulink Coder Quick Start, click **Quick Start**.
- Set code generation objectives and prepare a model for code generation. Click **C/C++ Code Advisor**.
- Configure model-wide code generation parameter settings. Click **Settings**.
- Configure data. Select **Code Interface > Default Code Mappings** or **Code Interface > Individual Element Code Mappings**, which open the Code Mappings editor and Property Inspector.
- Generate code or build an executable program. Select **Build > Generate Code**or **Build > Build**.
- Review generated code. Click **Open Report**.
- Create a protected model for simulation and code generation to share with a third party. Select **Share > Generate Protected Model**.
- Package generated code and artifacts for deployment to another computer system. Select **Share > Generate Code and Package**.



Open the Simulink Coder App

In the **Apps** gallery, under **Code generation**, click **Simulink Coder**. The **C Code** tab opens.

Examples

- “Generate Code by Using the Quick Start Tool”
- “Generate Code Using Simulink® Coder™”
- “Design Models for Rapid Prototyping Deployment”

Tips

- If you are working with a model hierarchy, open the **Simulink 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 data for a referenced model, navigate to the model in the hierarchy. Open the Code Mappings editor and Property Inspector by selecting **Code Interface > Default Code Mappings** or **Code InterfaceIndividual Element Code Mappings**. These tools apply to the active model, which can be the top model or a referenced model.

See Also

Topics

“Generate Code by Using the Quick Start Tool”
“Generate Code Using Simulink® Coder™”
“Design Models for Rapid Prototyping Deployment”

Introduced in R2019b

Run on Custom Hardware

Run external mode simulations

Description

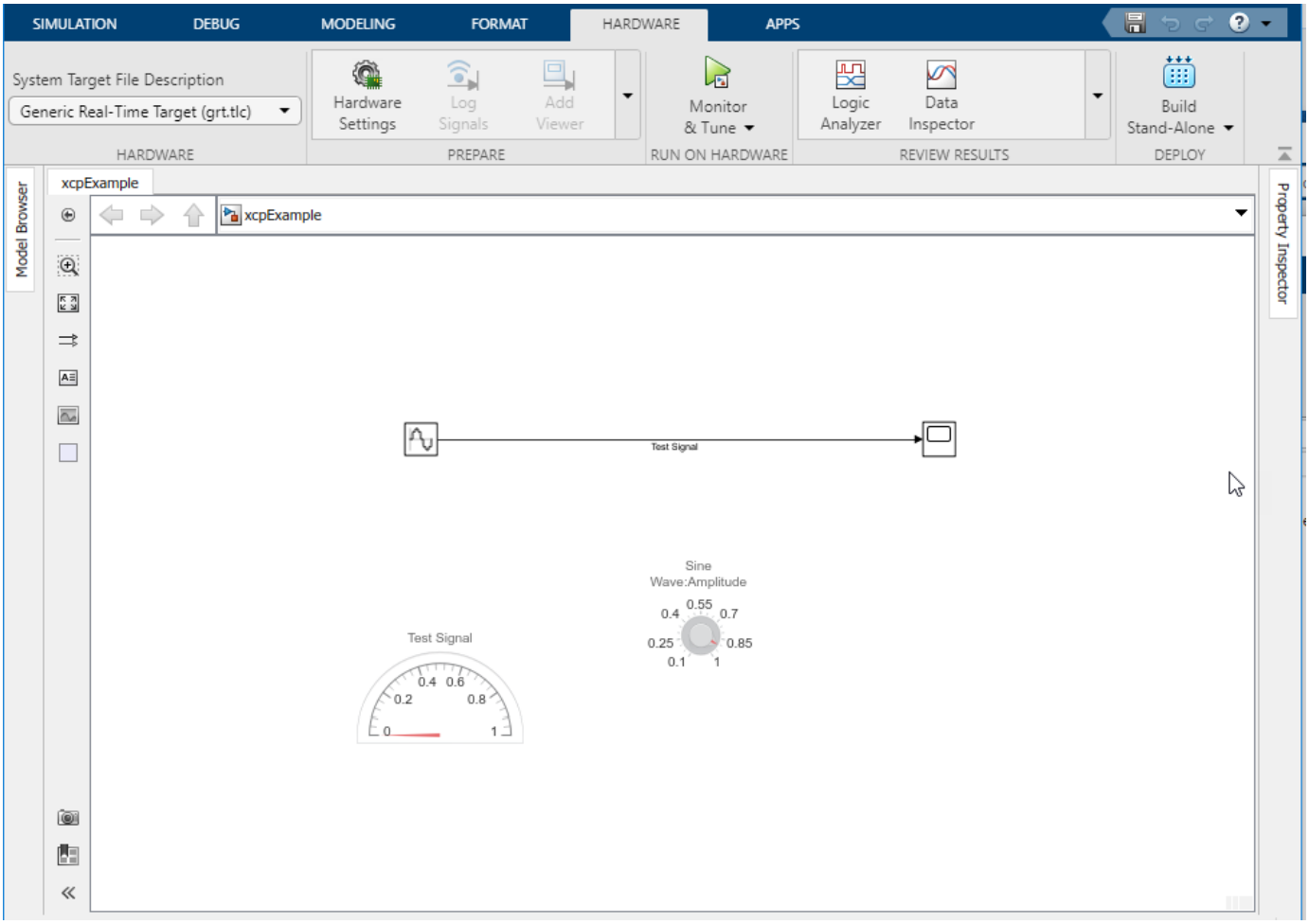
The **Run on Custom Hardware** app enables you to run external mode simulations on your development computer or other target hardware. In an external mode simulation, you can tune parameters in real time and monitor target application signals.

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

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` | `addSourceFiles` | `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 3-0 .)

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” (Embedded Coder).

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” (Embedded Coder)

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 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. Reduce the effort of preparing a model for C code generation by specifying default configurations for categories of data elements across a model. Override default configurations by configuring data elements individually. For smaller models, you can choose to configure each data element 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`.

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
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
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” (Embedded Coder)

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` | `Dictionary default` | `ExportedGlobal` | `ImportedExtern` | `ImportedExternPointer` | `Model default`

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`

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');
```

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` | `ExternalParameterObjects` | `Inports` | `ModelParameters` | `Outports` | `Signals` | `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` | `Dictionary default` | `ExportedGlobal` | `ImportedExtern` | `ImportedExternPointer` | `Model default`

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`

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

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 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 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” (Embedded Coder)

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');
```

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

`ExternalParameterObjects` | `GlobalDataStores` | `Inports` | `InternalData` | `ModelParameters` | `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`

Code mapping property that you return a value for. Specify one of these property names.

Information to Return	Property Name
Name of storage class	`StorageClass`

Information to Return	Property Name
Name of variable for data element in the generated code	`Identifier`

Example: `'Identifier'`

Output Arguments

`propertyValue` — Name of storage class or value of storage class property

character vector | string scalar | `Auto` | `Dictionary default` | `ExportedGlobal` | `ImportedExtern` | `ImportedExternPointer` | `Model default`

The property value is one of these values depending on the category and property that you specify.

Property	Value Returned
`StorageClass`	One of these values: `Auto`, , `Dictionary default`, `ExportedGlobal`, , `ImportedExtern`, `ImportedExternPointer`,

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”

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` |

Code mapping property for which to return a value. Specify one of these property names.

Information to Return	Property Name
Name of storage class	`StorageClass`
Name of variable for data store in the generated code	`Identifier`

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” (Embedded Coder)
“Configure Name and Arguments for Individual Step Functions” (Embedded Coder)

“Configure Default C Code Generation for Categories of Data Elements and Functions” (Embedded Coder)
“C Code Generation Configuration for Model Interface Elements” (Embedded Coder)

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” (Embedded Coder)
“C Code Generation Configuration for Model Interface Elements” (Embedded Coder)

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` |

Code mapping property for which to return a value. Specify one of these property names.

Information to Return	Property Name
Name of storage class	`StorageClass`
Name of variable for inport in the generated code	`Identifier`

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. Use this function to return the storage class or the value of a storage class property configured for the parameter.

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');
```

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
character vector | string scalar

Name of the model workspace parameter for which to return the code mapping information.

Example: `'Table1'`

Data Types: `char` | `string`

`property` — Code mapping property value to return
`StorageClass` | `Identifier` | `,`

Code mapping property for which to return a value. Specify one of these property names.

Information to Return	Property Name
Name of storage class	`StorageClass`
Name of variable for the parameter in the generated code	`Identifier`

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.

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`

Code mapping property for which to return a value. Specify one of these property names.

Information to Return	Property Name
Name of storage class	`StorageClass`
Name of variable for root-level outport in the generated code	`Identifier`

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` |

Code mapping property for which to return a value. Specify one of these property names.

Information to Return	Property Name
Name of storage class	`StorageClass`
Name of variable for signal data in the generated code	`Identifier`

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` | | | |

Code mapping property for which to return a value. Specify one of these property names.

Information to Return	Property Name
Name of storage class	`StorageClass`
Name of variable for state in the generated code	`Identifier`

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');
```

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

`ExternalParameterObjects` | `GlobalDataStores` | `Inports` | `InternalData` | `ModelParameters` | `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` | `Dictionary default` | `ExportedGlobal` | `ImportedExtern` | `ImportedExternPointer` | `Model default`

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'

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”

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` | `Dictionary default` | `ExportedGlobal` | `ImportedExtern` | `ImportedExternPointer` | `Model default`

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`

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” (Embedded Coder)
“Configure Name and Arguments for Individual Step Functions” (Embedded Coder)
“Configure Default C Code Generation for Categories of Data Elements and Functions” (Embedded Coder)
“C Code Generation Configuration for Model Interface Elements” (Embedded Coder)

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” (Embedded Coder)
“C Code Generation Configuration for Model Interface Elements” (Embedded Coder)

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');
```

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` | `Dictionary default` | `ExportedGlobal` | `ImportedExtern` | `ImportedExternPointer` | `Model default`

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`

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 for code generation. Use this function to map the specified model parameter to the storage class and storage class property settings that the code generator uses to produce C code for the parameter or.

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');
```

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

character vector | string | cell array of character vectors | cell array of string scalars

Name of the model workspace parameter 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` | `Dictionary default` | `ExportedGlobal` | `ImportedExtern` | `ImportedExternPointer` | `Model default`

Storage class to set for the specified model parameter. 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 in the generated code.

Data Types: `char` | `string`

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');
```

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` | `Dictionary default` | `ExportedGlobal` | `ImportedExtern` | `ImportedExternPointer` | `Model default`

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`

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` | `Dictionary default` | `ExportedGlobal` | `ImportedExtern` | `ImportedExternPointer` | `Model default`

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`

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` | `Dictionary default` | `ExportedGlobal` | `ImportedExtern` | `ImportedExternPointer` | `Model default`

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`

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)
```

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.

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');
```

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`

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” (Embedded Coder)

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. Reduce the effort of preparing a model for code generation by specifying default configurations for categories of data elements 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 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

getDataDefault	Get default code settings for data 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” (Embedded Coder)

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
`ExternalParameterObjects` | `GlobalDataStores` | `Inports` | `InternalData` | `ModelParameters` | `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` | `Dictionary default` | `ExportedGlobal` | `ImportedExtern` | `ImportedExternPointer` | `Model default`

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'

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
`ExternalParameterObjects` | `GlobalDataStores` | `Inports` | `InternalData` | `ModelParameters` | `Outports` | `SharedLocalDataStores`

Category of data elements to return a property value for.

Example: `'Inports'`

`property` — Code mapping property value to return
`StorageClass` | `Identifier`

Code mapping property that you return a value for. Specify one of these property names.

Information to Return	Property Name
Name of storage class	`StorageClass`
Name of variable for data element in the generated code	`Identifier`
Character vector or string scalar that names a memory section for a model defined in the Embedded Coder Dictionary.	`MemorySection`

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** on page 12-44.

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` (Embedded Coder) | `isBreakpointDataInterface` (Embedded Coder)

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` (Embedded Coder) | `coder.descriptor.BreakpointDataInterface` (Embedded Coder) | `coder.descriptor.DataInterface` (Embedded Coder)

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: [1x0 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`

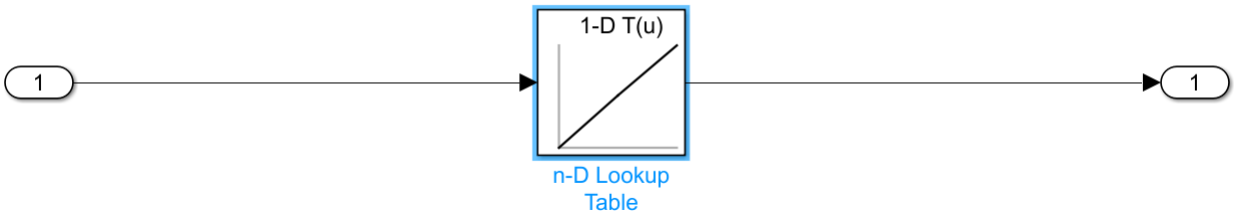
Specify the `Parameters` data interface type.

Example: `Parameters`

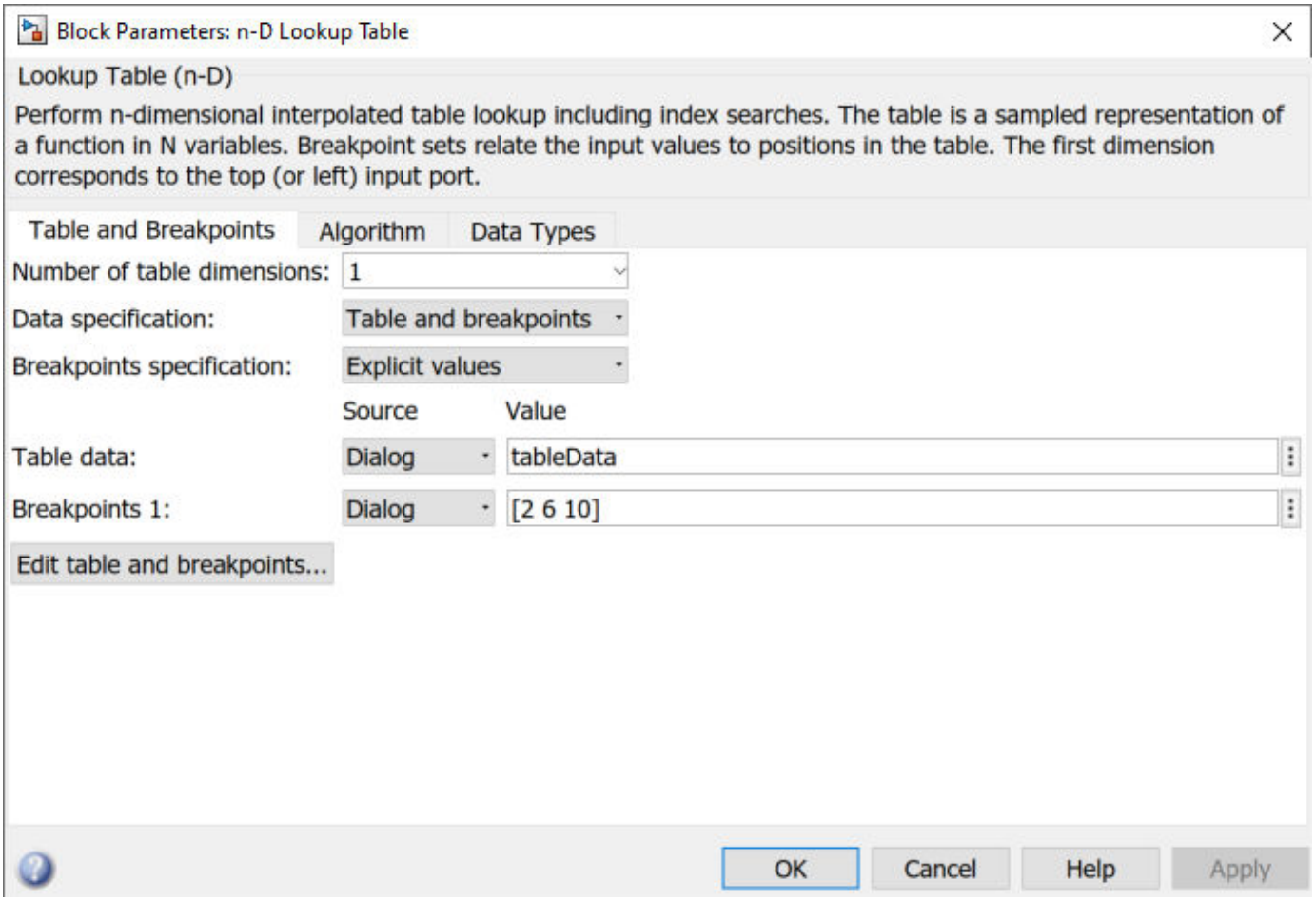
Examples

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]`.



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

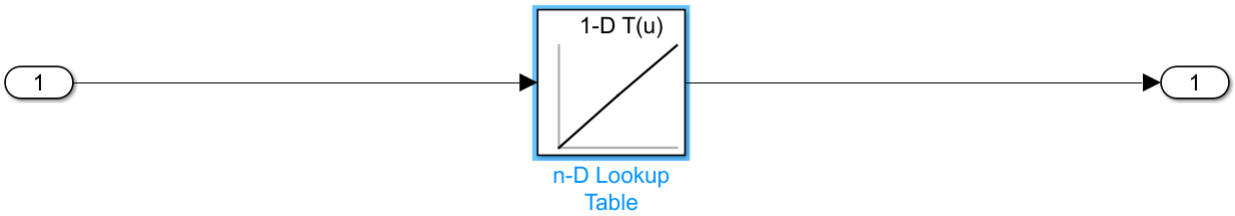
Returns `1` if the value in `StepValue` is a power of 2.

Data Types: `logical`

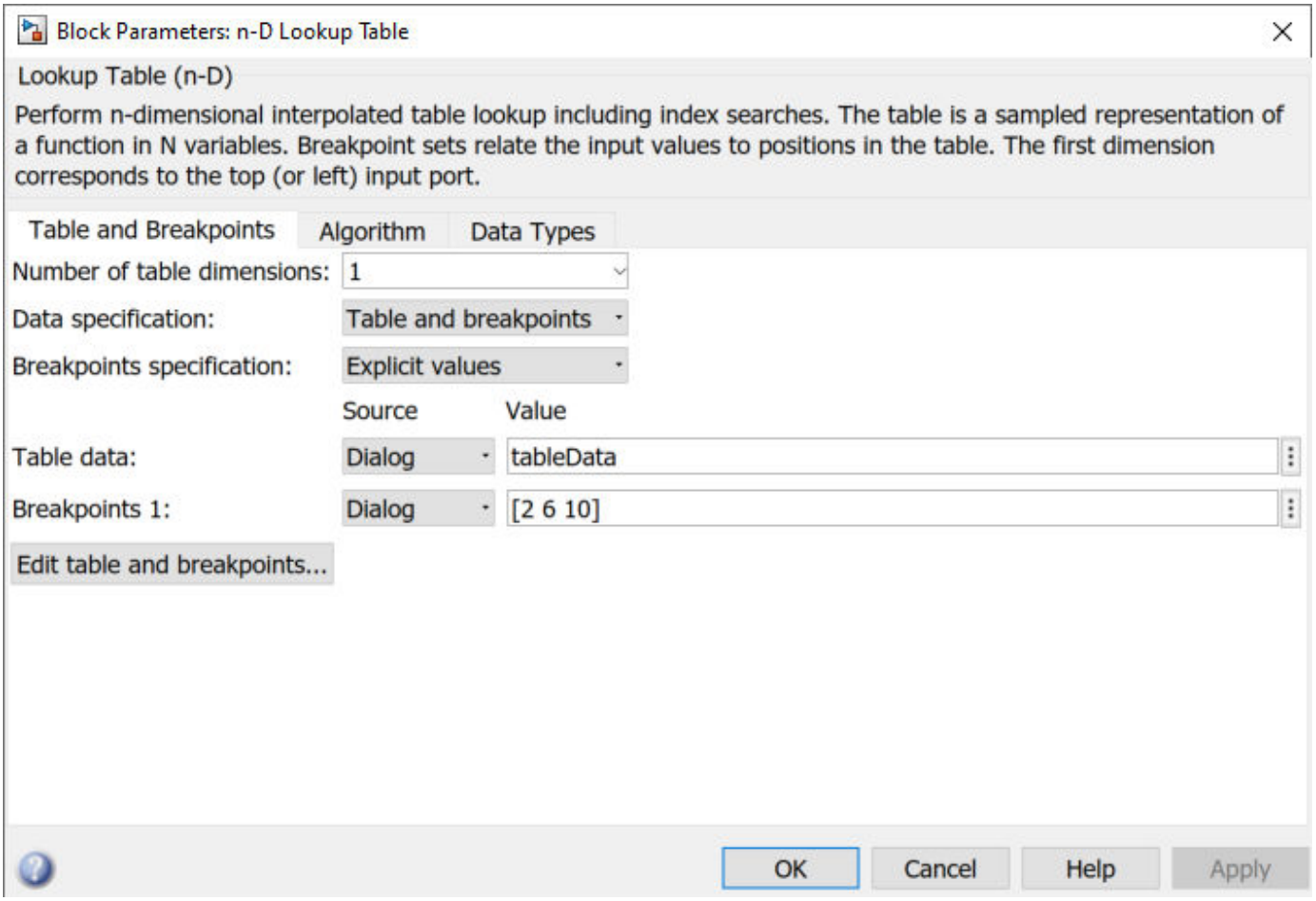
Examples

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

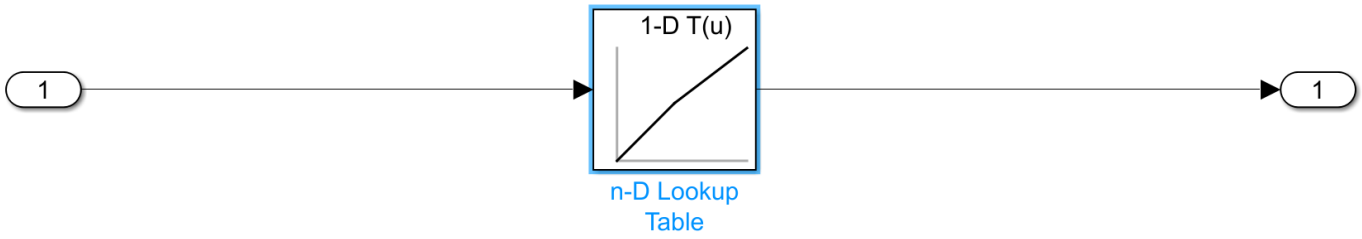
`double vector`

All values that are in the non-evenly spaced breakpoint set data.

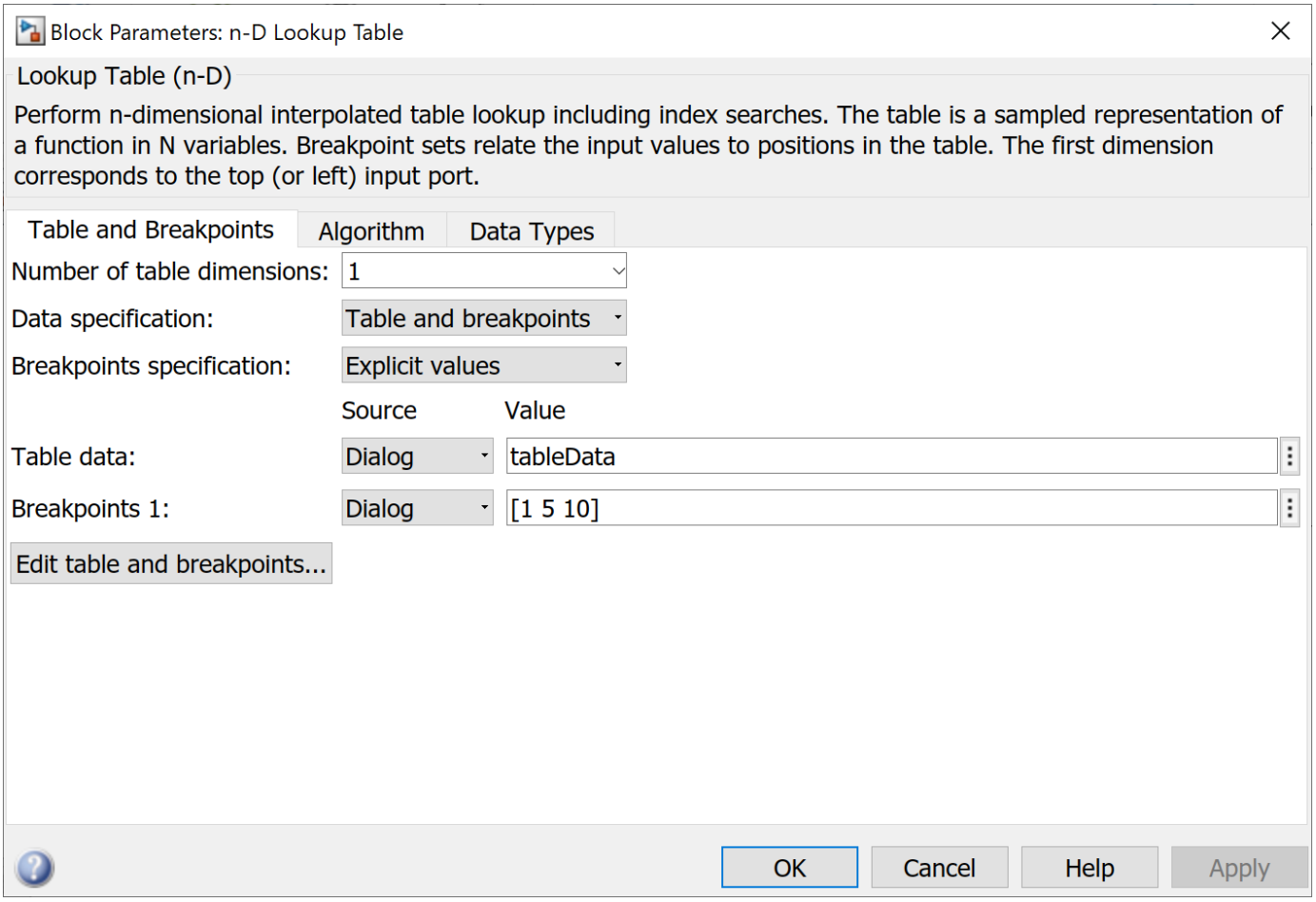
Examples

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]`.



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" (Embedded Coder).

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" (Embedded Coder).

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" (Embedded Coder).

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" (Embedded Coder).

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" (Embedded Coder).

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" (Embedded Coder).

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" (Embedded Coder).

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" (Embedded Coder).

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 on page 15-11 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” on page 12-17.

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” (Embedded Coder), and “Configure Default Code Generation for Functions” (Embedded Coder).

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” (Embedded Coder).

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” (Embedded Coder)
“Configure Default Code Generation for Functions” (Embedded Coder)
“Analyze the Generated Code Interface” (Embedded Coder)
“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” on page 12-17.

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” (Embedded Coder), “Interactively Configure C++ Interface” (Embedded Coder), and “Configure Default Code Generation for Functions” (Embedded Coder).

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” (Embedded Coder).

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” (Embedded Coder)
“Interactively Configure C++ Interface” (Embedded Coder)
“Configure Default Code Generation for Functions” (Embedded Coder)
“Analyze the Generated Code Interface” (Embedded Coder)
“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" (Embedded Coder).

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” on page 12-17.

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” (Embedded Coder), “Interactively Configure C++ Interface” (Embedded Coder), and “Configure Default Code Generation for Functions” (Embedded Coder).

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” (Embedded Coder).

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” (Embedded Coder)
“Interactively Configure C++ Interface” (Embedded Coder)
“Configure Default Code Generation for Functions” (Embedded Coder)
“Analyze the Generated Code Interface” (Embedded Coder)
“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” on page 12-17. 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” (Embedded Coder).

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” (Embedded Coder), and “Configure Default Code Generation for Functions” (Embedded Coder).

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” (Embedded Coder).

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” (Embedded Coder)
“Configure Default Code Generation for Functions” (Embedded Coder)
“Analyze the Generated Code Interface” (Embedded Coder)
“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...
   ```

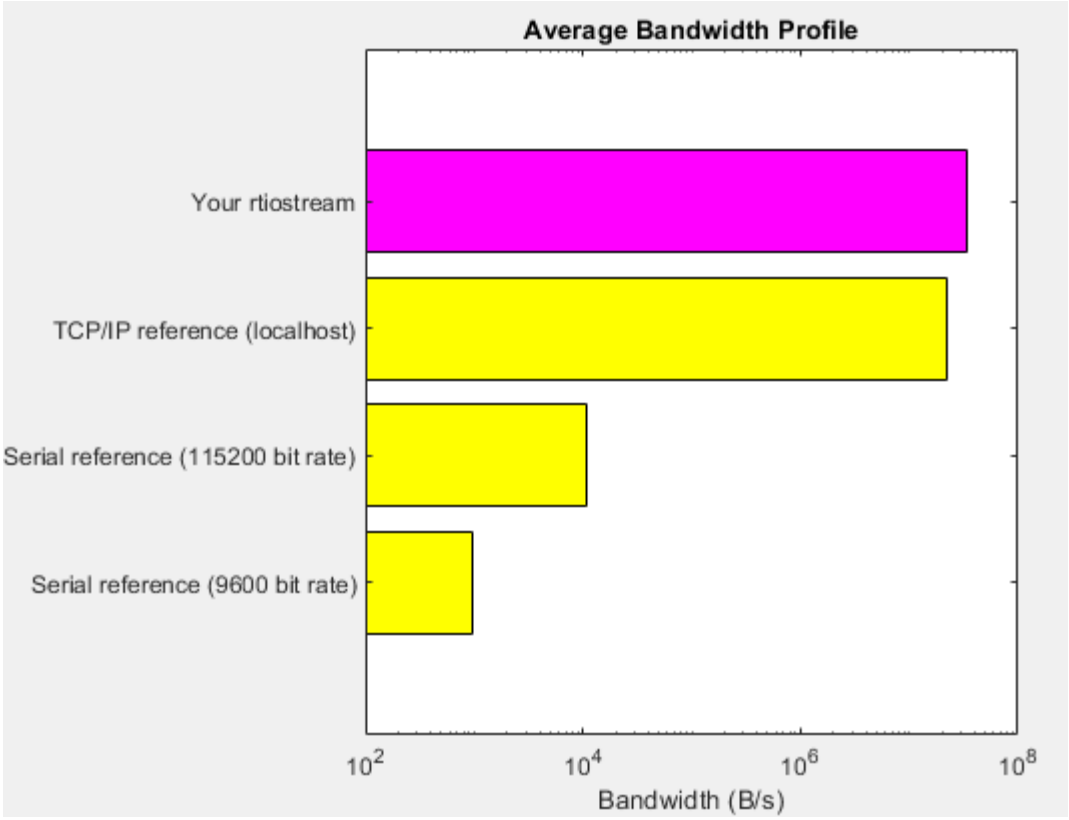
```
### Hardware characteristics discovered
Size of char    : 8 bit
Size of short   : 16 bit
Size of int     : 32 bit
Size of long    : 32 bit
Size of float   : 32 bit
Size of double  : 64 bit
Size of pointer : 64 bit
Byte ordering   : Little Endian

### 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
“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” on page 19-9
“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” on page 19-9
“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`	`rtGetInf.c`	`rtmodel.h`
`rtwdemo_rtwintro.h`	`rtGetInf.h`	
`rtwdemo_rtwintro_private.h`	`rtGetNaN.c`	
`rtwdemo_rtwintrotypes.h`	`rtGetNaN.h`	
	`rt_nonfinite.c`	
	`rt_nonfinite.h`	
	`rtwtypes.h`	
	`multiword_types.h`	
	`builtin_typeid_types.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`	`rtGetInf.c`	`rtmodel.h`
`Amplifier.h`	`rtGetInf.h`	
`Amplifier_private.h`	`rtGetNaN.c`	
`Amplifier_types.h`	`rtGetNaN.h`	
	`rt_nonfinite.c`	
	`rt_nonfinite.h`	
	`rtwtypes.h`	
	`multiword_types.h`	
	`builtin_typeid_types.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” (Embedded Coder)
“Software-in-the-Loop Simulation” (Embedded Coder)

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
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

Topics
“Customize Post-Code-Generation Build Processing”

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** (Embedded Coder) 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.

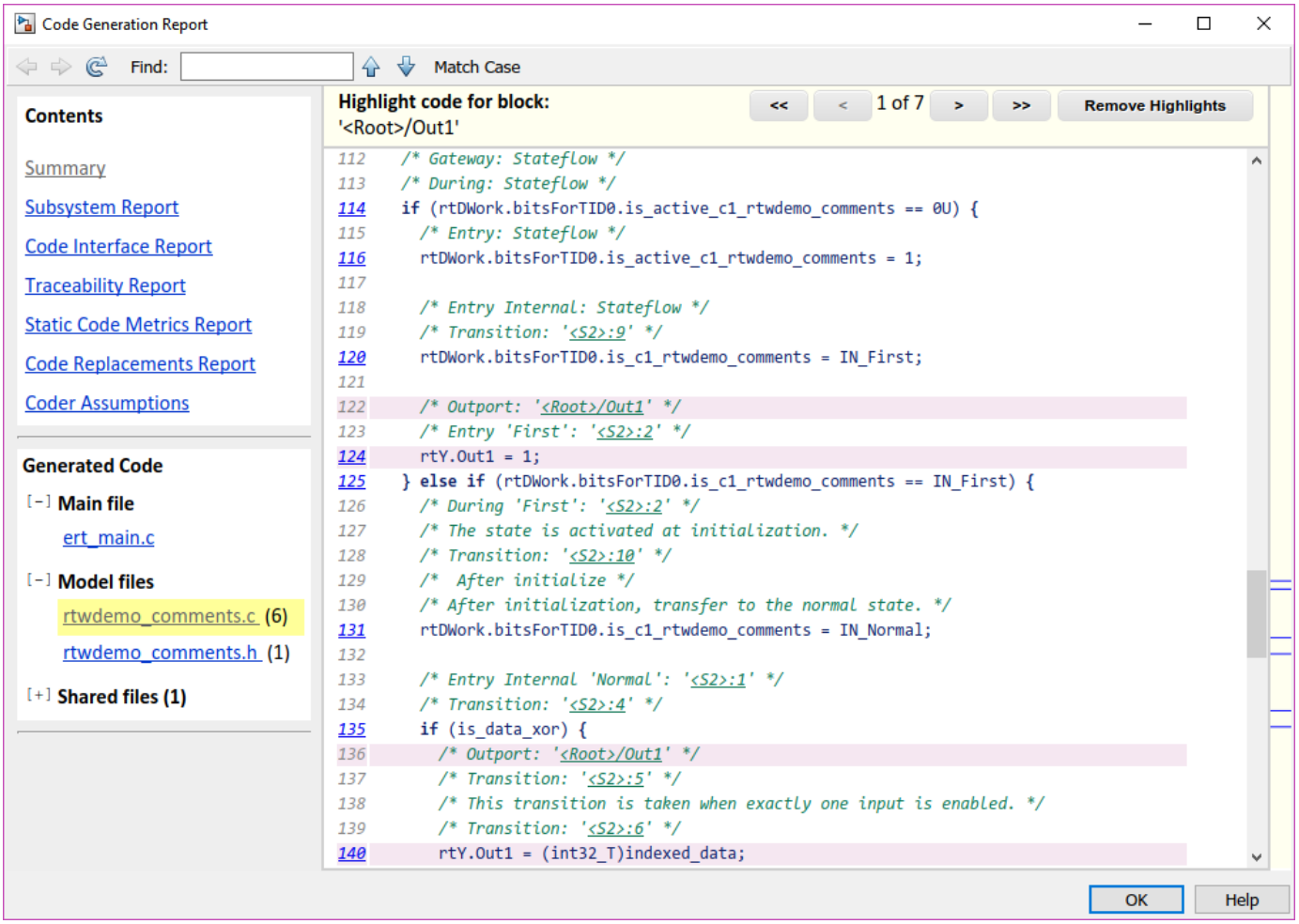
Examples

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')
```



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” (Embedded Coder)
“Model-to-Code Traceability” (Embedded Coder)
“Model-to-code” (Embedded Coder)

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" (Embedded Coder)
"Define Callbacks for Protected Models"
"Reference Protected Models from Third Parties"
"Code Interfaces for SIL and PIL" (Embedded Coder)

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 on page 3-272 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”

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 3-0 .

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” (Embedded Coder)

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” (Embedded Coder).

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” (Embedded Coder) 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” (Embedded Coder) 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” (Embedded Coder) 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” (Embedded Coder)

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” (Embedded Coder).

See Also

`target.DebugIOTool` | `target.create`

Topics

“Use Debugger for PIL Target Connectivity” (Embedded Coder)

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.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 = 'pathTo avrdude';
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', ...
                       'pathTo'avrdude', ...
                       {'-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” (Embedded Coder)

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” (Embedded Coder) 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” (Embedded Coder)

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” (Embedded Coder) 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” (Embedded Coder)

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” (Embedded Coder) 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” (Embedded Coder)

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” (Embedded Coder)

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.Connection`
- `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” (Embedded Coder).

See Also

`target.ExecutionService` | `target.ExecutionTool`

Topics
“Use Debugger for PIL Target Connectivity” (Embedded Coder)
“DebugIOTool Template” (Embedded Coder)

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” (Embedded Coder).

See Also

`target.ApplicationExecutionTool` | `target.ExecutionTool` | `target.create`

Topics

“Use Debugger for PIL Target Connectivity” (Embedded Coder)

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” (Embedded Coder).

See Also

`target.DebugIOTool`

Topics

“Use Debugger for PIL Target Connectivity” (Embedded Coder)

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.Command | target.Command | target.SystemCommandExecutionTool | target.create

Topics

“Set Up PIL Connectivity by Using target Package” (Embedded Coder)

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” (Embedded Coder) 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” (Embedded Coder)

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” (Embedded Coder).

See Also

`target.API` | `target.APIImplementation` | `target.BuildDependencies` | `target.create`

Topics

“Use Debugger for PIL Target Connectivity” (Embedded Coder)

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” (Embedded Coder) 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” (Embedded Coder)

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”

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`

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`

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`

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” (Embedded Coder)

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” (Embedded Coder) 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” (Embedded Coder) 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” (Embedded Coder)

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” (Embedded Coder).

See Also

`target.Board` | `target.create`

Topics

“Use Debugger for PIL Target Connectivity” (Embedded Coder)

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”

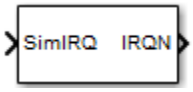
Introduced in R2006a

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

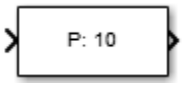
Topics
“Asynchronous Events”
“Asynchronous Events”

Introduced in R2006a

Asynchronous Task Specification

Specify priority of asynchronous task represented by referenced model triggered by asynchronous interrupt

Library: Simulink Coder / Asynchronous



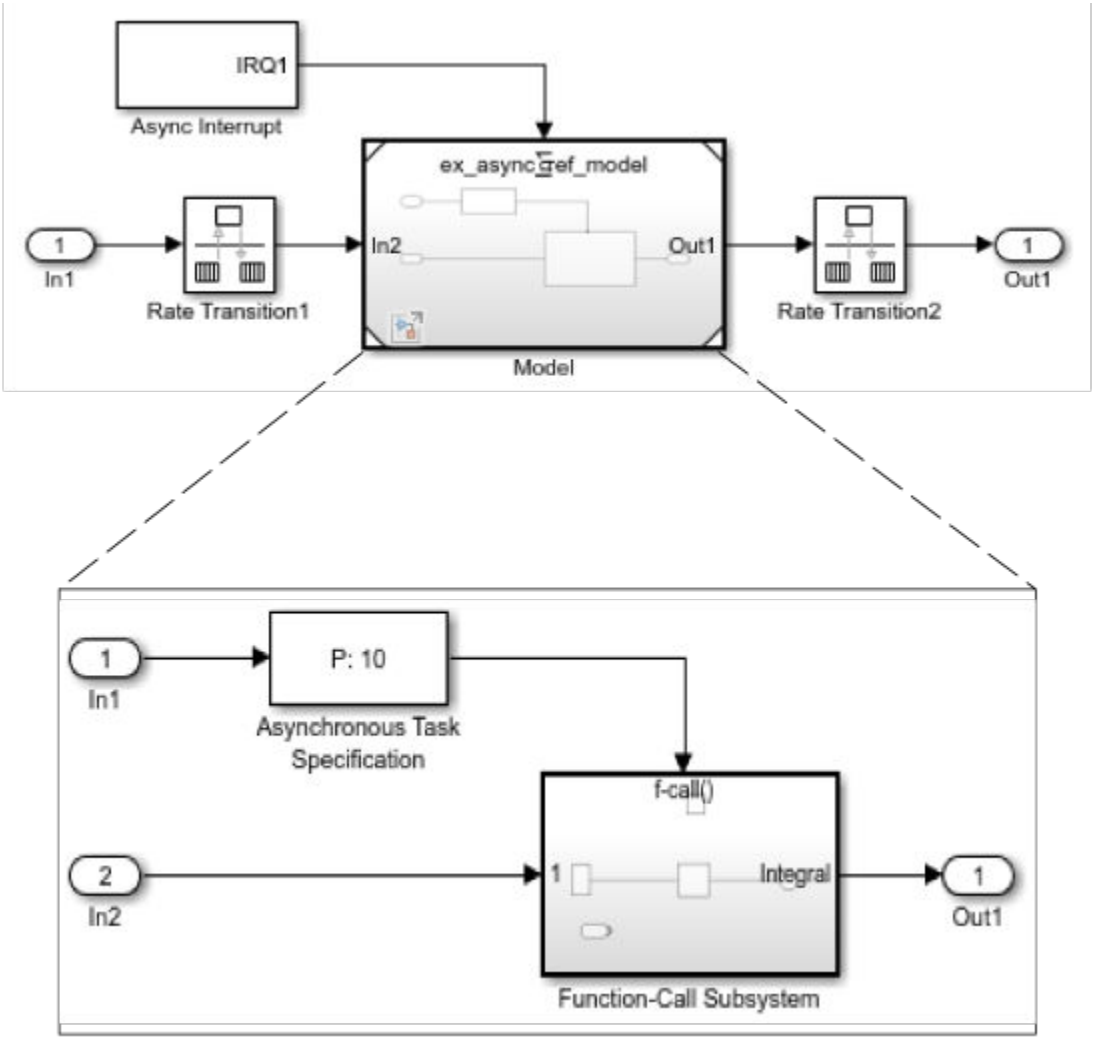
Description

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.

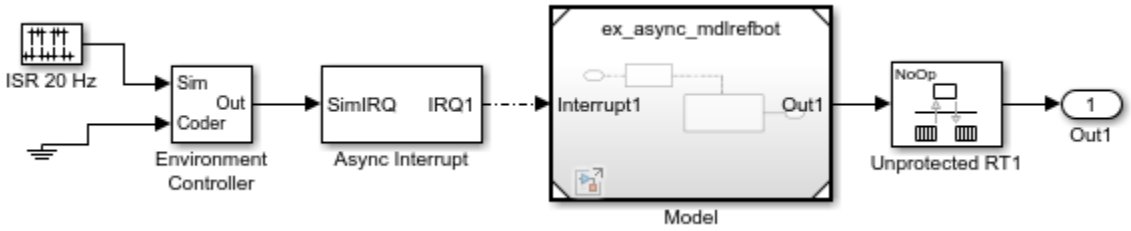
Parameters

`Task priority` — Priority of asynchronous task that calls function-call subsystem
`10` (default)

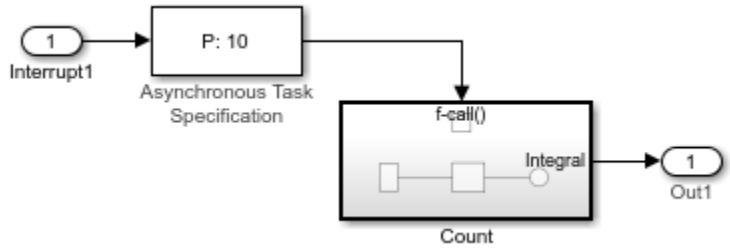
Specify an integer or [] as the priority of the asynchronous task that calls the connected function-call subsystem. The priority must be a value that generates relevant rate transition behaviors.

- 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

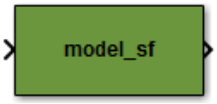
Topics
“Asynchronous Events”
“Spawn and Synchronize Execution of RTOS Task”
“Pass Asynchronous Events in RTOS as Input To a Referenced Model”
“Convert an Asynchronous Subsystem into a Model Reference”
“Rate Transitions and Asynchronous Blocks”
“Asynchronous Support”

“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” (Embedded Coder)

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

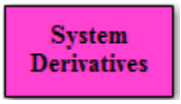
Topics

“Place External C/C++ Code in Generated Code” (Embedded Coder)

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

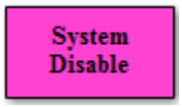
Topics

“Place External C/C++ Code in Generated Code” (Embedded Coder)

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

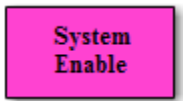
Topics

“Place External C/C++ Code in Generated Code” (Embedded Coder)

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

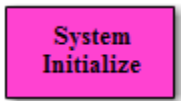
Topics

“Place External C/C++ Code in Generated Code” (Embedded Coder)

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

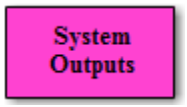
Topics

“Place External C/C++ Code in Generated Code” (Embedded Coder)

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

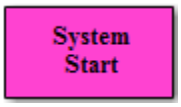
Topics

“Place External C/C++ Code in Generated Code” (Embedded Coder)

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

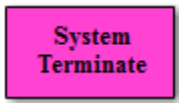
Topics

“Place External C/C++ Code in Generated Code” (Embedded Coder)

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” (Embedded Coder)

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

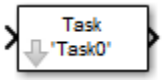
“Place External C/C++ Code in Generated Code” (Embedded Coder)

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

Simulation Target Parameters

- “Target library” on page 5-2
- “Auto tuning” on page 5-3

Target library

Description

Specify the target deep learning library used during simulation. To enable this parameter, select `C++` for the **Language**.

Category: Simulation Target

Settings

Default: `MKL-DNN` if **GPU acceleration** is off. `cuDNN` if **GPU acceleration** is on.

`MKL-DNN`

Use this option for simulation that uses the Intel® Math Kernel Library for Deep Neural Networks (Intel MKL-DNN).

`cuDNN`

Use this option for simulation that uses the CUDA® Deep Neural Network library (cuDNN).

`TensorRT`

Use this option for simulation that takes advantage of the NVIDIA® TensorRT – high performance deep learning inference optimizer and run-time library.

Dependencies

- `MKL-DNN` is available when **GPU acceleration** on the **Simulation Target** pane is disabled and **Language** is set to `C++`.
- `cuDNN` or `TensorRT` requires a GPU Coder™ license.
- `cuDNN` or `TensorRT` is available when **GPU acceleration** on the **Simulation Target** pane is enabled.

Command-Line Information

Parameter: `SimDLTargetLib`
Type: character vector
Value: `'MKL-DNN'` | `'cuDNN'` | `'TensorRT'`
Default: `MKL-DNN`

See Also

Related Examples

- “Model Configuration Parameters: Simulation Target”
- “Deep Learning in Simulink by Using MATLAB Function Block” (GPU Coder)

Auto tuning

Description

Use auto tuning for cuDNN library.

Category: Simulation Target

Settings

Default: On

On

Use this option to enable auto tuning feature. Enabling auto tuning allows the cuDNN library to find the fastest convolution algorithms. This increases performance for larger networks such as `SegNet` and `ResNet`.

Off

Disable auto tuning for the cuDNN library.

Dependencies

- This parameter requires a GPU Coder license.
- To enable this parameter, select `C++` for the **Language** and `cuDNN` for **Target library**.

Command-Line Information

Parameter: `SimDLAutoTuning`
Value: `'on' | 'off'`
Default: `'on'`

See Also

Related Examples

- “Model Configuration Parameters: Simulation Target”
- “Deep Learning in Simulink by Using MATLAB Function Block” (GPU Coder)

GPU Acceleration Parameters

- “Model Configuration Parameters: GPU Acceleration” on page 6-2
- “Simulation Target: GPU Acceleration Tab Overview” on page 6-3
- “Custom compute capability” on page 6-4
- “Dynamic memory allocation threshold” on page 6-5
- “Stack size per GPU thread” on page 6-6
- “Include error checks in generated code” on page 6-7
- “Additional compiler flags” on page 6-8

Model Configuration Parameters: GPU Acceleration

The **Simulation Target > GPU Acceleration** category includes parameters for configuring GPU-specific settings of the generated code.

These parameters require a GPU Coder license.

On the Configuration Parameters dialog box, the following configuration parameters are on the **Simulation Target > GPU Acceleration** pane.

Parameter	Description
"Custom compute capability" on page 6-4	Specify the name of the NVIDIA virtual GPU architecture for code generation.
"Dynamic memory allocation threshold" on page 6-5	Specify the size above which the private variables are allocated on the heap instead of the stack.
"Stack size per GPU thread" on page 6-6	Specify the maximum stack limit per GPU thread.
"Include error checks in generated code" on page 6-7	Add run-time error-checking functionality to the generated CUDA code.
"Additional compiler flags" on page 6-8	Specify additional flags to the `nvcc` compiler.

See Also

More About

- "Model Configuration Parameters: Simulation Target"
- "Simulation Acceleration by Using GPU Coder" (GPU Coder)

Simulation Target: GPU Acceleration Tab Overview

Set up GPU-specific information about simulation acceleration for a model's active configuration set, including device memory settings, code profiling and analysis, and device architecture and compiler.

These parameters require a GPU Coder license.

See Also

More About

- “Model Configuration Parameters: GPU Acceleration” on page 6-2
- “Simulation Acceleration by Using GPU Coder” (GPU Coder)

Custom compute capability

Description

Specify the name of the NVIDIA virtual GPU architecture for code generation.

Category: Simulation Target > GPU Acceleration

Settings

Default: `''`

Specify the name of the NVIDIA virtual GPU architecture for which the CUDA input files must be compiled.

For example, to specify a virtual architecture type `-arch=compute_50`. You can specify a real architecture using `-arch=sm_50`. For more information, see the *Options for Steering GPU Code Generation* topic in the CUDA toolkit documentation.

Dependencies

- This parameter requires a GPU Coder license.
- This parameter is enabled by **GPU acceleration** on the **Simulation Target** pane.

Command-Line Information

Parameter: `SimGPUCustomComputeCapability`
Type: character vector
Value: `''` or a valid `user-specified` virtual architecture specification
Default: `''`

See Also

Related Examples

- “Model Configuration Parameters: GPU Acceleration” on page 6-2
- “Simulation Acceleration by Using GPU Coder” (GPU Coder)

Dynamic memory allocation threshold

Description

Specify the memory allocation threshold.

Category: Simulation Target > GPU Acceleration

Settings

Default: 200

Specify the size above which the private variables are allocated on the heap instead of the stack, as an integer value.

Dependencies

- This parameter requires a GPU Coder license.
- This parameter is enabled by **GPU acceleration** on the **Simulation Target** pane.

Command-Line Information

Parameter: `SimGPUMallocThreshold`
Type: integer
Value: any valid value
Default: `200`

See Also

Related Examples

- “Model Configuration Parameters: GPU Acceleration” on page 6-2
- “Simulation Acceleration by Using GPU Coder” (GPU Coder)

Stack size per GPU thread

Description

Specify the stack limit per GPU thread.

Category: Simulation Target > GPU Acceleration

Settings

Default: 1024

Specify the maximum stack limit per GPU thread as an integer value.

Dependencies

- This parameter requires a GPU Coder license.
- This parameter is enabled by **GPU acceleration** on the **Simulation Target** pane.

Command-Line Information

Command-Line Information
Parameter: `SimGPUStackLimitPerThread`
Type: integer
Value: any valid value
Default: `1024`

See Also

Related Examples

- “Model Configuration Parameters: GPU Acceleration” on page 6-2
- “Simulation Acceleration by Using GPU Coder” (GPU Coder)

Include error checks in generated code

Description

Add run-time error-checking functionality to the generated CUDA code.

Category: Simulation Target > GPU Acceleration

Settings

Default: Off

On

Generates code with error-checking for CUDA API and kernel calls and performs run-time checks.

Off

The generated CUDA code does not contain error-checking functionality.

Dependencies

- This parameter requires a GPU Coder license.
- This parameter is enabled by **GPU acceleration** on the **Simulation Target** pane.

Command-Line Information

Parameter: `SimGPUErrorChecks`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

Related Examples

- “Model Configuration Parameters: GPU Acceleration” on page 6-2
- “Simulation Acceleration by Using GPU Coder” (GPU Coder)

Additional compiler flags

Description

Specify additional flags to the NVIDIA `nvcc` compiler.

Category: Simulation Target > GPU Acceleration

Settings

Default: `''`

Pass additional flags to the GPU compiler. For example, `--fmad=false` instructs the `nvcc` compiler to disable contraction of floating-point multiply and add to a single Floating-Point Multiply-Add (FMAD) instruction.

For similar NVIDIA compiler options, see the topic on *NVCC Command Options* in the CUDA toolkit documentation.

Dependencies

- This parameter requires a GPU Coder license.
- This parameter is enabled by **GPU acceleration** on the **Simulation Target** pane.

Command-Line Information

Parameter: `SimGPUCompilerFlags`
Type: character vector
Value: `''` or a valid `user-specified` flag
Default: `''`

See Also

Related Examples

- “Model Configuration Parameters: GPU Acceleration” on page 6-2
- “Simulation Acceleration by Using GPU Coder” (GPU Coder)

Code Generation Parameters: Code Generation

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” on page 7-7	Specify which target file configuration will be used.
“Browse” on page 7-9	Browse file configuration options.
“Language” on page 7-10	Specify C or C++ code generation.
“Generate GPU code” on page 7-12	Use GPU Coder for CUDA code generation. This parameter requires a GPU Coder license.
“Description” on page 7-13	A description of the target file.
“Generate code only” on page 7-36	Specify code generation versus an executable build.
“Package code and artifacts” on page 7-37	Specify whether to automatically package generated code and artifacts for relocation.
“Zip file name” on page 7-39	Specify the name of the `.zip` file in which to package generated code and artifacts for relocation.
“Compiler optimization level” on page 7-19	Control compiler optimizations for building generated code.
“Custom compiler optimization flags” on page 7-21	Specify custom compiler optimization flags.
“Toolchain” on page 7-14	Specify the toolchain to use when building an executable or library.
“Build configuration” on page 7-16	Specify compiler optimization or debug settings for toolchain.
“Tool/Options” on page 7-18	Display or customize build configuration settings.
“Generate makefile” on page 7-22	Enable generation of a makefile based on a template makefile.
“Make command” on page 7-24	Specify a make command and optionally append makefile options.
“Template makefile” on page 7-26	Specify the template makefile from which to generate the makefile.
“Select objective” on page 7-28	Select a code generation objective to use with the Code Generation Advisor.
“Prioritized objectives” on page 7-30	List of prioritized code generation objectives.

Parameter	Description
“Set Objectives” on page 7-32	Open Configuration Set Objectives dialog box.
“Set Objectives — Code Generation Advisor Dialog Box” (Embedded Coder)	Select and prioritize code generation objectives.
“Check model before generating code” on page 7-34	Choose whether to run Code Generation Advisor checks before generating code.
“Check Model” on page 7-33	Check whether the model meets code generation objectives.

These configuration parameters are under the **Advanced parameters**.

Parameter	Description
“Custom FFT library callback” on page 7-40	Specify a callback class for FFTW library calls in code generated for FFT functions in MATLAB code.
“Custom BLAS library callback” on page 7-42	Specify BLAS library callback class for BLAS calls in code generated from MATLAB code.
“Custom LAPACK library callback” on page 7-44	Specify LAPACK library callback class for LAPACK calls in code generated from MATLAB code.
“Verbose build” on page 7-46	Display code generation progress.
“Retain .rtw file” on page 7-47	Specify *`model`*`.rtw` file retention.
“Profile TLC” on page 7-48	Profile the execution time of TLC files.
“Enable TLC assertion” on page 7-51	Produce the TLC stack trace.
“Start TLC coverage when generating code” on page 7-50	Generate the TLC execution report.
“Start TLC debugger when generating code” on page 7-49	Specify use of the TLC debugger
“Show Custom Hardware App in Simulink Toolstrip” on page 7-52	Read-only internal parameter for Simulink toolstrip.
“Show Embedded Hardware App in Simulink Toolstrip” on page 7-53	Read-only internal parameter for Simulink toolstrip.
“Package” (Embedded Coder)	Specify a package that contains memory sections you want to apply to model-level functions and internal data.
“Refresh package list” (Embedded Coder)	Add user-defined packages that are on the search path to list of packages.
“Initialize/Terminate” (Embedded Coder)	Specify whether to apply a memory section to Initialize/Start and Terminate functions.
“Execution” (Embedded Coder)	Specify whether to apply a memory section to execution functions.
“Shared utility” (Embedded Coder)	Specify whether to apply memory sections to shared utility functions.

Parameter	Description
“Constants” (Embedded Coder)	Specify whether to apply a memory section to constants.
“Inputs/Outputs” (Embedded Coder)	Specify whether to apply a memory section to root input and output.
“Internal data” (Embedded Coder)	Specify whether to apply a memory section to internal data.
“Parameters” (Embedded Coder)	Specify whether to apply a memory section to parameters.
“Validation results” (Embedded Coder)	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” on page 19-2
- “Model Configuration Parameters: Code Generation Report” on page 8-2
- “Model Configuration Parameters: Comments” on page 9-2
- “Model Configuration Parameters: Code Generation Identifiers” on page 10-2
- “Model Configuration Parameters: Code Generation Custom Code” on page 11-2
- “Model Configuration Parameters: Code Generation Interface” on page 12-2

See Also

More About

- “Model Configuration”
- “Control Data and Function Placement in Memory by Inserting Pragmas” (Embedded Coder)

Code Generation: General Tab Overview

Set up general information about code generation for a model's active configuration set, including target selection, documentation, and build process parameters.

To open the **Code Generation** pane, select **Configuration Parameters > Code Generation**.

To get help on an option

1. Right-click the option's text label.
2. Select **What's This** from the popup menu.

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 7-2

System target file

Description

Specify the system target file.

Category: Code Generation

Settings

Default: `grt.tlc`

You can specify the system target file in these ways:

- Use the System Target File Browser. Click the **Browse** button, which lets you select a preset target configuration consisting of a system target file, template makefile, and make command.
- Enter the name of your system target file in this field.

Tips

- The System Target File Browser lists system target files found on the MATLAB path. Some system target files require additional licensed products.
- Using ERT-based system target files such as `ert.tlc` to generate code requires an Embedded Coder license.
- When you switch from a system target file that is not ERT-based to a file that is ERT-based, the configuration parameter **Default parameter behavior** sets to `Inlined` by default. However, you can change the setting of **Default parameter behavior** later. For more information, see “Default parameter behavior” on page 19-9.
- To configure your model for rapid simulation, select `rsim.tlc`.
- To configure your model for Simulink Real-Time™, select `slrealtime.tlc`.

Command-Line Information

Parameter: `SystemTargetFile`
Type: character vector
Value: valid system target file
Default: `'grt.tlc'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact ERT based (requires Embedded Coder license)

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 7-2
- “Compare System Target File Support Across Products”

Browse

Description

Open the System Target File Browser, which lets you select a preset target configuration consisting of a system target file, template makefile, and make command. The value you select is filled into **“System target file” on page 7-7**.

Category: Code Generation

Tips

- The System Target File Browser lists system target files found on the MATLAB path. Some system target files require additional licensed products, such as the Embedded Coder product.
- To configure your model for rapid simulation, select `rsim.tlc`.
- To configure your model for Simulink Real-Time, select `slrealtime.tlc`.

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 7-2
- “Configure a System Target File”
- “Compare System Target File Support Across Products”

Language

Description

Specify C or C++ code generation.

Category: Code Generation

Settings

Default: `C`

`C`

Generates C code and places the generated files in your build folder.

`C++`

Generates C++ code and places the generated files in your build folder.

On the **Code Generation** > **Interface** pane, if you additionally set the **Code interface packaging** parameter to `C++ class`, the build generates a C++ class interface to model code. The generated interface encapsulates required model data into C++ class attributes and model entry point functions into C++ class methods.

If you set **Code interface packaging** to a value other than `C++ class`, the build generates C++ compatible `.cpp` files containing model interfaces enclosed within an `extern "C"` link directive.

You might need to configure the Simulink Coder software to use a compiler before you build a system.

Dependencies

Selecting `C++` enables and selects the value `C++ class` for the **Code interface packaging** parameter on the **Code Generation** > **Interface** pane.

Command-Line Information

Parameter: `TargetLang`
Type: character vector
Value: `'C'` | `'C++'`
Default: `'C'`

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” on page 7-2
- “Select C or C++ Programming Language”
- “Select and Configure C or C++ Compiler”
- “Configure C Code Generation for Model Entry-Point Functions” (Embedded Coder)

Generate GPU code

Description

Use GPU Coder for CUDA code generation.

Category: Code Generation

Settings

Default: Off

On

Enables code generation by using GPU Coder.

Off

Disables code generation by using GPU Coder.

Dependencies

- This parameter requires a GPU Coder license.
- To enable this parameter, select `C++` for the **Language** and `grt.tlc` or `ert.tlc` for **System target file**.
- This parameter enables **Support long long**.

Command-Line Information

Parameter: `GenerateGPUCode`
Type: character vector
Value: `'CUDA' | 'none'`
Default: `'CUDA'`

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 7-2
- “Code Generation from Simulink Models with GPU Coder” (GPU Coder)

Description

Description

This field displays the description of the system target file. You can use this description to differentiate between two system target files that have the same file name. To change the value of this description, click the Browse button.

Category: Code Generation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 7-2
- “Browse” on page 7-9

Toolchain

Description

Specify the toolchain to use when building an executable or library.

Note This parameter only appears when the model is configured to use a toolchain-based code generation target, as described in “Choose Build Approach and Configure Build Process”.

Category: Code Generation

Settings

Default: `Automatically locate an installed toolchain`

The list of available toolchains depends on the host computer platform, and can include custom toolchains that you added.

When **Toolchain** is set to `Automatically locate an installed toolchain`, the code generator:

1. Searches your host computer for installed toolchains.
2. Selects the most current toolchain.
3. Displays the name of the selected toolchain immediately below the drop down menu.

Tip

Click the **Configuration Parameters** > **Code Generation** > **Advanced parameters** > **Toolchain** > **Validate Toolchain** button to verify that the registration information for the toolchain is valid. When the validation process is complete, a separate **Validation report** window opens and displays the results. The Validation report states whether the toolchain registration Passed or Failed and provides status for each step and build tool in the validation process. If the tool chain definition omits a build tool, validation skips the unspecified tool. To fix a failure (for example, the build tool definition omits a required build tool such as compiler or linker), edit the toolchain definition and repeat the registration process.

Command-Line Information

Parameter: `Toolchain`
Type: character vector
Value: `'Automatically locate an installed toolchain'` | A valid toolchain name
Default: `'Automatically locate an installed toolchain'`

Recommended Settings

Application	Setting
Debugging	No impact

Application	Setting
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 7-2
- “Toolchain Configuration”
- “Add Custom Toolchains to MATLAB® Coder™ Build Process”

Build configuration

Description

Specify compiler optimization or debug settings for toolchain.

Note This parameter only appears when the model is configured to use a toolchain-based code generation target, as described in “Choose Build Approach and Configure Build Process”.

Category: Code Generation

Settings

Default: `Faster Builds`

`Faster Builds`

Optimize for shorter build times.

`Faster Runs`

Optimize for faster-running executable.

`Debug`

Optimize for debugging.

`Specify`

Selecting `Specify` displays a table of tools with editable options. Use this table to customize settings for the current model. See “Tool/Options” on page 7-18.

This interaction helps synchronize the **Toolchain** value and manually specified **Build configuration** values.

Modifying the **Build configuration** value can change the **Toolchain** value. The `Automatically locate an installed toolchain` is the only value for **Toolchain** that is affected by changing the **Build configuration** to `Specify`.

- Changing the **Build configuration** to `Specify`, changes the **Toolchain** value `Automatically locate an installed toolchain` (default) to the value of the toolchain that was located (for example, `Microsoft Visual C++ 2012 v11.0 |(64-bit Windows)`).
- Changing the **Build configuration** from `Specify` to another value does not change the **Toolchain** value.

Tip

Click **Show settings** to display a table of tools with options for the current build configuration. See “Tool/Options” on page 7-18.

Customize the toolchain options for the `Specify` build configuration. These options only apply to the current project.

To extract macro definitions (including compiler optimization flags) from the generated makefile for toolchain approach builds on Windows or UNIX systems, see the *`model`*`.bat` description in “Manage Build Process Files”.

Dependencies

Selecting `Specify` displays a table of tools with editable options. Use this table to customize settings for the current model. See “Tool/Options” on page 7-18.

Command-Line Information

Parameter: `BuildConfiguration`
Type: character vector
Value: `'Faster Builds'` | `'Faster Runs'` | `'Debug'` | `'Specify'`
Default: `'Faster Builds'`

Recommended Settings

Application	Setting
Debugging	`Debug`
Traceability	No impact
Efficiency	`Faster Runs`
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 7-2
- “Toolchain Configuration”
- “Add Custom Toolchains to MATLAB® Coder™ Build Process”
- “Control Compiler Optimizations”

Tool/Options

Description

Display or customize build configuration settings.

Note These parameters only appear when the model is configured to use the toolchain approach, as described in “Choose Build Approach and Configure Build Process”

Category: Code Generation

Settings

The tools column can include: Assembler, C Compiler, Linker, Shared Library Linker, C++ Compiler, C ++ Linker, C++ Shared Library Linker, Archiver, Download, Execute, Make Tool. The options can vary by tool and toolchain and can contain macros. Consult third-party toolchain documentation for more information about options you can use with a specific tool.

Dependencies

To display a table of tools and options for the current build configuration, click **Show settings**, next to **Build configuration**.

To create a custom build configuration by editing a table of Tool/Options, set **Build configuration** to `Specify`.

Command-Line Information

Parameter: `CustomToolchainOptions`
Type: character vector
Value: Specify the baseline toolchain settings. Use a new-line-delineated character vector to specify each option and its values.
Default: `''`

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 7-2
- “Toolchain Configuration”
- “Add Custom Toolchains to MATLAB® Coder™ Build Process”

Compiler optimization level

Description

Control compiler optimizations for building generated code, using flexible, generalized controls.

Note This parameter only appears when the model is configured to use a template makefile-based code generation target, as described in “Choose Build Approach and Configure Build Process”.

Category: Code Generation

Settings

Default: `Optimizations off (faster builds)`

`Optimizations off (faster builds)`

Customizes compilation during the build process to minimize compilation time.

`Optimizations on (faster runs)`

Customizes compilation during the makefile build process to minimize run time.

`Custom`

Allows you to specify custom compiler optimization flags to be applied during the makefile build process.

Tips

- Target-independent values `Optimizations on (faster runs)` and `Optimizations off (faster builds)` allow you to easily toggle compiler optimizations on and off during code development.
- `Custom` allows you to enter custom compiler optimization flags at Simulink GUI level, rather than editing compiler flags into template makefiles (TMFs) or supplying compiler flags to make commands.
- If you specify compiler options for your makefile build using `OPT_OPTS`, `MEX_OPTS` (except `MEX_OPTS="-v"`), or `MEX_OPT_FILE`, the value of **Compiler optimization level** is ignored and a warning is issued about the ignored parameter.

Dependencies

This parameter enables **Custom compiler optimization flags**.

Command-Line Information

Parameter: `RTWCompilerOptimization`
Type: character vector
Value: `'off'` | `'on'` | `'custom'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	`Optimizations off (faster builds)`
Traceability	`Optimizations off (faster builds)`
Efficiency	`Optimizations on (faster runs)` (execution), No impact (ROM, RAM)
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 7-2
- “Custom compiler optimization flags” on page 7-21
- “Control Compiler Optimizations”

Custom compiler optimization flags

Description

Specify compiler optimization flags to be applied to building the generated code for your model.

Note This parameter only appears when the model is configured to use a template makefile-based code generation target, as described in “Choose Build Approach and Configure Build Process”.

Category: Code Generation

Settings

Default: `''`

Specify compiler optimization flags without quotes, for example, `-O2`.

Dependency

This parameter is enabled by selecting the value `Custom` for the parameter **Compiler optimization level**.

Command-Line Information

Parameter: `RTWCustomCompilerOptimizations`
Type: character vector
Value: `''` | user-specified flags
Default: `''`

Recommended Settings

See “Compiler optimization level” on page 7-19.

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 7-2
- “Compiler optimization level” on page 7-19
- “Control Compiler Optimizations”

Generate makefile

Description

Enable generation of a makefile based on a template makefile.

Note This option only appears when the model is configured to use a template makefile-based code generation target, as described in “Choose Build Approach and Configure Build Process”.

Category: Code Generation

Settings

Default: on

 On

Generates a makefile for a model during the build process.

 Off

Suppresses the generation of a makefile. You must set up post code generation build processing, including compilation and linking, as a user-defined command.

Dependencies

This parameter enables:

- **Make command**
- **Template makefile**

Command-Line Information

Parameter: `GenerateMakefile`
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” on page 7-2
- “Customize Post-Code-Generation Build Processing”
- “Customize Build Process with STF_make_rtw_hook File”
- “Target Development and the Build Process”

Make command

Description

Specify a make command and optionally append makefile options.

Note This parameter only appears when the model is configured to use a template makefile-based code generation target, as described in “Choose Build Approach and Configure Build Process”.

Category: Code Generation

Settings

Default: `make_rtw`

An internal MATLAB command used by code generation software to control the build process. The specified make command is invoked when you start a build.

- Each target has an associated make command, automatically supplied when you select a target file using the System Target File Browser.
- Some third-party targets supply a make command. See the vendor's documentation.
- You can supply makefile options in the **Make command** field. The options are passed to the command-line invocation of the `make` utility, which adds them to the overall flags passed to the compiler. Append the options after the make command, as in the following example:

  ```
  make_rtw OPTS="-DMYDEFINE=1"
  ```

  The syntax for makefile options differs slightly for different compilers.

Tip

- Most targets use the default command.
- You should not invoke `make_rtw` or other internal make commands directly from MATLAB code. To initiate a model build from MATLAB code, use documented build commands such as `slbuild` or `slbuild`.

Dependency

This parameter is enabled by **Generate makefile**.

Command-Line Information

Parameter: `MakeCommand`
Type: character vector
Value: valid make command MATLAB language file
Default: `'make_rtw'`

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” on page 7-2
- “Template Makefiles and Make Options”
- “Customize Build Process with STF_make_rtw_hook File”
- “Target Development and the Build Process”

Template makefile

Description

Specify the template makefile from which to generate the makefile.

Note This parameter only appears when the model is configured to use a template makefile-based code generation target, as described in “Choose Build Approach and Configure Build Process”.

Category: Code Generation

Settings

Default: `grt_default_tmf`

The template makefile determines which compiler runs, during the make phase of the build, to compile the generated code. You can specify template makefiles in the following ways:

- Generate a value by selecting a target configuration using the System Target File Browser.
- Explicitly enter a custom template makefile filename (including the extension). The file must be on the MATLAB path.

Tips

- If you do not include a filename extension for a custom template makefile, the code generator attempts to find and execute a MATLAB language file.
- You can customize your build process by modifying an existing template makefile or by providing your own template makefile.

Dependency

This parameter is enabled by **Generate makefile**.

Command-Line Information

Parameter: `TemplateMakefile`
Type: character vector
Value: valid template makefile filename
Default: `'grt_default_tmf'`

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” on page 7-2
- “Template Makefiles and Make Options”
- “Compare System Target File Support Across Products”

Select objective

Description

Select a code generation objective for reviewing your model configuration settings by using the Code Generation Advisor.

Category: Code Generation

Settings

Default: `Unspecified`

`Unspecified`

No objective specified. Do not optimize code generation settings by using the Code Generation Advisor.

`Debugging`

Specifies debugging objective. Optimize code generation settings for debugging the code generation build process by using the Code Generation Advisor.

`Execution efficiency`

Specifies execution efficiency objective. Optimize code generation settings to achieve fast execution time by using the Code Generation Advisor.

Dependency

This parameter name appears only for GRT-based targets. For ERT-based targets, the parameter **Prioritized objectives** lists the code generation objectives.

Command-Line Information

Parameter: `'ObjectivePriorities'`
Type: cell array of character vectors or string array
Value: `{''}` | `{'Debugging'}` | `{'Execution efficiency'}`
Default: `{''}`

Recommended Settings

Application	Setting
Debugging	`Debugging`
Traceability	Not applicable for GRT-based targets
Efficiency	`Execution efficiency`
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 7-2
- “Prioritized objectives” on page 7-30
- “Configure Model for Code Generation Objectives by Using Code Generation Advisor” (Embedded Coder)
- “Application Objectives Using Code Generation Advisor”

Prioritized objectives

Description

List of the objectives that you specify in order of priority for reviewing your model configuration settings by using the Code Generation Advisor. To specify objectives, click the **Set Objectives** button and use the Set Objectives - Code Generation Advisor dialog box.

Category: Code Generation

Settings

Default: `Unspecified`

`Unspecified`

No objective specified. Do not optimize code generation settings using the Code Generation Advisor.

`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.

Dependencies

- This parameter appears only for ERT-based targets. For GRT-based targets, select an objective by using the parameter “Select objective” on page 7-28.
- When you generate code, this parameter requires Embedded Coder.

Command-Line Information

Parameter: `'ObjectivePriorities'`
Type: cell array of character vectors or string array
Value: `{''}` | `{'Execution efficiency'}` | `{'ROM efficiency'}` | `{'RAM efficiency'}` | `{'Traceability'}` | `{'Safety precaution'}` | `{'Debugging'}` | `{'MISRA C:2012 guidelines'}` | `{'Polyspace'}` | cell array of objective names
Default: `{''}`

Recommended Settings

Application	Setting
Debugging	`Debugging`
Traceability	`Traceability`
Efficiency	`Execution efficiency` (execution), `ROM efficiency` (ROM), `RAM efficiency` (RAM)
Safety precaution	`Safety precaution`

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 7-2
- “Set Objectives — Code Generation Advisor Dialog Box” (Embedded Coder)
- “Configure Model for Code Generation Objectives by Using Code Generation Advisor” (Embedded Coder)
- “Application Objectives Using Code Generation Advisor”

Set Objectives

Description

Open Set Objectives - Code Generation Advisor dialog box.

Category: Code Generation

Dependency

This button appears only for ERT-based targets.

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 7-2
- “Configure Model for Code Generation Objectives by Using Code Generation Advisor” (Embedded Coder)
- “Application Objectives Using Code Generation Advisor”

Check Model

Description

Run the Code Generation Advisor checks.

Category: Code Generation

Settings

1. Specify code generation objectives using the **Select objective** parameter (available with GRT-based targets) or in the Configuration Set Objectives dialog box, by clicking **Set Objectives** (available with ERT-based targets).
2. Click **Check Model**. The Code Generation Advisor runs the code generation objectives checks and provide suggestions for changing your model to meet the objectives.

Dependency

You must specify objectives before checking the model.

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 7-2
- “Configure Model for Code Generation Objectives by Using Code Generation Advisor” (Embedded Coder)
- “Application Objectives Using Code Generation Advisor”

Check model before generating code

Description

Choose whether to run Code Generation Advisor checks before generating code.

Category: Code Generation

Settings

Default: `Off`

`Off`

Generates code without checking whether the model meets code generation objectives. The code generation report summary and file headers indicate the specified objectives and that the validation was not run.

`On (proceed with warnings)`

Checks whether the model meets code generation objectives using the Code Generation Objectives checks in the Code Generation Advisor. If the Code Generation Advisor reports a warning, the code generator continues producing code. The code generation report summary and file headers indicate the specified objectives and the validation result.

`On (stop for warnings)`

Checks whether the model meets code generation objectives using the Code Generation Objectives checks in the Code Generation Advisor. If the Code Generation Advisor reports a warning, the code generator does not continue producing code.

Command-Line Information

Parameter: `CheckMdlBeforeBuild`
Type: character vector
Value: `'Off'` | `'Warning'` | `'Error'`
Default: `'Off'`

Recommended Settings

Application	Setting
Debugging	`On (proceed with warnings)` or `On (stop for warnings)`
Traceability	`On (proceed with warnings)` or `On (stop for warnings)`
Efficiency	`On (proceed with warnings)` or `On (stop for warnings)`
Safety precaution	`On (proceed with warnings)` or `On (stop for warnings)`

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 7-2
- “Configure Model for Code Generation Objectives by Using Code Generation Advisor” (Embedded Coder)
- “Application Objectives Using Code Generation Advisor”

Generate code only

Description

Specify code generation versus an executable build.

Category: Code Generation

Settings

Default: off

 On

The build process generates code and a makefile, but it does not invoke the make command.

 Off

The build process generates and compiles code, and creates an executable file.

Tip

Generate code only generates a makefile only if you select **Generate makefile**.

Command-Line Information

Parameter: `GenCodeOnly`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	Off
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 7-2
- “Customize Post-Code-Generation Build Processing”

Package code and artifacts

Description

Specify whether to automatically package generated code and artifacts for relocation.

Category: Code Generation

Settings

Default: off

 On

After code generation, to package generated code and artifacts for relocation, the build process runs this `packNGo` command:

```
packNGo(pwd,...
        'IncludeReport',true,...
        'packType','hierarchical',...
        'fileName',zipName,...
        'nestedZipFiles',false);
```

 Off

The build process does not run `packNGo` after code generation.

Dependency

Selecting this parameter enables **Zip file name** and clearing this parameter disables **Zip file name**.

Command-Line Information

Parameter: `PackageGeneratedCodeAndArtifacts`
Type: character vector
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” on page 7-2
- “Relocate Code to Another Development Environment”
- “Limitations”

Zip file name

Description

Specify the name of the `.zip` file in which to package generated code and artifacts for relocation.

Category: Code Generation

Settings

Default: `''`

You can enter the name of the `zip` file in which to package generated code and artifacts for relocation. The file name can be specified with or without the `.zip` extension. If you do not specify an extension or an extension other than `.zip`, the `zip` utility adds the `.zip` extension. If a value is not specified, the build process uses the name *`model`*`.zip`, where *`model`* is the name of the top model for which code is being generated.

Dependency

This parameter is enabled by **Package code and artifacts**.

Command-Line Information

Parameter: `PackageName`
Type: character vector
Value: valid name for a `.zip` file
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” on page 7-2
- “Relocate Code to Another Development Environment”
- “Limitations”

Custom FFT library callback

Description

Specify a callback class for FFTW library calls in code generated for FFT functions in MATLAB code. This parameter applies to MATLAB code in a MATLAB Function block, a Stateflow chart, or a System object™ associated with a MATLAB System block.

To improve the execution speed of FFT functions, the code generator produces calls to the FFTW library that you specify in the callback class.

Category: Code Generation

Settings

Default: `''`

Specify the name of an FFT library callback class. If this parameter is empty, the code generator uses its own algorithms for FFT functions instead of calling the FFTW library.

Limitation

The class definition file must be in a folder on the MATLAB path.

Tip

Specify only the name of the class. Do not specify the name of the class definition file.

Command-Line Information

Parameter: `CustomFFTCallback`
Type: character vector
Value: class name
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

More About

- “Model Configuration Parameters: Code Generation” on page 7-2

- “Speed Up Fast Fourier Transforms in Code Generated from a MATLAB Function Block”

External Websites

- www.fftw.org

Custom BLAS library callback

Description

Specify BLAS library callback class for BLAS calls in code generated from MATLAB code. This parameter applies to MATLAB code in a MATLAB Function block, a Stateflow chart, or a System object associated with a MATLAB System block.

Category: Code Generation

Settings

Default: `''`

Specify the name of a BLAS callback class that derives from `coder.BLASCallback`. If you specify a BLAS callback class, for certain low-level vector and matrix operations, the code generator produces BLAS calls by using the CBLAS C interface to your BLAS library. The callback class provides the CBLAS header and data type information and the information required to link to your BLAS library. If this parameter is empty, the code generator produces code for the vector and matrix functions instead of a BLAS call.

Limitation

The class definition file must be in a folder on the MATLAB path.

Tip

Specify only the name of the class. Do not specify the name of the class definition file.

Command-Line Information

Parameter: `CustomBLASCallback`
Type: character vector
Value: class name
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” on page 7-2
- “Speed Up Matrix Operations in Code Generated from a MATLAB Function Block”

Custom LAPACK library callback

Description

Specify LAPACK library callback class for LAPACK calls in code generated from MATLAB code. This parameter applies to MATLAB code in a MATLAB Function block, a Stateflow chart, or a System object associated with a MATLAB System block.

Category: Code Generation

Settings

Default: `''`

Specify the name of a LAPACK callback class that derives from `coder.LAPACKCallback`. If you specify a LAPACK callback class, for certain linear algebra functions, the code generator produces LAPACK calls by using the LAPACKE C interface to your LAPACK library. The callback class provides the name of your LAPACKE header file and the information required to link to your LAPACK library. If this parameter is empty, the code generator produces code for linear algebra functions instead of a LAPACK call.

Limitation

The class definition file must be in a folder on the MATLAB path.

Tip

Specify only the name of the class. Do not specify the name of the class definition file.

Command-Line Information

Parameter: `CustomLAPACKCallback`
Type: character vector
Value: class name
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” on page 7-2
- “Speed Up Linear Algebra in Code Generated from a MATLAB Function Block”

Verbose build

Description

Display code generation progress.

Category: Code Generation

Settings

Default: on

 On

The MATLAB Command Window displays progress information indicating code generation stages and compiler output during code generation.

 Off

Does not display progress information.

Command-Line Information

Parameter: `RTWVerbose`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 7-2
- “Configure Model for Debugging”

Retain .rtw file

Description

Specify *model*`.rtw` file retention.

Category: Code Generation

Settings

Default: off

 On

Retains the *model*`.rtw` file in the current build folder. This parameter is useful if you are modifying the target files and need to look at the file.

 Off

Deletes the *model*`.rtw` from the build folder at the end of the build process.

Command-Line Information

Parameter: `RetainRTWFile`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 7-2
- “Configure Model for Debugging”

Profile TLC

Description

Profile the execution time of TLC files.

Category: Code Generation

Settings

Default: off

 On

The TLC profiler analyzes the performance of TLC code executed during code generation, and generates an HTML report.

 Off

Does not profile the performance.

Command-Line Information

Parameter: `ProfileTLC`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 7-2
- “Configure Model for Debugging”

Start TLC debugger when generating code

Description

Specify use of the TLC debugger

Category: Code Generation

Settings

Default: Off

 On

The TLC debugger starts during code generation.

 Off

Does not start the TLC debugger.

Tips

- You can also start the TLC debugger by entering the `-dc` argument into the **System target file** field.
- To invoke the debugger and run a debugger script, enter the `-df` *`filename`* argument into the **System target file** field.

Command-Line Information

Parameter: `TLCDebug`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 7-2
- “Configure Model for Debugging”

Start TLC coverage when generating code

Description

Generate the TLC execution report.

Category: Code Generation

Settings

Default: off

 On

Generates `.log` files containing the number of times each line of TLC code is executed during code generation.

 Off

Does not generate a report.

Tip

You can also generate the TLC execution report by entering the `-dg` argument into the **System target file** field.

Command-Line Information

Parameter: `TLCCoverage`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 7-2
- “Configure Model for Debugging”

Enable TLC assertion

Description

Produce the TLC stack trace

Category: Code Generation

Settings

Default: off

 On

The build process halts if a user-supplied TLC file contains an `%assert` directive that evaluates to FALSE.

 Off

The build process ignores TLC assertion code.

Command-Line Information

Parameter: `TLCAssert`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 7-2
- “Configure Model for Debugging”

Show Custom Hardware App in Simulink Toolstrip

Description

Read-only internal parameter for Simulink toolstrip

Category: Code Generation

Settings

Default: off

 On

Run on Custom Hardware app is active.

 Off

Run on Custom Hardware app is not active.

Command-Line Information

Parameter: `ShowCustomHardwareApp`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact
Efficiency	No impact
Safety precaution	No recommendation

Show Embedded Hardware App in Simulink Toolstrip

Description

Read-only internal parameter for Simulink toolstrip

Category: Code Generation

Settings

Default: off

 On

Run on Hardware Board app is active.

 Off

Run on Hardware Board app is not active.

Command-Line Information

Parameter: `ShowEmbeddedHardwareApp`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact
Efficiency	No impact
Safety precaution	No recommendation

Code Generation Parameters: Report

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” on page 8-5	Document generated code in an HTML report.
“Open report automatically” on page 8-7	Specify whether to display code generation reports automatically.
“Generate model Web view” (Embedded Coder)	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” (Embedded Coder)	Include static code metrics report in the code generation report.

These configuration parameters are under the **Advanced parameters**.

Parameter	Description
“Code-to-model” (Embedded Coder)	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” (Embedded Coder)	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” (Embedded Coder)	Open the **Model-to-code navigation** dialog box for specifying a build folder containing previously-generated model code to highlight.
“Eliminated / virtual blocks” (Embedded Coder)	Include summary of eliminated and virtual blocks in code generation report.
“Traceable Simulink blocks” (Embedded Coder)	Include summary of Simulink blocks in code generation report.
“Traceable Stateflow objects” (Embedded Coder)	Include summary of Stateflow objects in code generation report.
“Traceable MATLAB functions” (Embedded Coder)	Include summary of MATLAB functions in code generation report.
“Summarize which blocks triggered code replacements” (Embedded Coder)	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”

Code Generation: Report Tab Overview

Control the code generation report that the code generator automatically creates.

Configuration

To create a code generation report during the build process, select the **Create code generation report** parameter.

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Report” on page 8-2
- “Generate a Code Generation Report”
- “Reports for Code Generation”
- “HTML Code Generation Report Extensions” (Embedded Coder)

Create code generation report

Description

Document generated code in an HTML report.

Category: Code Generation > Report

Settings

Default: Off

 On

Generates a summary of code generation source files in an HTML report. Places the report files in an `html` subfolder within the build folder. In the report,

- The **Summary** section lists version and date information. The **Configuration Settings at the Time of Code Generation** link opens a noneditable view of the Configuration Parameters dialog that shows the Simulink model settings, including TLC options, at the time of code generation.
- The **Subsystem Report** section contains information on nonvirtual subsystems in the model.
- The **Code Interface Report** section provides information about the generated code interface, including model entry point functions and input/output data (requires an Embedded Coder license and the ERT target).
- The **Traceability Report** section allows you to account for **Eliminated / Virtual Blocks** that are untraceable, versus the listed **Traceable Simulink Blocks / Stateflow Objects / MATLAB Scripts**, providing a complete mapping between model elements and code (requires an Embedded Coder license and the ERT system target file).
- The **Static Code Metrics Report** section provides statistics of the generated code. Metrics are estimated from static analysis of the generated code.
- The **Code Replacements Report** section allows you to account for code replacement library (CRL) functions that were used during code generation, providing a mapping between each replacement instance and the Simulink block that triggered the replacement.

In the **Generated Files** section, you can click the names of source code files generated from your model to view their contents in a MATLAB Web browser window. In the displayed source code,

- Global variable instances are hyperlinked to their definitions.
- If you selected the traceability option **Code-to-model**, hyperlinks within the displayed source code let you view the blocks or subsystems from which the code was generated. Click on the hyperlinks to view the relevant blocks or subsystems in a Simulink model window (requires an Embedded Coder license and the ERT system target file).
- If you selected the traceability option **Model-to-code**, you can view the generated code for a block in the model. To highlight a block's generated code in the HTML report, right-click the block and select **C/C++ Code > Navigate to C/C++ Code** (requires an Embedded Coder license and the ERT system target file).
- If you set the **Code coverage tool** parameter on the **Code Generation > Verification** pane, you can view the code coverage data and annotations in the generated code in the HTML

Code Generation Report (requires an Embedded Coder license and the ERT system target file).

Off

Does not generate a summary of files.

Dependency

This parameter enables

- **Open report automatically** on page 8-7
- **Code-to-model** (Embedded Coder) (ERT target)
- **Model-to-code** (Embedded Coder) (ERT target)
- **Eliminated / virtual blocks** (Embedded Coder) (ERT target)
- **Traceable Simulink blocks** (Embedded Coder) (ERT target)
- **Traceable Stateflow objects** (Embedded Coder) (ERT target)
- **Traceable MATLAB functions** (Embedded Coder) (ERT target)

.

Command-Line Information

Parameter: `GenerateReport`
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” on page 8-2
- “Reports for Code Generation”
- “HTML Code Generation Report Extensions” (Embedded Coder)
- “Configure Code Coverage with Third-Party Tools” (Embedded Coder)

Open report automatically

Description

Specify whether to display code generation reports automatically.

Category: Code Generation > Report

Settings

Default: Off

 On

Displays the code generation report automatically in a new browser window.

 Off

Does not display the code generation report, but the report is still available in the `html` folder.

Dependency

This parameter is enabled by **Create code generation report**.

Command-Line Information

Parameter: `LaunchReport`
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 Report” on page 8-2
- “Reports for Code Generation”
- “HTML Code Generation Report Extensions” (Embedded Coder)

Code Generation Parameters: Comments

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” on page 9-6	Specify which comments are in generated files.
“Simulink block comments” on page 9-8	Specify whether to insert Simulink block comments.
“Trace to model using” (Embedded Coder)	Specify format of comments for Simulink blocks, Stateflow elements and MATLAB function blocks.
“Stateflow object comments” on page 9-9	Specify whether to insert Stateflow object comments.
“MATLAB source code as comments” on page 9-11	Specify whether to insert MATLAB source code as comments.
“Show eliminated blocks” on page 9-13	Specify whether to insert eliminated block's comments.
“Verbose comments for 'Model default' storage class” on page 9-14	Reduce code size or improve code traceability by controlling the generation of comments.
“Operator annotations” (Embedded Coder)	Specify whether to include operator annotations for Polyspace in the generated code as comments.

Custom comments

Parameter	Description
“Simulink block descriptions” (Embedded Coder)	Specify whether to insert descriptions of blocks into generated code as comments.

Parameter	Description
“Stateflow object descriptions” (Embedded Coder)	Specify whether to insert descriptions of Stateflow objects into generated code as comments.
“Simulink data object descriptions” (Embedded Coder)	Specify whether to insert descriptions of Simulink data objects into generated code as comments.
“Requirements in block comments” (Embedded Coder)	Specify whether to include requirement descriptions assigned to Simulink blocks in generated code as comments.
“Custom comments (MPT objects only)” (Embedded Coder)	Specify whether to include custom comments for module packaging tool (MPT) signal and parameter data objects in generated code.
“MATLAB user comments” (Embedded Coder)	Specify whether to include MATLAB user comments as comments.
“Custom comments function” (Embedded Coder)	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” (Embedded Coder)	Specify a multi-line or single-line comment style for generated C or C++ code.
“Insert Polyspace comments” (Embedded Coder)	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 `&#x30A2;`. 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” (Embedded Coder)

Code Generation: Comments Tab Overview

Control the comments that the code generator creates and inserts into the generated code.

See Also

Related Examples

- “Model Configuration Parameters: Comments” on page 9-2

Include comments

Description

Specify which comments are in generated files.

Category: Code Generation > Comments

Settings

Default: On

 On

Places comments in the generated files based on the selections in the **Auto generated comments** pane.

 Off

Omits comments from the generated files.

Note This parameter does not apply to copyright notice comments in the generated code.

Dependencies

This parameter enables:

- **Simulink block comments** on page 9-8
- **Stateflow object comments** on page 9-9
- **MATLAB source code as comments** on page 9-11
- **Show eliminated blocks** on page 9-13
- **Verbose comments for 'Model default' storage class** on page 9-14

Command-Line Information

Parameter: `GenerateComments`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Comments” on page 9-2
- “Configure Code Comments”
- “Verify Generated Code by Using Code Tracing” (Embedded Coder)

Simulink block comments

Description

Specify whether to insert Simulink block comments.

Category: Code Generation > Comments

Settings

Default: On

 On

Inserts automatically generated comments that describe a block's code. The comments precede generated code in the generated file.

 Off

Suppresses comments.

Dependency

- **Include comments** on page 9-6 enables this parameter.
- This parameter enables **Trace to model using** (Embedded Coder).

Command-Line Information

Parameter: `SimulinkBlockComments`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Comments” on page 9-2
- “Trace Simulink Model Elements in Generated Code” (Embedded Coder)

Stateflow object comments

Description

Specify whether to insert Stateflow object comments.

Category: Code Generation > Comments

Settings

Default: Off

 On

Inserts automatically generated comments that contain Stateflow object IDs or MATLAB code line locations. The comments precede the generated code in the generated file. For example,

```
/* Entry 'First': '<S2>:2' */
rtY.Out1 = 1;
```

`'<S2>:2'` is a hyperlinked traceability tag that facilitates tracing of generated code to corresponding Stateflow element.

 Off

Suppresses comments.

Dependency

- **Include comments** on page 9-6 enables this parameter.
- This parameter enables **Trace to model using** (Embedded Coder).

Command-Line Information

Parameter: `StateflowObjectComments`
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: Comments” on page 9-2
- “Trace Stateflow Elements in Generated Code” (Embedded Coder)

MATLAB source code as comments

Description

Specify whether to insert MATLAB source code as comments.

Category: Code Generation > Comments

Settings

Default: Off

☑ On

Inserts MATLAB source code as comments in the generated code. The comment appears after the traceability tag and precedes the associated generated code. For example,

```
/* '<S2>:1:22' xb1 = x-1; */
xb1 = x;
```

Selecting this parameter adds the MATLAB code `xb1 = x-1;` in the traceability comment.

Includes the function signature in the function banner.

☐ Off

Suppresses comments and does not include the function signature in the function banner.

Dependency

Include comments on page 9-6 enables this parameter.

Command-Line Information

Parameter: `MATLABSourceComments`
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: Comments” on page 9-2
- “Include MATLAB Code as Comments in the Generated Code” (Embedded Coder)

Show eliminated blocks

Description

Specify whether to insert eliminated block's comments.

Category: Code Generation > Comments

Settings

Default: On

 On

Inserts statements in the generated code from blocks eliminated as the result of optimizations (such as parameter inlining).

 Off

Suppresses statements.

Dependency

Include comments on page 9-6 enables this parameter.

Command-Line Information

Parameter: `ShowEliminatedStatement`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Comments” on page 9-2

Verbose comments for 'Model default' storage class

Description

Reduce code size or improve code traceability by controlling the generation of comments. The comments appear interleaved in the code that initializes the fields of the model parameter structure, which appears in the *model*_data.c file or the *model*.c file. Each comment indicates the name of a parameter object (Simulink.Parameter) or MATLAB variable and the blocks that use the object or variable to set parameter values.

Parameter objects and MATLAB variables appear in the model parameter structure under either of these conditions:

- You apply the storage class `Model default` to the object or variable and, in the Code Mappings editor, you set the storage class of the corresponding category of data to the default setting, `Default`.
- You apply the storage class `Auto` to the object or variable and set the model configuration parameter **Default parameter behavior** to `Tunable`. In the Code Mappings editor, you set the storage class of the corresponding category of data to the default setting, `Default`.

For more information about parameter representation in the generated code, see “How Generated Code Stores Internal Signal, State, and Parameter Data”.

Category: Code Generation > Comments

Settings

Default: On

 On

Generate comments regardless of the number of parameter values stored in the parameter structure. Use this setting to improve traceability between the generated code and the parameter objects or variables that the model uses.

 Off

Generate comments only if the parameter structure contains fewer than 1000 parameter values. An array parameter with *n* elements represents *n* values. For large models, use this setting to reduce the size of the generated file.

Dependency

Include comments on page 9-6 enables this parameter.

Command-Line Information

Parameter: `ForceParamTrailComments`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Comments” on page 9-2
- “How Generated Code Stores Internal Signal, State, and Parameter Data”

Code Generation Parameters: Identifiers

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” (Embedded Coder)	Customize generated global variable identifiers.
“Global types” (Embedded Coder)	Customize generated global type identifiers.
“Field name of global types” (Embedded Coder)	Customize generated field names of global types.
“Subsystem methods” (Embedded Coder)	Customize generated function names for reusable subsystems.
“Subsystem method arguments” (Embedded Coder)	Customize generated function argument names for reusable subsystems.
“Local temporary variables” (Embedded Coder)	Customize generated local temporary variable identifiers.
“Local block output variables” (Embedded Coder)	Customize generated local block output variable identifiers.
“Constant macros” (Embedded Coder)	Customize generated constant macro identifiers.
“Shared utilities identifier format” (Embedded Coder)	Customize shared utility identifiers.
“Minimum mangle length” (Embedded Coder)	Specify the minimum number of characters for generating name-mangling text to help avoid name collisions.
“Maximum identifier length” on page 10-5	Specify maximum number of characters in generated function, type definition, variable names.
“System-generated identifiers” (Embedded Coder)	Specify whether the code generator uses shorter, more consistent names for the $N token in system-generated identifiers.
“Generate scalar inlined parameters as” (Embedded Coder)	Control expression of scalar inlined parameter values in the generated code.
“Use the same reserved names as Simulation Target” on page 10-7	Specify whether to use the same reserved names as those specified in the **Simulation Target** pane.
“Reserved names” on page 10-8	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” (Embedded Coder)	Specify character length of `$C` token.
“EMX array utility functions identifier format” (Embedded Coder)	Customize generated identifiers for `emxArray` (embeddable `mxArray`) utility functions.
“EMX array types identifier format” (Embedded Coder)	Customize generated identifiers for `emxArray` (embeddable `mxArray`) types.
“Custom token text” (Embedded Coder)	Specify text to insert for $U token.
“Duplicate enumeration member names” on page 10-10	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” (Embedded Coder)	Specify rules for naming signals in generated code.
“M-function” (Embedded Coder)	
“Parameter naming” (Embedded Coder)	Specify rule for naming parameters in generated code.
“M-function” (Embedded Coder)	
“#define naming” (Embedded Coder)	Specify rule for naming `#define` parameters (defined with storage class `Define (Custom)`) in generated code.
“M-function” (Embedded Coder)	

See Also

More About

- “Code Appearance”
- “Model Configuration”

Code Generation: Identifiers Tab Overview

Select the automatically generated identifier naming rules.

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Identifiers” on page 10-2
- “Construction of Generated Identifiers”
- “Identifier Name Collisions and Mangling”
- “Specify Identifier Length to Avoid Naming Collisions”
- “Specify Reserved Names for Generated Identifiers”
- “Customize Generated Identifier Naming Rules” (Embedded Coder)

Maximum identifier length

Description

Specify maximum number of characters in generated function, type definition, variable names.

Category: Code Generation > Identifiers

Settings

Default: 31
Minimum: 31
Maximum: 256

You can use this parameter to limit the number of characters in function, type definition, and variable names.

Tips

- Consider increasing identifier length for models having a deep hierarchical structure.
- When generating code from a model that uses model referencing, the **Maximum identifier length** must be large enough to accommodate the root model name, and possibly, the name-mangling text. A code generation error occurs if **Maximum identifier length** is too small.
- This parameter must be the same for both top-level and referenced models.
- When a name conflict occurs between a symbol within the scope of a higher level model and a symbol within the scope of a referenced model, the symbol from the referenced model is preserved. Name mangling is performed on the symbol from the higher level model.

Command-Line Information

Parameter: `MaxIdLength`
Type: integer
Value: valid value
Default: 31

Recommended Settings

Application	Setting
Debugging	Valid value
Traceability	>30
Efficiency	No impact
Safety precaution	>30

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Identifiers” on page 10-2
- “Construction of Generated Identifiers”
- “Identifier Name Collisions and Mangling”
- “Identifier Format Control” (Embedded Coder)

Use the same reserved names as Simulation Target

Description

Specify whether to use the same reserved names as those specified in the **Simulation Target** pane.

Category: Code Generation > Identifiers

Settings

Default: Off

 On

Enables using the same reserved names as those specified in the **Simulation Target** pane.

 Off

Disables using the same reserved names as those specified in the **Simulation Target** pane.

Command-Line Information

Parameter: `UseSimReservedNames`
Type: character vector
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 Identifiers” on page 10-2

Reserved names

Description

Enter the names of variables or functions in the generated code that match the names of variables or functions specified in custom code.

Category: Code Generation > Identifiers

Settings

Default: `{}`

This action changes the names of variables or functions in the generated code to avoid name conflicts with identifiers in custom code. Reserved names must be shorter than 256 characters.

Tips

- Do not enter code generator keywords since these names cannot be changed in the generated code. For a list of keywords to avoid, see “Reserved Keywords”.
- Start each reserved name with a letter or an underscore to prevent error messages.
- Each reserved name must contain only letters, numbers, or underscores.
- Separate the reserved names using commas or spaces.
- You can also specify reserved names by using the command line:

  ```
  config_param_object.set_param('ReservedNameArray', {'abc','xyz'})
  ```

  where `config_param_object` is the object handle to the model settings in the Configuration Parameters dialog box.

Command-Line Information

Parameter: `ReservedNameArray`
Type: cell array of character vectors or string array
Value: reserved names shorter than 256 characters
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 10-2

Duplicate enumeration member names

Description

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.

Category: Code Generation > Identifiers

Settings

Default: `error`

`none`

When the code generator detects two enumeration types with the same member names, the code generation proceeds.

`warning`

When the code generator detects two enumeration types with the same member names, the software issues a warning message and the code generation proceeds.

`error`

When the code generator detects two enumeration types with the same member names, the software issues an error message and terminates the code generation.

Command-Line Information

Parameter: `EnumMemberNameClash`
Type: `character vector`
Value: `'none' | 'warning' | 'error'`
Default: `'error'`

Example

Consider these enumerations:

```
typedef int32_T enumA;
#define a       (0)
#define p       (1)

typedef int32_T enumB;
#define b       (0)
#define p       (1)
```

The enumerations have the same `int32` storage type. The enumeration member `p` with value `1` is the same for `enumA` and `enumB`.

Generate an error or warning message or allow code generation for duplicate enumeration member names by using the **Duplicate enumeration member names** configuration parameter.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Use Enumerated Data in Generated Code”
- “Use Enumerated Data in Simulink Models”
- “Model Configuration Parameters: Code Generation Identifiers” on page 10-2

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" on page 11-4	Specify whether to use the same custom code settings as those in the **Simulation Target > Custom Code** pane.
"Source file" on page 11-5	Specify custom code to include near the top of the generated model source file.
"Header file" on page 11-6	Specify custom code to include near the top of the generated model header file.
"Initialize function" on page 11-7	Specify custom code to include in the generated model initialize function.
"Terminate function" on page 11-8	Specify custom code to include in the generated model terminate function.
"Include directories" on page 11-9	Specify a list of include folders to add to the include path.
"Source files" on page 11-11	Specify a list of additional source files to compile and link with the generated code.
"Libraries" on page 11-12	Specify a list of additional libraries to link with the generated code.
"Defines" on page 11-13	Specify preprocessor macro definitions to be added to the compiler command line.

See Also

More About

- "Model Configuration"

Code Generation: Custom Code Tab Overview

Enter custom code to include in generated model files and create a list of additional folders, source files, and libraries to use when building the model.

Configuration

1. Select the type of information to include from the list on the left side of the pane.
2. Enter custom code or enter text to identify a folder, source file, or library.
3. Click **Apply**.

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Custom Code” on page 11-2
- “Integrate External Code by Using Model Configuration Parameters”

Use the same custom code settings as Simulation Target

Description

Specify whether to use the same custom code settings as those in the **Simulation Target > Custom Code** pane.

Category: Code Generation > Custom Code

Settings

Default: Off

 On

Enables using the same custom code settings as those in the **Simulation Target > Custom Code** pane.

 Off

Disables using the same custom code settings as those in the **Simulation Target > Custom Code** pane.

Command-Line Information

Parameter: `RTWUseSimCustomCode`
Type: character vector
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 Custom Code” on page 11-2
- “Integrate External Code by Using Model Configuration Parameters”

Source file

Description

Specify custom code to include near the top of the generated model source file.

Category: Code Generation > Custom Code

Settings

Default: `''`

The code generator places code near the top of the generated *`model`*`.c` or *`model`*`.cpp` file, outside of any function.

Command-Line Information

Parameter: `CustomSourceCode`
Type: character vector
Value: C code
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 Custom Code” on page 11-2
- “Integrate External Code by Using Model Configuration Parameters”

Header file

Description

Specify custom code to include near the top of the generated model header file.

Category: Code Generation > Custom Code

Settings

Default:`''`

The code generator places this code near the top of the generated *`model`*`.h` header file. If you are including a header file, in your custom header file add `#ifndef` code. This avoids multiple inclusions. For example, in `rtwtypes.h` the following #include guards are added:

```
#ifndef RTW_HEADER_rtwtypes_h_
#define RTW_HEADER_rtwtypes_h_
...
#endif /* RTW_HEADER_rtwtypes_h_ */
```

Command-Line Information

Parameter: `CustomHeaderCode`
Type: character vector
Value: C code
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 Custom Code” on page 11-2
- “Integrate External Code by Using Model Configuration Parameters”

Initialize function

Description

Specify custom code to include in the generated model initialize function.

Category: Code Generation > Custom Code

Settings

Default: `''`

The code generator places code inside the model's initialize function in the *`model`*`.c` or *`model`*`.cpp` file.

Command-Line Information

Parameter: `CustomInitializer`
Type: character vector
Value: C code
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 Custom Code” on page 11-2
- “Integrate External Code by Using Model Configuration Parameters”

Terminate function

Specify custom code to include in the generated model terminate function.

Description

Specify custom code to include in the generated model terminate function.

Category: Code Generation > Custom Code

Settings

Default: `''`

Specify code to appear in the model's generated terminate function in the *`model`*`.c` or *`model`*`.cpp` file.

Dependency

A terminate function is generated only if you select the **Terminate function required** check box.

Command-Line Information

Parameter: `CustomTerminator`
Type: character vector
Value: C code
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 Custom Code” on page 11-2
- “Integrate External Code by Using Model Configuration Parameters”

Include directories

Description

Specify a list of include folders to add to the include path.

Category: Code Generation > Custom Code

Settings

Default:`''`

Enter a space-separated list of include folders to add to the include path when compiling the generated code.

- Specify absolute or relative paths to the folders.
- Relative paths must be relative to the folder containing your model files, not relative to the build folder.
- The order in which you specify the folders is the order in which they are searched for header, source, and library files.

Note If you specify a Windows path containing one or more spaces, you must enclose the path in double quotes. For example, the second and third paths in the **Include directories** entry below must be double-quoted:

```
C:\Project "C:\Custom Files" "C:\Library Files"
```

If you set the equivalent command-line parameter `CustomInclude`, each path containing spaces must be separately double-quoted within the single-quoted third argument character vector, for example,

```
>> set_param('mymodel', 'CustomInclude', ...
             'C:\Project "C:\Custom Files" "C:\Library Files"')
```

Command-Line Information

Parameter: `CustomInclude`
Type: character vector
Value: folder path
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 Custom Code” on page 11-2
- “Integrate External Code by Using Model Configuration Parameters”

Source files

Description

Specify a list of additional source files to compile and link with the generated code.

Category: Code Generation > Custom Code

Settings

Default: `''`

Enter a space-separated list of source files to compile and link with the generated code.

Limitation

This parameter does not support Windows file names that contain embedded spaces.

Tip

You can specify just the file name if the file is in the current MATLAB folder or in one of the include folders.

Command-Line Information

Parameter: `CustomSource`
Type: character vector
Value: file name
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 Custom Code” on page 11-2
- “Integrate External Code by Using Model Configuration Parameters”

Libraries

Description

Specify a list of additional libraries to link with the generated code.

Category: Code Generation > Custom Code

Settings

Default: `''`

Enter a space-separated list of static library files to link with the generated code.

Limitation

This parameter does not support Windows file names that contain embedded spaces.

Tip

You can specify just the file name if the file is in the current MATLAB folder or in one of the include folders.

Command-Line Information

Parameter: `CustomLibrary`
Type: character vector
Value: library file name
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 Custom Code” on page 11-2
- “Integrate External Code by Using Model Configuration Parameters”

Defines

Description

Specify preprocessor macro definitions to be added to the compiler command line.

Category: Code Generation > Custom Code

Settings

Default: `''`

Enter a list of macro definitions for the compiler command line. Specify the parameters with a space-separated list of macro definitions. If a makefile is generated, these macro definitions are added to the compiler command line in the makefile. The list can include simple definitions (for example, `-DDEF1`), definitions with a value (for example, `-DDEF2=1`), and definitions with a space in the value (for example, `-DDEF3="my value"`). Definitions can omit the `-D` (for example, `-DFOO=1` and `FOO=1` are equivalent). If the toolchain uses a different flag for definitions, the code generator overrides the `-D` and uses the appropriate flag for the toolchain.

Command-Line Information

Parameter: `CustomDefine`
Type: character vector
Value: preprocessor macro definition
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 Custom Code” on page 11-2
- “Integrate External Code by Using Model Configuration Parameters”

Code Generation Parameters: Interface

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” on page 12-9	Specify a code replacement library the code generator uses when producing code for a model.
“Code replacement libraries” on page 12-11	Specify multiple code replacement libraries the code generator use when producing code for a model.
“Shared code placement” on page 12-13	Specify the location for generating utility functions, exported data type definitions, and declarations of exported data with custom storage class.
“Support: floating-point numbers” (Embedded Coder)	Specify whether to generate floating-point data and operations.
“Support: non-finite numbers” on page 12-15	Specify whether to generate non-finite data and operations on non-finite data.
“Support: complex numbers” (Embedded Coder)	Specify whether to generate complex data and operations.
“Support: absolute time” (Embedded Coder)	Specify whether to generate and maintain integer counters for absolute and elapsed time values.
“Support: continuous time” (Embedded Coder)	Specify whether to generate code for blocks that use continuous time.
“Support: variable-size signals” (Embedded Coder)	Specify whether to generate code for models that use variable-size signals.
“Code interface packaging” on page 12-17	Select the packaging for the generated C or C++ code interface.
“Multi-instance code error diagnostic” on page 12-20	Select the severity level for diagnostics displayed when a model violates requirements for generating multi-instance code.
“Pass root-level I/O as” (Embedded Coder)	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” (Embedded Coder)	Specify whether to log or monitor error status.
“Include model types in model class” (Embedded Coder)	Specify whether to generate model type definitions in a model class.
“Array layout” on page 12-44	Specify layout of array data for code generation as column-major or row-major

Parameter	Description
“External functions compatibility for row-major code generation” on page 12-46	Select diagnostic action if Simulink encounters a function that has no specified array layout
“Generate C API for: signals” on page 12-22	Generate C API data interface code with a signals structure.
“Generate C API for: parameters” on page 12-23	Generate C API data interface code with parameter tuning structures.
“Generate C API for: states” on page 12-24	Generate C API data interface code with a states structure.
“Generate C API for: root-level I/O” on page 12-25	Generate C API data interface code with a root-level I/O structure.
“ASAP2 interface” on page 12-26	Generate code for the ASAP2 data interface.
“External mode” on page 12-27	Generate code for the external mode data interface.
“Transport layer” on page 12-28	Specify the transport protocol for communications.
“MEX-file arguments” on page 12-30	Specify arguments to pass to an external mode interface MEX-file for communicating with executing targets.
“Static memory allocation” on page 12-32	Control memory buffer for external mode communication.
“Static memory buffer size” on page 12-39	Specify the memory buffer size for external mode communication.
“Target library” on page 12-34	Specify the target deep learning library used during code generation. `cuDNN` or `TensorRT` requires a GPU Coder license.
“ARM Compute Library version” on page 12-36	Specify the version of ARM® Compute Library.
“ARM Compute Library architecture” on page 12-37	Specify the ARM architecture supported in the target hardware.
“Auto tuning” on page 12-38	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” on page 12-48	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” (Embedded Coder)	Specify whether to generate code for non-inlined S-functions.
“Maximum word length” on page 12-50	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)” on page 12-43	Number of bytes of the character buffer generated for dynamic string signals without maximum length.
“Multiword type definitions” (Embedded Coder)	Specify whether to use system-defined or user-defined type definitions for multiword data types in generated code.
“Classic call interface” on page 12-52	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” (Embedded Coder)	Control how the generated code allocates memory for model data.
“Single output/update function” on page 12-54	Specify whether to generate the *`model`*`_step` function.
“Terminate function required” (Embedded Coder)	Specify whether to generate the *`model`*`_terminate` function.
“Combine signal/state structures” (Embedded Coder)	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” (Embedded Coder)	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” on page 12-56	Specify MAT-file logging.
“MAT-file variable name modifier” (Embedded Coder)	Select the text to add to MAT-file variable names.
“Existing shared code” (Embedded Coder)	Specify folder that contains existing shared code
“Remove disable function” (Embedded Coder)	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” (Embedded Coder)	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” on page 12-41	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” on page 12-42	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” (Embedded Coder)	Specify whether to generate a destructor for the C++ model class.
“Use dynamic memory allocation for model block instantiation” (Embedded Coder)	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” on page 12-9	Create custom Code Replacement libraries using code replacement tool.
“Ignore custom storage classes” (Embedded Coder)	Specify whether to apply or ignore custom storage classes.
“Ignore test point signals” (Embedded Coder)	Specify allocation of memory buffers for test points.
“Implement each data store block as a unique access point” (Embedded Coder)	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”

Code Generation: Interface Tab Overview

Select the target software environment, output variable name modifier, and data exchange interface.

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 12-2
- “Run-Time Environment Configuration”

Code replacement library

Description

Specify a code replacement library that the code generator uses when producing code for a model.

Category: Code Generation > Interface

Settings

Default: `None`

`None`

Does not use a code replacement library.

`Named code replacement libraries`

Generates calls to a specific platform, compiler, or standards code replacement library. The list of named libraries depends on system target file.

Tips

- Before setting this parameter, verify that your compiler supports the library that you want to use. If you select a parameter value that your compiler does not support, compiler errors can occur.
- If you set parameter **Shared code placement** to `Shared location` or you generate code for models in a model reference hierarchy,
  - Models that are sharing the location or are in the model hierarchy must specify the same code replacement library (same name, tables, and table entries).
  - If you change the name or contents of the code replacement library and rebuild the model from the same folder as the previous build, the code generator reports a checksum warning (see "Manage the Shared Utility Code Checksum"). The warning prompts you to remove the existing folder and stop or stop code generation.
- If both of the following conditions exist for a model that contains Stateflow charts, the Simulink software regenerates code for the charts and recompiles the generated code.
  - You *do not* set parameter **Shared code placement** to `Shared location`.
  - You change the code replacement library name or contents before regenerating code.

Dependencies

- This parameter appears only for GRT-based targets. For ERT-based targets, use the parameter "Code replacement libraries" on page 12-11.

Command-Line Information

Parameter: `CodeReplacementLibrary`
Type: character vector
Value: `'None'` | `'GNU C99 extensions'`
Default: `'None'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Valid library
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 12-2
- “Run-Time Environment Configuration”
- “What Is Code Replacement Customization?” (Embedded Coder)
- “Develop a Code Replacement Library” (Embedded Coder)

Code replacement libraries

Description

In Embedded Coder, specify a list of code replacement libraries that the code generator uses when producing code for a model. To select the code replacement library, click the **Select** button and use the **Select code replacement libraries-prioritized** dialog box.

Category: Code Generation > Interface

Settings

Default: `None`

`None`

Does not use a code replacement library.

`Named code replacement libraries`

Generates calls to multiple platform-specific, compiler, custom, or standard code replacement libraries. The list of named libraries depends on:

- Installed support packages.
- System target file, language, standard math library, and device vendor configuration.
- Whether you created and registered code replacement libraries, by using Embedded Coder.

Tips

- Order the code replacement libraries based on the priority of code replacement. Libraries at the top of the list have a higher priority of replacement.
- Specify multiple code replacement libraries enclosed by single quotes and separated by commas.
- Before setting this parameter, verify that your compiler supports the library that you want to use. If you select a parameter value that your compiler does not support, compiler errors can occur.
- If you set parameter **Shared code placement** to `Shared location` or you generate code for models in a model reference hierarchy:
  - For models that are sharing the location or are in the model hierarchy, you must specify the same code replacement library (same name, tables, and table entries).
  - If you change the name or contents of the code replacement library and rebuild the model from the same folder as the previous build, the code generator reports a checksum warning (see “Manage the Shared Utility Code Checksum”). The warning prompts you to remove the existing folder and stop or stop code generation.
- If both of the following conditions exist for a model that contains Stateflow charts, the Simulink software regenerates code for the charts and recompiles the generated code.
  - You *do not* set parameter **Shared code placement** to `Shared location`.
  - Before regenerating code, you change the code replacement library name or contents.

Dependencies

- This parameter appears only for ERT-based targets. For GRT-based targets, use the parameter “Code replacement library” on page 12-9.
- When you generate code, this parameter requires Embedded Coder.

Command-Line Information

Parameter: `CodeReplacementLibrary`
Type: character vector
Value: `'None'` | `'GNU C99 extensions'` | `'Intel SSE (Windows)'` | `'Intel SSE (Linux)'` | `'Intel AVX (Windows)'` | `'Intel AVX (Linux)'` | `'Intel AVX-512 (Windows)'` | `'Intel AVX-512 (Linux)'` | `'AUTOSAR 4.0'` | `'GCC ARM Cortex-A'` | `'myCustomLibrary'`
Default: `'None'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Valid libraries
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 12-2
- “What Is Code Replacement Customization?” (Embedded Coder)
- “Develop a Code Replacement Library” (Embedded Coder)
- “Optimize Generated Code by Using Multiple Code Replacement Libraries” (Embedded Coder)

Shared code placement

Description

Specify the location for generating utility functions, exported data type definitions, and declarations of exported data with custom storage class.

Category: Code Generation > Interface

Settings

Default: `Auto`

`Auto`

The code generator places utility code within the *`codeGenFolder`*`/slprj/`*`target`*`/_sharedutils` (or *`codeGenFolder`*`/`*`targetSpecific`*`/_shared`) folder for a model that contains Existing Shared Code (Embedded Coder) or at least one of the following blocks:

- Model blocks
- Simulink Function blocks
- Function Caller blocks
- Calls to Simulink Functions from Stateflow or MATLAB Function blocks
- Stateflow graphical functions when you select the **Export Chart Level Functions** parameter

If a model does not contain one of the above blocks or Existing Shared Code (Embedded Coder), the code generator places utility code in the build folder (generally, the folder that contains *`model`*`.c` or *`model`*`.cpp`).

`Shared location`

Directs code for utilities to be placed within the *`codeGenFolder`*`/slprj/`*`target`*`/_sharedutils` (or *`codeGenFolder`*`/`*`targetSpecific`*`/_shared`) folder.

Command-Line Information

Parameter: `UtilityFuncGeneration`
Type: character vector
Value: `'Auto'` | `'Shared location'`
Default: `'Auto'`

Recommended Settings

Application	Setting
Debugging	`Shared location` (GRT) No impact (ERT)
Traceability	`Shared location` (GRT) No impact (ERT)
Efficiency	No impact (execution, RAM) `Shared location` (ROM)

Application	Setting
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 12-2
- “Run-Time Environment Configuration”
- “Manage Build Process Folders”
- “Sharing Utility Code”

Support: non-finite numbers

Description

Specify whether to generate non-finite data and operations on non-finite data.

Category: Code Generation > Interface

Settings

Default: on

 On

Generates non-finite data (for example, `NaN` and `Inf`) and related operations.

 Off

Does not generate non-finite data and operations. If you clear this option, an error occurs if the code generator encounters non-finite data or expressions. The error message reports offending blocks and parameters.

Note Code generation is optimized with the assumption that non-finite data are absent. However, if your application produces non-finite numbers through signal data or MATLAB code, the behavior of the generated code might be inconsistent with simulation results when processing non-finite data.

Dependencies

- For ERT-based targets, parameter **Support: floating-point numbers** enables **Support: non-finite numbers**.
- If `off` for top model, must be `off` for referenced models.
- When you select parameter **MAT-File Logging**, you must also select **Support: non-finite numbers** and, if you use an ERT-based system target file, **Support: floating-point numbers**.

Command-Line Information

Parameter: `SupportNonFinite`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Off (execution, ROM), No impact (RAM)

Application	Setting
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 12-2

Code interface packaging

Description

Select the packaging for the generated C or C++ code interface.

Category: Code Generation > Interface

Settings

Default: `Nonreusable function` if parameter **Language** is set to `C`; `C++ class` if **Language** is set to `C++`

`C++ class`

Generate a C++ class interface to model code. The generated interface encapsulates required model data into C++ class attributes and model entry point functions into C++ class methods.

`Nonreusable function`

Generate nonreusable code. Model data structures are statically allocated and accessed by model entry point functions directly in the model code.

`Reusable function`

Generate reusable, multi-instance code that is reentrant, as follows:

- For a GRT-based model, the generated *`model`*`.c` source file contains an allocation function that dynamically allocates model data for each instance of the model. For an ERT-based model, you can use parameter **Use dynamic memory allocation for model initialization** to control whether an allocation function is generated.
- The generated code passes the real-time model data structure in, by reference, as an argument to *`model`*`_step` and the other model entry point functions.
- The real-time model data structure is exported with the *`model`*`.h` header file.

For an ERT-based model, you can use parameter **Pass root-level I/O as** to control how root-level input and output arguments are passed to the reusable model entry-point functions. They can be included in the real-time model data structure that is passed to the functions, passed as individual arguments, or passed as references to an input structure and an output structure.

Tips

- Entry points are exported with *`model`*`.h`. To call the entry-point functions from handwritten code, add an `#include model.h` directive to the code.
- When you select `Reusable function`, the code generator generates a pointer to the real-time model object (*`model`*`_M`).
- When you select `Reusable function`, the code generator can generate code that compiles but is not reentrant. For example, if a signal, DWork structure, or parameter data has a storage class other than `Auto`, global data structures are generated.

Dependencies

- The value `C++ class` is available only if parameter **Language** is set to `C++`.
- Selecting `Reusable function` or `C++ class` enables parameter **Multi-instance code error diagnostic**.
- For an ERT-based system target file, selecting `Reusable function` enables parameters **Pass root-level I/O as** and **Use dynamic memory allocation for model initialization**.
- To enable parameter **Classic call interface**, select `Nonreusable function`.
- For an ERT-based system target file, selecting `C++ class` enables the following model configuration controls for customizing the model class interface:
  - **Data Member Visibility/Access Control** subpane
  - Parameters **Generate destructor** and **Use dynamic memory allocation for model block instantiation**
- For an ERT-based system target file, you can select `Reusable function` with the static `ert_main.c` module, if you do the following:
  - Set parameter **Pass root-level I/O as** to `Part of model data structure`.
  - Select parameter **Use dynamic memory allocation for model initialization**.
- For an ERT-based system target file, you cannot select `Reusable function` if you are:
  - Customizing the `model_step` function prototype
  - Selecting subsystem block parameter **Function with separate data**
  - Using a subsystem that
    - Has multiple ports that share source
    - Has a port that is used by multiple instances of the subsystem and has different sample times, data types, complexity, frame status, or dimensions across the instances
    - Has output marked as a global signal
    - For each instance contains identical blocks with different names or parameter settings
- Selecting `Reusable function` does not change the code generated for function-call subsystems.

Command-Line Information

Parameter: `CodeInterfacePackaging`
Type: character vector
Value: `'C++ class'` | `'Nonreusable function'` | `'Reusable function'`
Default: `'Nonreusable function'` if `TargetLang` is set to `'C'`; `'C++ class'` if `TargetLang` is set to `'C++'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	`Reusable function` or `C++ class`

Application	Setting
Safety precaution	No impact

See Also

`model_step`

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 12-2
- “Configure C Code Generation for Model Entry-Point Functions”
- “Generate Reentrant Code from Top Models”
- “Combine Code Generated for Multiple Models or Multiple Instances of a Model”
- “Generate Reentrant Code from Top Models”
- “Generate C++ Class Interface to Model or Subsystem Code”
- “Interactively Configure C++ Interface” (Embedded Coder)
- “Generate Reentrant Code from Subsystems”
- “S-Functions for Code Reuse”
- “Static Main Program Module” (Embedded Coder)
- “Configure C Code Generation for Model Entry-Point Functions” (Embedded Coder)
- “Generate Modular Function Code for Nonvirtual Subsystems” (Embedded Coder)
- “Generate Component Source Code for Export to External Code Base” (Embedded Coder)

Multi-instance code error diagnostic

Description

Select the severity level for diagnostics displayed when a model violates requirements for generating multi-instance code.

Category: Code Generation > Interface

Settings

Default: `Error`

`None`

Proceed with build without displaying a diagnostic message.

`Warning`

Proceed with build after displaying a warning message.

`Error`

Abort build after displaying an error message.

Under certain conditions, the software can:

- Generate code that compiles but is not reentrant. For example, if a signal or DWork structure has a storage class other than `Auto`, global data structures are generated.
- Be unable to generate valid and compilable code. For example, if the model contains an S-function that is not code-reuse compliant or a subsystem triggered by a wide function-call trigger, the code generator produces invalid code, displays an error message, and terminates the build.

Dependencies

This parameter is enabled by setting parameter **Code interface packaging** to `Reusable function` or `C++ class`.

Command-Line Information

Parameter: `MultiInstanceErrorCode`
Type: character vector
Value: `'None'` | `'Warning'` | `'Error'`
Default: `'Error'`

Recommended Settings

Application	Setting
Debugging	`Warning` or `Error`
Traceability	No impact
Efficiency	`None`
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 12-2
- “Configure C Code Generation for Model Entry-Point Functions”
- “Generate Reentrant Code from Top Models”
- “Generate C++ Class Interface to Model or Subsystem Code”
- “Control Generation of Functions for Subsystems”
- “Generate Reentrant Code from Subsystems”
- “Generate Reentrant Code from Subsystems”
- “Generate Modular Function Code for Nonvirtual Subsystems” (Embedded Coder)

Generate C API for: signals

Description

Generate C API data interface code with a signals structure.

Category: Code Generation > Interface

Settings

Default: off

 On

Generates C API interface to global block outputs.

 Off

Does not generate C API signals.

Command-Line Information

Parameter: `RTWCAPISignals`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact during development Off for production code generation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 12-2
- “Exchange Data Between Generated and External Code Using C API”

Generate C API for: parameters

Description

Generate C API data interface code with parameter tuning structures.

Category: Code Generation > Interface

Settings

Default: off

 On

Generates C API interface to global block parameters.

 Off

Does not generate C API parameters.

Command-Line Information

Parameter: `RTWCAPIParams`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact during development Off for production code generation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 12-2
- “Exchange Data Between Generated and External Code Using C API”

Generate C API for: states

Description

Generate C API data interface code with a states structure.

Category: Code Generation > Interface

Settings

Default: off

 On

Generates C API interface to discrete and continuous states.

 Off

Does not generate C API states.

Command-Line Information

Parameter: `RTWCAPIStates`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact during development Off for production code generation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 12-2
- “Exchange Data Between Generated and External Code Using C API”

Generate C API for: root-level I/O

Description

Generate C API data interface code with a root-level I/O structure.

Category: Code Generation > Interface

Settings

Default: off

 On

Generates a C API interface to root-level inputs and outputs.

 Off

Does not generate a C API interface to root-level inputs and outputs.

Command-Line Information

Parameter: `RTWCAPIRootIO`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact during development Off for production code generation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 12-2
- “Exchange Data Between Generated and External Code Using C API”

ASAP2 interface

Description

Generate code for the ASAP2 data interface.

Category: Code Generation > Interface

Settings

Default: off

 On

Generates code for the ASAP2 data interface.

 Off

Does not generate code for the ASAP2 data interface.

Command-Line Information

Parameter: `GenerateASAP2`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact during development Off for production code generation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 12-2
- “Export ASAP2 File for Data Measurement and Calibration”

External mode

Description

Generate code for the external mode data interface.

Category: Code Generation > Interface

Settings

Default: off

 On

Generates code for the external mode data interface.

 Off

Does not generate code for the external mode data interface.

Command-Line Information

Parameter: `ExtMode`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact during development Off for production code generation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 12-2
- “External Mode Simulations for Parameter Tuning and Signal Monitoring”
- “External Mode Simulation by Using XCP Communication”
- “External Mode Simulation with TCP/IP or Serial Communication”

Transport layer

Description

Specify the transport protocol for communications.

Category: Code Generation > Interface

Settings

Default: `tcpip`

`tcpip`

Use a TCP/IP transport mechanism. Selecting this parameter sets parameter **MEX-file name** to `ext_comm`.

`serial`

Use a serial transport mechanism. Selecting this parameter sets parameter **MEX-file name** to `ext_serial_win32_comm`.

`XCP on TCP/IP`

Use XCP protocol with TCP/IP transport layer. Selecting this parameter sets parameter **MEX-file name** to `ext_xcp`.

`XCP on Serial`

Use XCP protocol with serial transport layer. Selecting this parameter sets parameter **MEX-file name** to `ext_xcp`.

`customTransportLayer`

Use custom transport layer.

Tips

The Configuration Parameters dialog box displays parameter **MEX-file name** next to **Transport layer**. You cannot edit the value for **MEX-file name**. The value is specified either in *`matlabroot`*`/toolbox/simulink/simulink/extmode_transports.m` for targets provided by MathWorks, or in an `sl_customization.m` file for custom targets and transport mechanisms.

The command-line parameter is an index. To get the transport layer index, use these commands:

```
cs = getActiveConfigSet(modelName);
index = Simulink.ExtMode.Transports.getExtModeTransportIndex(cs, transportLayer);
```

`transportLayer` is one of these values:

- `'tcpip'`
- `'serial'`
- `'XCP on TCP/IP'`
- `'XCP on Serial'`
- *`customTransportLayer`*

To select the transport layer:

```
set_param(cs, 'ExtModeTransport', index)
```

To determine the transport layer:

```
transportLayerName = Simulink.ExtMode.Transports.getExtModeTransport(cs, index)
```

Dependency

Selecting parameter **External mode** enables this parameter.

Command-Line Information

Parameter: `ExtModeTransport`
Type: integer
Value: See “Tips” on page 12-28
Default: `0`

Recommended Settings

Application	No impact
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 12-2
- “External Mode Simulation by Using XCP Communication”
- “External Mode Simulation with TCP/IP or Serial Communication”
- “Customize XCP Slave Software”
- “Create a Transport Layer for TCP/IP or Serial External Mode Communication”

MEX-file arguments

Description

Specify arguments to pass to an external mode interface MEX-file for communicating with executing targets.

Category: Code Generation > Interface

Settings

Default: `''`

For XCP communication with a TCP/IP transport layer, there are four optional arguments:

- Network name of your target processor –– For example, `'localhost'` if the target process is your development computer or the IP address `'148.27.151.12'`.
- Verbosity level –– `0` for no information or `1` to display detailed information during data transfer.
- Port number of TCP/IP server –– An integer value between `256` and `65535`, with a default of `17725`.
- Symbols file name –– File format is PDB for Windows or ELF for Linux®.

For XCP communication with a serial transport layer, there are four optional arguments:

- Verbosity level –– `0` for no information or `1` for detailed information.
- Serial port ID –– On Windows, `'COM1'` or 1 for `COM1`, `'COM2'` or 2 for `COM2`, etc. On Linux, `'/dev/ttyS0'`, etc.
- Baud –– `1200`, `2400`, `4800`, `9600`, `14400`, `19200`, `38400`, `57600` (default), or `115200`.
- Symbols file name –– File format is PDB for Windows or ELF for Linux.

For TCP/IP interfaces, `ext_comm` allows three optional arguments:

- Network name of your target processor –– For example, `'myComputer'` or `'148.27.151.12'`.
- Verbosity level –– `0` for no information or `1` to display detailed information during data transfer.
- Port number of TCP/IP server –– An integer value between `256` and `65535`, with a default of `17725`.

For a serial transport, `ext_serial_win32_comm` allows three optional arguments:

- Verbosity level –– `0` for no information or `1` for detailed information.
- Serial port ID –– `1` for `COM1`, etc.
- Baud –– `1200`, `2400`, `4800`, `9600`, `14400`, `19200`, `38400`, `57600` (default), or `115200`.

Specify the arguments in the list order. For example, if you want to specify the verbosity level (the second argument), then you must also specify the network name of the target processor (the first argument). You can use white space or commas as argument delimiters.

```
'148.27.151.12' 1 30000
```

Dependency

Selecting parameter **External mode** enables this parameter.

Command-Line Information

Parameter: `ExtModeMexArgs`
Type: character vector
Value: valid arguments
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 Interface” on page 12-2
- “External Mode Simulation by Using XCP Communication”
- “External Mode Simulation with TCP/IP or Serial Communication”

Static memory allocation

Description

Control memory buffer for external mode communication.

Category: Code Generation > Interface

Settings

Default: off

 On

Enables parameter **Static memory buffer size** for allocating dynamic memory.

 Off

Uses a static memory buffer for external mode simulation instead of allocating dynamic memory (calls to `malloc`).

Tips

To determine how much memory to allocate, select verbose mode on the target. That selection displays the amount of memory the target tries to allocate and the amount of memory available.

If you set parameter **Transport layer** to `XCP on TCP/IP` or `XCP on Serial`, you can use parameter **Static memory buffer size** to specify the size of XCP slave memory that is allocated for signal logging.

Dependencies

- Selecting parameter **External mode** enables this parameter.
- This parameter enables parameter **Static memory buffer size**.
- If parameter **Transport layer** is set to `XCP on TCP/IP` or `XCP on Serial`, you cannot disable this parameter.

Command-Line Information

Parameter: `ExtModeStaticAlloc`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 12-2
- “External Mode Simulation with TCP/IP or Serial Communication”
- “Control Memory Allocation for Communication Buffers in Target”
- “XCP Platform Abstraction Layer”

Target library

Description

Specify the target deep learning library used during code generation.

Category: Code Generation > Interface

Settings

Default: `None`

`MKL-DNN`

Use this option for generating code that uses the Intel Math Kernel Library for Deep Neural Networks (Intel MKL-DNN).

`ARM Compute`

Use this option for generating code that uses the ARM Compute Library.

`cuDNN`

Use this option for generating code that uses the CUDA Deep Neural Network library (cuDNN).

`TensorRT`

Use this option for generating code that takes advantage of the NVIDIA TensorRT – high performance deep learning inference optimizer and run-time library.

Dependencies

- To enable this parameter, select `C++` for the **Language** and `grt.tlc` or `ert.tlc` for **System target file**.
- `MKL-DNN` or `ARM Compute` is available when **GPU acceleration** on the **Code Generation** pane is disabled.
- `cuDNN` or `TensorRT` requires a GPU Coder license.
- `cuDNN` or `TensorRT` is available when **GPU acceleration** on the **Code Generation** pane is enabled.

Command-Line Information

Parameter: `DLTargetLib`
Type: character vector
Value: `'None'` | `'MKL-DNN'` | `'ARM Compute'` | `'cuDNN'` | `'TensorRT'`
Default: `'None'`

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 12-2

- “Deep Learning in Simulink by Using MATLAB Function Block” (GPU Coder)

ARM Compute Library version

Description

Specify the version of ARM Compute Library.

Category: Code Generation > Interface

Settings

Default: `19.05`

Version of ARM Compute Library used on the target hardware, specified as a character vector. If you set this parameter to a version later than `'19.05'`, the software is set to the default value of `'19.05'`.

Dependencies

To enable this parameter, select `ARM Compute` for **Target Library**.

Command-Line Information

Parameter: `DLArmComputeVersion`
Type: character vector
Value: `'19.05'` | `'19.02'` | `'18.11'` | `'18.08'` | `'18.05'`
Default: `'19.05'`

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 12-2
- “Deep Learning in Simulink by Using MATLAB Function Block” (GPU Coder)

ARM Compute Library architecture

Description

Specify the ARM architecture supported in the target hardware.

Category: Code Generation > Interface

Settings

Default: `Unspecified`

ARM architecture supported in the target hardware, specified as a character vector. The specified architecture must be the same as the architecture for the ARM Compute Library on the target hardware.

Dependencies

To enable this parameter, select `ARM Compute` for **Target Library**.

Command-Line Information

Parameter: `DLArmComputeArch`
Type: character vector
Value: `'Unspecified'` | `'armv8'` | `'armv7'`
Default: `'Unspecified'`

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 12-2
- “Deep Learning in Simulink by Using MATLAB Function Block” (GPU Coder)

Auto tuning

Description

Use auto tuning for cuDNN library.

Category: Code Generation > Interface

Settings

Default: On

On

Use this option to enable auto tuning feature. Enabling auto tuning allows the cuDNN library to find the fastest convolution algorithms. This increases performance for larger networks such as `SegNet` and `ResNet`.

Off

Disable auto tuning for the cuDNN library.

Dependencies

- This parameter requires a GPU Coder license.
- To enable this parameter, select `C++` for the **Language** and `cuDNN` for **Target library**.

Command-Line Information

Parameter: `DLAutoTuning`
Value: `'on' | 'off'`
Default: `'on'`

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 12-2
- “Deep Learning in Simulink by Using MATLAB Function Block” (GPU Coder)

Static memory buffer size

Description

Specify the memory buffer size for external mode communication.

Category: Code Generation > Interface

Settings

Default: `1000000`

Enter the number of bytes to preallocate for external mode communications buffers in the target.

Tips

- If you enter too small a value for your application, external mode issues an out-of-memory error.
- To determine how much memory to allocate, select verbose mode on the target. That selection displays the amount of memory the target tries to allocate and the amount of memory available.
- If parameter **Transport layer** is set to `XCP on TCP/IP` or `XCP on Serial`, you can use this parameter to specify the size of XCP slave memory allocated for signal logging.

Dependency

Selecting parameter **Static memory allocation** enables this parameter.

Command-Line Information

Parameter: `ExtModeStaticAllocSize`
Type: integer
Value: valid value
Default: `1000000`

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” on page 12-2
- “External Mode Simulation with TCP/IP or Serial Communication”

- “Control Memory Allocation for Communication Buffers in Target”

LUT object struct order for even spacing specification

Description

Change the order of the fields in the generated structure for a lookup table object whose specification parameter is set to even spacing.

Category: Code Generation > Interface > Advanced Parameters

Settings

Default: `Size,Breakpoints,Table`

`Size,Breakpoints,Table`

Display structure in the order size, breakpoints, table.

`Size,Table,Breakpoints`

Display structure in the order size, table, breakpoints.

Command-Line Information

Parameter: `LUTObjectStructOrderEvenSpacing`
Type: character vector
Value: `'Size,Breakpoints,Table'` | `'Size,Table,Breakpoints'`
Default: `'Size,Breakpoints,Table'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

LUT object struct order for explicit value specification

Description

Change the order of the fields in the generated structure for a lookup table object whose specification parameter is set to explicit value.

Category: Code Generation > Interface> Advanced Parameters

Settings

Default: `Size,Breakpoints,Table`

`Size,Breakpoints,Table`

Display structure in the order size, breakpoints, table.

`Size,Table,Breakpoints`

Display structure in the order size, table, breakpoints.

Command-Line Information

Parameter: `LUTObjectStructOrderExplicitValues`
Type: character vector
Value: `'Size,Breakpoints,Table'` | `'Size,Table,Breakpoints'`
Default: `'Size,Breakpoints,Table'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Buffer size of dynamically-sized string (bytes)

Description

Number of bytes of the character buffer generated for dynamic string signals without maximum length.

Category: Code Generation > Interface

Settings

Default: 256

Command-Line Information

Parameter: `DynamicStringBufferSize`
Type: character vector
Value: scalar
Default: 256

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Array layout

Description

Specify the layout of array data for code generation as column-major or row-major.

Category: Code Generation > Interface

Settings

Default: `Column-major`

`Column-major`

Generate code in column-major array layout. For example, consider matrix `A`, which is a 4x3 matrix:

```
A =
    1    2    3
    4    5    6
    7    8    9
    10   11   12
```

In column-major array layout, the elements of the columns are contiguous in memory. `A` is represented in the generated code as:

```
1    4    7    10    2    5    8    11    3    6    9    12
```

`Row-major`

Generate code in row-major array layout. For example, for matrix `A`, in row-major array layout, the elements of the rows are contiguous. `A` is represented in the generated code as:

```
1    2    3    4    5    6    7    8    9    10    11    12
```

Select parameter **Use algorithms optimized for row-major array layout** to enable efficient row-major algorithms.

Command-Line Information

Parameter: `ArrayLayout`
Type: character vector
Value: `'Column-major'` | `'Row-major'`
Default: `'Column-major'`

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” on page 12-2
- “Code Generation of Matrices and Arrays”

External functions compatibility for row-major code generation

Description

Select the diagnostic action if Simulink encounters a function that has no specified array layout.

Category: Code Generation > Interface

Settings

Default: `error`

`none`
: The code generator produces code without generating an error or warning.

`warning`
: The code generator builds the model and displays a warning message.

`error`
: The code generator terminates the build and displays an error message.

Dependencies

To enable this parameter, set parameter **Array layout** on page 12-44 to `Row-major`.

This parameter works only if your S-function or custom C function has a multidimensional array.

Command-Line Information

Parameter: `UnsupportedSFcnMsg`
Type: character vector
Value: `'none'` | `'warning'` | `'error'`
Default: `'error'`

Recommended Settings

Application	Setting
Debugging	`warning`
Traceability	No impact
Efficiency	No impact
Safety precaution	`error`

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 12-2
- “Generate Row-Major Code for S-Functions”

- “Code Generation of Matrices and Arrays”

Standard math library

Description

Specify standard math library for model.

Category: Code Generation > Interface

Settings

Default: `C99 (ISO)` or, if **Language** is set to `C++`, `C++03 (ISO)`

`C89/C90 (ANSI)`

Generates calls to the ISO/IEC 9899:1990 C standard math library.

`C99 (ISO)`

Generates calls to the ISO/IEC 9899:1999 C standard math library.

`C++03 (ISO)`

Generates calls to the ISO/IEC 14882:2003 C++ standard math library.

`C++11 (ISO)`

Generates calls to the ISO/IEC 14882:2011 C++ standard math library.

Tips

- Before setting this parameter, verify that your compiler supports the library you want to use. If you select a parameter value that your compiler does not support, compiler errors can occur.
- If you are using a compiler that does not support ISO/IEC 9899:1999 C, set this parameter to `C89/C90 (ANSI)`.
- The build process checks whether the specified standard math library and toolchain are compatible. If they are not compatible, a warning occurs during code generation and the build process continues.
- If you use a Linux GCC compiler for the build process, the software uses the compiler default language standard to compile the generated code. Depending on the compiler version, the default language standard varies. For more information, see https://gcc.gnu.org/projects/cxx-status.html.

  For example, if you use the GCC compiler version 8.x and set the parameter **Standard math library** to `C++03`, the software uses the `C++14` language standard during compilation. The usage of `C++14` enables building the generated code that integrates custom code that uses `C++11` or `C++14` features.

  If you want to enforce the language standard specified in the parameter on the compilation process, then you must add language standard specific compiler flags manually. If your model is configured to use a toolchain for building code, use this procedure:

  1. In the Configuration Parameters dialog box, set **Build configuration** to `Specify`.
  2. Under **Toolchain details**, at the end of the `C Compiler` and `C++ Compiler` **Options** field entries, manually add these compiler flags:

- When the **Standard math library** is set to `C99(ISO)`, add the flag `-std=c99 -pedantic`.
- When the **Standard math library** is set to `C89/90(ANSI)`, add the flag `-ansi -pedantic`.
- When the **Standard math library** is set to `C++03`, add the flag `-std=c++03 -pedantic`.
- When the **Standard math library** is set to `C++11`, add the flag `-std=c++11 -pedantic`.

Dependencies

- `C++03` is available for use only if you select `C++` for the **Language** parameter.
- When you change the value of the **Language** parameter, the standard math library is updated to `C99 (ISO)` for C and `C++03 (ISO)` or `C++11 (ISO)` for C++.

Command-Line Information

Parameter: `TargetLangStandard`
Type: character vector
Value: `'C89/C90 (ANSI)'` | `'C99 (ISO)'` | `'C++03 (ISO)'` | `'C++11 (ISO)'`
Default: For C, `'C99 (ISO)'`; for C++ `'C++03 (ISO)'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Valid library
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 12-2
- “Specify Single-Precision Data Type for Embedded Application”
- “Run-Time Environment Configuration”

Maximum word length

Description

Specify a maximum word length, in bits, for which the code generation process generates system-defined multiword type definitions.

Category: Code Generation > Interface

Settings

Default: `256` for ERT targets, `2048` for GRT targets

Specify a maximum word length, in bits, for which the code generation process generates multiword type definitions into the file `multiword_types.h`. All multiword type definitions up to and including this number of bits are generated. 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 requires an Embedded Coder license when generating code.
- This parameter is enabled by selecting the value `User defined` for the parameter **Multiword type definitions**.

Command-Line Information

Parameter: MultiwordLength
Type: integer

Value: valid quantity of bits representing a word size
Default: `256` for ERT targets, `2048` for GRT targets

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 12-2

Classic call interface

Description

Specify whether to generate model function calls compatible with the main program module of the GRT target in models created before R2012a.

Category: Code Generation > Interface

Settings

Default: off (except on for GRT models created before R2012a)

 On

Generates model function calls that are compatible with the main program module of the GRT system target file (`grt_main.c` or `grt_main.cpp`) in models created before R2012a.

This option provides a quick way to use code generated in the current release with a GRT-based custom target that has a main program module based on pre-R2012a `grt_main.c` or `grt_main.cpp`.

 Off

Disables the classic call interface.

Tips

The following are unsupported:

- Data type replacement
- **Code interface packaging** set to `Reusable function` or `C++ class`.
- Nonvirtual subsystem option **Function with separate data**

Dependencies

- Setting **Code interface packaging** to `C++ class` disables this option.
- Selecting this option disables the incompatible option **Single output/update function**. Clearing this option enables (but does not select) **Single output/update function**.

Command-Line Information

Parameter: `GRTInterface`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'` (except `'on'` for GRT models created before R2012a)

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Off
Efficiency	Off (execution, ROM), No impact (RAM)
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 12-2
- “Use Discrete and Continuous Time” (Embedded Coder)

Single output/update function

Description

Specify whether to generate the *model*_step function.

Category: Code Generation > Interface

Settings

Default: on

 On

Generates the *model*_step function for a model. This function contains the output and update function code for the blocks in the model and is called by rt_OneStep to execute processing for one clock period of the model at interrupt level.

 Off

Does not combine output and update function code into a single function, and instead generates the code in separate *model*_output and *model*_update functions.

Tips

Errors or unexpected behavior can occur if a Model block is part of a cycle, the Model block is a direct feedthrough block, and an algebraic loop results. For more information about direct feed through, see “Algebraic Loop Concepts”.

Simulink Coder ignores this parameter for a referenced model if any of the following conditions apply to that model:

- Is multi-rate
- Has a continuous sample time
- Is logging states (using the **States** or **Final states** parameters in the **Configuration Parameters** > **Data Import/Export** pane

Dependencies

- Setting **Code interface packaging** to C++ class forces on and disables this option.
- This option and **Classic call interface** are mutually incompatible and cannot both be selected through the GUI. Selecting **Classic call interface** forces off and disables this option and clearing **Classic call interface** enables (but does not select) this option.
- When you use this option, you must clear the option **Minimize algebraic loop occurrences** on the **Model Referencing** pane.
- If you customize ert_main.c or .cpp to read model outputs after each base-rate model step, selecting both parameters **Support: continuous time** and **Single output/update function** can cause output values read from ert_main for a continuous output port to differ from the corresponding output values in the logged data for the model. This is because, while logged data is a snapshot of output at major time steps, output read from ert_main after the base-rate model

step potentially reflects 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 the **Single output/update function** parameter), 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	
Select the **Single output/update function** parameter and insert code in the generated *`model`*`.c` or `.cpp` file to return 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: `CombineOutputUpdateFcns`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	On
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 12-2
- “rt_OneStep and Scheduling Considerations” (Embedded Coder)

MAT-file logging

Description

Specify MAT-file logging

Category: Code Generation > Interface

Settings

Default: on for the GRT target, off for ERT-based targets

 On

Enable MAT-file logging. When you select this option, the generated code saves to MAT-files simulation data specified in one of the following ways:

- **Configuration Parameters > Data Import/Export** (see “Model Configuration Parameters: Data Import/Export”)
- To Workspace blocks
- To File blocks
- Scope blocks with the **Log data to workspace** parameter enabled

In simulation, this data would be written to the MATLAB workspace, as described in “Export Simulation Data” and “Configure Signal Data for Logging”. Setting MAT-file logging redirects the data to a MAT-file instead. The file is named *`model`*`.mat`, where *`model`* is the name of your model.

 Off

Disable MAT-file logging. Clearing this option has the following benefits:

- Eliminates overhead associated with supporting a file system, which typically is not a requirement for embedded applications
- Eliminates extra code and memory usage required to initialize, update, and clean up logging variables
- Under certain conditions, eliminates code and storage associated with root output ports
- Omits the comparison between the current time and stop time in the *`model`*`_step`, allowing the generated program to run indefinitely, regardless of the stop time setting

Dependencies

- When you select **MAT-file logging**, you must also select the configuration parameters **Support: non-finite numbers** and, if you use an ERT-based system target file, **Support: floating-point numbers**.
- Selecting this option enables **MAT-file variable name modifier**.
- For ERT-based system target files, clear this parameter if you are using exported function calls.

Limitations

- The code generator does not support MAT-file logging for custom data types (data types that are not built into Simulink).
- MAT-file logging does not support file-scoped data, for example, data items to which you apply the built-in storage class `FileScope`.
- In a referenced model, only the following data logging features are supported:
  - To File blocks
  - State logging — the software stores the data in the MAT-file for the top model.
- In the context of the Embedded Coder product, MAT-file logging does not support the following IDEs: Analog Devices® VisualDSP++®, Texas Instruments™ Code Composer Studio™, Wind River® DIAB/GCC.
- MAT-file logging does not support Outport blocks to which you apply the storage class `ImportedExternPointer` or storage classes that yield nonaddressable data in the generated code. For example, the storage class `GetSet` causes the Outport to appear in the generated code as a function call, which is not addressable. This limitation applies whether you apply the storage class directly by using, for example, the Model Data Editor, or by resolving the Outport to a `Simulink.Signal` object that uses the storage class. As a workaround, apply the storage class to the signal that enters the Outport block.

Command-Line Information

Parameter: `MatFileLogging`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'` for the GRT target, `'off'` for ERT-based targets

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact
Efficiency	Off
Safety precaution	Off

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 12-2
- “Log Program Execution Results”
- “Log Data for Analysis”
- “Virtualized Output Ports Optimization” (Embedded Coder)
- “Virtualized Output Ports Optimization” (Embedded Coder)

Code Generation Parameters: GPU Code

Model Configuration Parameters: GPU Code

The **Code Generation > GPU Code** category includes parameters for configuring GPU-specific settings of the generated code.

These parameters require a GPU Coder license.

On the Configuration Parameters dialog box, the following configuration parameters are on the **Code Generation > GPU Code** pane.

Parameter	Description
“GPU index” on page 13-4	Specify the CUDA device to target.
“Compute capability” on page 13-5	Specify the minimum compute capability for which CUDA code is generated.
“Custom compute capability” on page 13-6	Specify the name of the NVIDIA virtual GPU architecture for code generation.
“Memory mode” on page 13-7	Specify the Memory allocation (`malloc`) mode to be used in the generated CUDA code.
“Maximum blocks per kernel” on page 13-9	Specify the maximum number of CUDA blocks created during a kernel launch.
“Dynamic memory allocation threshold” on page 13-10	Specify the size above which the private variables are allocated on the heap instead of the stack.
“Stack size per GPU thread” on page 13-11	Specify the maximum stack limit per GPU thread.
“Include error checks in generated code” on page 13-12	Add run-time error-checking functionality to the generated CUDA code.
“Kernel name prefix” on page 13-13	Specify custom kernel name prefixes.
“Additional compiler flags” on page 13-14	Specify additional flags to the `nvcc` compiler.
“cuBLAS” on page 13-15	Replacement of math function calls with NVIDIA `cuBLAS` library calls.
“cuSOLVER” on page 13-16	Replacement of math function calls with NVIDIA `cuSOLVER` library calls.
“cuFFT” on page 13-17	Replacement of math function calls with NVIDIA `cuFFT` library calls.

See Also

More About

- “Model Configuration”
- “Code Generation from Simulink Models with GPU Coder” (GPU Coder)

Code Generation: GPU Code Tab Overview

Set up GPU-specific information about code generation for a model's active configuration set, including device memory settings, CUDA libraries, code profiling and analysis, and device architecture and compiler.

These parameters require a GPU Coder license.

See Also

More About

- “Model Configuration Parameters: GPU Code” on page 13-2
- “Code Generation from Simulink Models with GPU Coder” (GPU Coder)

GPU index

Description

Specify the CUDA device to target.

Category: Code Generation > GPU Code

Settings

Default: -1

In a multi GPU environment such as NVIDIA Drive platforms, specify the CUDA device to target.

Note GPU index can be used with `gpuArray` only if `gpuDevice` and GPU index point to the same GPU. If `gpuDevice` points to a different GPU, a `CUDA_ERROR_INVALID_VALUE` runtime error is thrown.

Dependencies

- This parameter requires a GPU Coder license.
- To enable this parameter, select **Generate GPU code** on the **Code Generation** pane.

Command-Line Information

Parameter: `GPUDeviceID`
Type: integer
Value: `-1` or a valid `user-specified` index value
Default: `-1`

See Also

Related Examples

- “Model Configuration Parameters: GPU Code” on page 13-2
- “Code Generation from Simulink Models with GPU Coder” (GPU Coder)

Compute capability

Description

Specify the minimum compute capability of the GPU device for which CUDA code is generated.

Category: Code Generation > GPU Code

Settings

Default: `3.5`

Select the minimum compute capability for code generation. The compute capability identifies the features supported by the GPU hardware. It is used by applications at run time to determine which hardware features, instructions are available on the GPU device. If you specify custom compute capability, GPU Coder ignores this setting.

To see the CUDA compute capability requirements for code generation, consult the following table.

Target	Compute Capability
CUDA MEX	See “GPU Support by Release” (Parallel Computing Toolbox).
Source code, static or dynamic library, and executables	3.2 or higher.
Deep learning applications in 8-bit integer precision	6.1, 6.3 or higher.
Deep learning applications in half-precision (16-bit floating point)	5.3, 6.0, 6.2 or higher.

Dependencies

- This parameter requires a GPU Coder license.
- To enable this parameter, select **Generate GPU code** on the **Code Generation** pane.

Command-Line Information

Parameter: `GPUComputeCapability`
Type: character vector
Value: `'3.2'|'3.5'|'3.7'|'5.0'|'5.2'|'5.3'|'6.0'|'6.1'|'6.2'|'7.0'|'7.1'|'7.2'|'7.5'|`
Default: `'3.5'`

See Also

Related Examples

- “Model Configuration Parameters: GPU Code” on page 13-2
- “Code Generation from Simulink Models with GPU Coder” (GPU Coder)

Custom compute capability

Description

Specify the name of the NVIDIA virtual GPU architecture for code generation.

Category: Code Generation > GPU Code

Settings

Default: `''`

Specify the name of the NVIDIA virtual GPU architecture for which the CUDA input files must be compiled.

For example, to specify a virtual architecture type `-arch=compute_50`. You can specify a real architecture using `-arch=sm_50`. For more information, see the *Options for Steering GPU Code Generation* topic in the CUDA toolkit documentation.

Dependencies

- This parameter requires a GPU Coder license.
- To enable this parameter, select **Generate GPU code** on the **Code Generation** pane.

Command-Line Information

Parameter: `GPUCustomComputeCapability`
Type: character vector
Value: `''` or a valid `user-specified` virtual architecture specification
Default: `''`

See Also

Related Examples

- “Model Configuration Parameters: GPU Code” on page 13-2
- “Code Generation from Simulink Models with GPU Coder” (GPU Coder)

Memory mode

Description

Specify the memory allocation (`malloc`) mode to use in the generated CUDA code.

Category: Code Generation > GPU Code

Settings

Default: `discrete`

`discrete`

The generated code uses the `cudaMalloc` API for transferring data between the CPU and the GPU. From the programmers point-of-view, the discrete mode has a traditional memory architecture with separate CPU and GPU global memory address space.

`unified`

The generated code uses the `cudaMallocManaged` API that uses a shared (unified) CPU and GPU global memory address space.

For NVIDIA embedded targets only. See “Deprecating support for unified memory allocation mode on host” (GPU Coder).

Dependencies

- This parameter requires a GPU Coder license.
- To enable this parameter, select **Generate GPU code** on the **Code Generation** pane.

Command-Line Information

Parameter: `GPUMallocMode`
Type: character vector
Value: `'discrete'` | `'unified'`
Default: `'discrete'`

Compatibility Considerations

Deprecating support for unified memory allocation mode on host

In a future release, support for the unified memory allocation (`cudaMallocManaged`) mode will be removed when targeting NVIDIA GPU devices on the host development computer. When targeting GPU devices on the host, select `'discrete'` for the **Memory mode** parameter.

You can continue to use unified memory allocation mode when targeting NVIDIA embedded platforms.

See Also

Related Examples

- “Model Configuration Parameters: GPU Code” on page 13-2
- “Code Generation from Simulink Models with GPU Coder” (GPU Coder)

Maximum blocks per kernel

Description

Specify the maximum number of CUDA blocks created during a kernel launch.

Because GPU devices have limited streaming multiprocessor (SM) resources, limiting the number of blocks for each kernel can avoid performance losses from scheduling, loading and unloading of blocks.

If the number of iterations in a loop is greater than the maximum number of blocks per kernel, the code generator creates CUDA kernels with striding.

When you specify the maximum number of blocks for each kernel, the code generator creates 1-D kernels. To force the code generator to create 2-D or 3-D kernels, use the `coder.gpu.kernel` pragma. The `coder.gpu.kernel` pragma takes precedence over the maximum number of kernels for each CUDA block.

Category: Code Generation > GPU Code

Settings

Default: 0

Specify the maximum number of CUDA blocks created during a kernel launch.

Dependencies

- This parameter requires a GPU Coder license.
- To enable this parameter, select **Generate GPU code** on the **Code Generation** pane.

Command-Line Information

Parameter: `GPUMaximumBlocksPerKernel`
Type: integer
Value: any valid value
Default: `0`

See Also

Related Examples

- “Model Configuration Parameters: GPU Code” on page 13-2
- “Code Generation from Simulink Models with GPU Coder” (GPU Coder)

Dynamic memory allocation threshold

Description

Specify the memory allocation threshold.

Category: Code Generation > GPU Code

Settings

Default: 200

Specify the size above which the private variables are allocated on the heap instead of the stack, as an integer value.

Dependencies

- This parameter requires a GPU Coder license.
- To enable this parameter, select **Generate GPU code** on the **Code Generation** pane.

Command-Line Information

Parameter: `GPUMallocThreshold`
Type: integer
Value: any valid value
Default: `200`

See Also

Related Examples

- “Model Configuration Parameters: GPU Code” on page 13-2
- “Code Generation from Simulink Models with GPU Coder” (GPU Coder)

Stack size per GPU thread

Description

Specify the stack limit per GPU thread.

Category: Code Generation > GPU Code

Settings

Default: 1024

Specify the maximum stack limit per GPU thread as an integer value.

Dependencies

- This parameter requires a GPU Coder license.
- To enable this parameter, select **Generate GPU code** on the **Code Generation** pane.

Command-Line Information

Command-Line Information
Parameter: `GPUStackLimitPerThread`
Type: integer
Value: any valid value
Default: `1024`

See Also

Related Examples

- “Model Configuration Parameters: GPU Code” on page 13-2
- “Code Generation from Simulink Models with GPU Coder” (GPU Coder)

Include error checks in generated code

Description

Add run-time error-checking functionality to the generated CUDA code.

Category: Code Generation > GPU Code

Settings

Default: Off

On

Generates code with error-checking for CUDA API and kernel calls and performs run-time checks.

Off

The generated CUDA code does not contain error-checking functionality.

Dependencies

- This parameter requires a GPU Coder license.
- To enable this parameter, select **Generate GPU code** on the **Code Generation** pane.

Command-Line Information

Parameter: `GPUErrorChecks`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

Related Examples

- “Model Configuration Parameters: GPU Code” on page 13-2
- “Code Generation from Simulink Models with GPU Coder” (GPU Coder)

Kernel name prefix

Description

Specify custom kernel name prefixes.

Category: Code Generation > GPU Code

Settings

Default: `''`

Specify a custom name prefix for all the kernels in the generated code. For example, using the value `'CUDA_'` creates kernels with names `CUDA_kernel1`, `CUDA_kernel2`, and so on. If no name is provided, GPU Coder prepends the kernel name with the name of the entry-point function. Kernel names can contain upper-case letters, lowercase letters, digits 0–9, and underscore character _. GPU Coder removes unsupported characters from the kernel names and appends `alpha` to prefixes that do not begin with an alphabetic letter.

Dependencies

- This parameter requires a GPU Coder license.
- To enable this parameter, select **Generate GPU code** on the **Code Generation** pane.

Command-Line Information

Parameter: `GpuKernelNamePrefix`
Type: character vector
Value: `''` or a valid `user-specified` name
Default: `''`

See Also

Related Examples

- “Model Configuration Parameters: GPU Code” on page 13-2
- “Code Generation from Simulink Models with GPU Coder” (GPU Coder)

Additional compiler flags

Description

Specify additional flags to the NVIDIA `nvcc` compiler.

Category: Code Generation > GPU Code

Settings

Default: `''`

Pass additional flags to the GPU compiler. For example, `--fmad=false` instructs the `nvcc` compiler to disable contraction of floating-point multiply and add to a single Floating-Point Multiply-Add (FMAD) instruction.

For similar NVIDIA compiler options, see the topic on *NVCC Command Options* in the CUDA toolkit documentation.

Dependencies

- This parameter requires a GPU Coder license.
- To enable this parameter, select **Generate GPU code** on the **Code Generation** pane.

Command-Line Information

Parameter: `GPUCompilerFlags`
Type: character vector
Value: `''` or a valid `user-specified` flag
Default: `''`

See Also

Related Examples

- “Model Configuration Parameters: GPU Code” on page 13-2
- “Code Generation from Simulink Models with GPU Coder” (GPU Coder)

cuBLAS

Description

Replacement of math function calls with NVIDIA `cuBLAS` library calls.

Category: Code Generation > GPU Code

Settings

Default: On

On

Allows GPU Coder to replace appropriate math function calls with calls to the `cuBLAS` library. For functions that have no replacements in CUDA, GPU Coder uses portable MATLAB functions and attempts to map them to the GPU.

Off

Disable the use of the `cuBLAS` library in the generated code.

Dependencies

- This parameter requires a GPU Coder license.
- To enable this parameter, select **Generate GPU code** on the **Code Generation** pane.

Command-Line Information

Parameter: `GPUcuBLAS`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

See Also

Related Examples

- “Model Configuration Parameters: GPU Code” on page 13-2
- “Code Generation from Simulink Models with GPU Coder” (GPU Coder)

cuSOLVER

Description

Replacement of math function calls with NVIDIA `cuSOLVER` library calls.

Category: Code Generation > GPU Code

Settings

Default: On

On

Allows GPU Coder to replace appropriate math function calls with calls to the `cuSOLVER` library. For functions that have no replacements in CUDA, GPU Coder uses portable MATLAB functions and attempts to map them to the GPU.

Off

Disable the use of the `cuSOLVER` library in the generated code.

Dependencies

- This parameter requires a GPU Coder license.
- To enable this parameter, select **Generate GPU code** on the **Code Generation** pane.

Command-Line Information

Parameter: `GPUcuSOLVER`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

See Also

Related Examples

- “Model Configuration Parameters: GPU Code” on page 13-2
- “Code Generation from Simulink Models with GPU Coder” (GPU Coder)

cuFFT

Description

Replacement of math function calls with NVIDIA `cuFFT` library calls.

Category: Code Generation > GPU Code

Settings

Default: On

On

Allows GPU Coder to replace appropriate `fft` calls with calls to the `cuFFT` library.

Off

Disables use of the `cuFFT` library in the generated code. With this option, GPU Coder uses C `FFTW` libraries where available or generates kernels from portable MATLAB `fft` code.

Dependencies

- This parameter requires a GPU Coder license.
- To enable this parameter, select **Generate GPU code** on the **Code Generation** pane.

Command-Line Information

Parameter: `GPUcuFFT`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

See Also

Related Examples

- “Model Configuration Parameters: GPU Code” on page 13-2
- “Code Generation from Simulink Models with GPU Coder” (GPU Coder)

Simulink Coder Parameters: Advanced Parameters

Use Simulink Coder Features

Description

Enable “Simulink Coder” features for models deployed to “Simulink Supported Hardware”.

Note If you enable this parameter in a model where Simulink Coder is not installed or available in the environment, a question dialog box prompts you to update the model to build without Simulink Coder features.

Category: Hardware Implementation

Settings

 On

Enable the Simulink Coder features.

 Off

Disable the Simulink Coder features.

Indicates that this parameter is enabled. To disable it, first disable the “Use Embedded Coder Features” (Embedded Coder) parameter.

Dependencies

This parameter requires a Simulink Coder or Embedded Coder license.

Command-Line Information

Parameter: `UseSimulinkCoderFeatures`
Value: `'on'` or `'off'`
Default: `'on'`

See Also

Related Examples

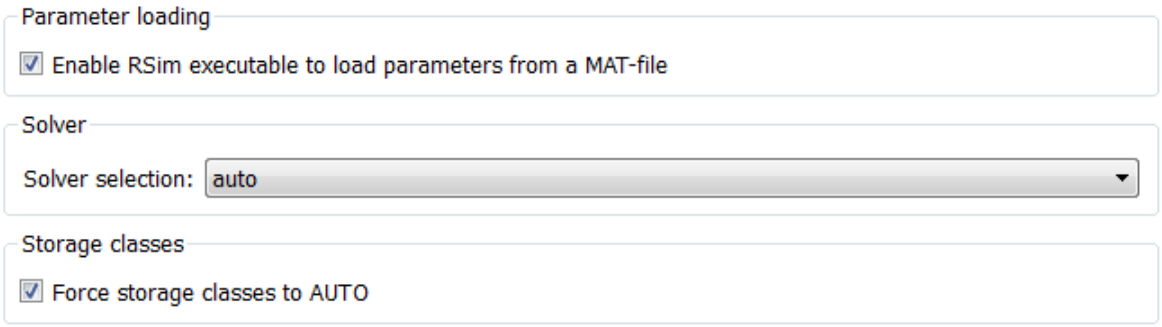
- “Model Configuration”

Configuration Parameters for Simulink Models

- “Code Generation Pane: RSim Target” on page 15-2
- “Code Generation Pane: S-Function Target” on page 15-5
- “Code Generation Pane: Tornado Target” on page 15-8
- “Recommended Settings Summary for Model Configuration Parameters” on page 15-22

Code Generation Pane: RSim Target

The **Code Generation** > **RSim Target** pane includes these parameters when the Simulink Coder product is installed on your system and you specify the `rsim.tlc` system target file.



In this section...
“Code Generation: RSim Target Tab Overview” on page 15-2
“Enable RSim executable to load parameters from a MAT-file” on page 15-2
“Solver selection” on page 15-3
“Force storage classes to AUTO” on page 15-4

Code Generation: RSim Target Tab Overview

Set configuration parameters for rapid simulation.

Configuration

This tab appears only if you specify `rsim.tlc` as the “System target file” on page 7-7.

See Also

- “Configure and Build Model for Rapid Simulation”
- “Run Rapid Simulations”
- “Code Generation Pane: RSim Target” on page 15-2

Enable RSim executable to load parameters from a MAT-file

Specify whether to load RSim parameters from a MAT-file.

Settings

Default: on

 On

Enables RSim to load parameters from a MAT-file.

 Off

Disables RSim from loading parameters from a MAT-file.

Command-Line Information

Parameter: `RSIM_PARAMETER_LOADING`
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

"Create a MAT-File That Includes a Model Parameter Structure"

Solver selection

Instruct the target how to select the solver.

Settings

Default: `auto`

`auto`

Lets the code generator choose the solver. The code generator uses the Simulink solver module if you specify a variable-step solver on the Solver pane. Otherwise, the code generator uses a built-in solver.

`Use Simulink solver module`

Instructs the code generator to use the variable-step solver that you specify on the **Solver** pane.

`Use fixed-step solvers`

Instructs the code generator to use the fixed-step solver that you specify on the **Solver** pane.

Command-Line Information

Parameter: `RSIM_SOLVER_SELECTION`
Type: character vector
Value: `'auto'` | `'usesolvermodule'` | `'usefixstep'`
Default: `'auto'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Force storage classes to AUTO

Specify whether to retain your storage class settings in a model or to use the automatic settings.

Settings

Default: on

 On

Forces the Simulink software to determine storage classes.

 Off

Causes the model to retain storage class settings.

Tips

- Turn this parameter on for flexible custom code interfacing.
- Turn this parameter off to retain storage class settings such as `ExportedGlobal` or `ImportExtern`.

Command-Line Information

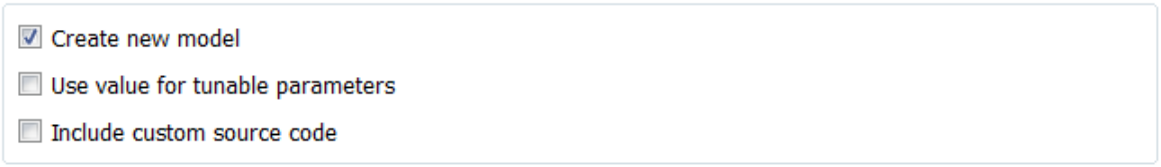
Parameter: `RSIM_STORAGE_CLASS_AUTO`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Code Generation Pane: S-Function Target

The **Code Generation** > **S-Function Target** pane includes the following parameters when the Simulink Coder product is installed on your system and you specify the `rtwsfcn.tlc` system target file.



In this section...
“Code Generation S-Function Target Tab Overview” on page 15-5
“Create new model” on page 15-5
“Use value for tunable parameters” on page 15-6
“Include custom source code” on page 15-6

Code Generation S-Function Target Tab Overview

Control code generated for the S-function target (`rtwsfcn.tlc`).

Configuration

This tab appears only if you specify the S-function target (`rtwsfcn.tlc`) as the “System target file” on page 7-7.

See Also

- “Accelerate Simulation, Reuse Code, or Protect Intellectual Property by Using S-Function Target”
- “Code Generation Pane: S-Function Target” on page 15-5

Create new model

Create a new model containing the generated S-function block.

Settings

Default: on

 On

Creates a new model, separate from the current model, containing the generated S-function block.

 Off

Generates code but a new model is not created.

Command-Line Information

Parameter: `CreateModel`

Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

See Also

“Accelerate Simulation, Reuse Code, or Protect Intellectual Property by Using S-Function Target”

Use value for tunable parameters

Use the variable value instead of the variable name in generated block mask edit fields for tunable parameters.

Settings

Default: off

 On

Uses variable values for tunable parameters instead of the variable name in the generated block mask edit fields.

 Off

Uses variable names for tunable parameters in the generated block mask edit fields.

Command-Line Information
Parameter: `UseParamValues`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

“Accelerate Simulation, Reuse Code, or Protect Intellectual Property by Using S-Function Target”

Include custom source code

Include custom source code in the code generated for the S-function.

Settings

Default: off

 On

Include provided custom source code in the code generated for the S-function.

 Off

Do not include custom source code in the code generated for the S-function.

Command-Line Information
Parameter: `AlwaysIncludeCustomSrc`
Type: character vector

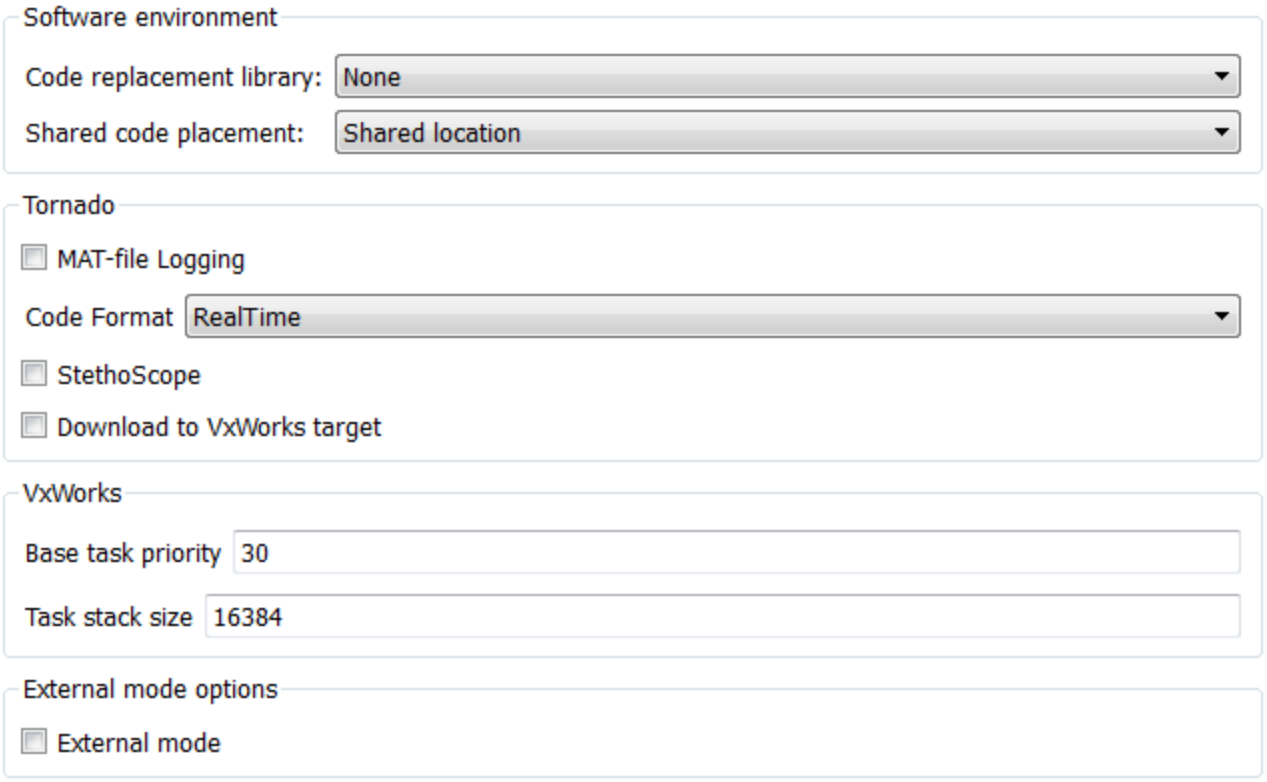
Value: `'on'` | `'off'`
Default: `'off'`

See Also

“Accelerate Simulation, Reuse Code, or Protect Intellectual Property by Using S-Function Target”

Code Generation Pane: Tornado Target

The **Code Generation** > **Tornado Target** pane includes the following parameters when the Simulink Coder product is installed on your system and you specify the `tornado.tlc` system target file.



In this section...
“Code Generation: Tornado Target Tab Overview” on page 15-9 “Standard math library” on page 15-9 “Code replacement library” on page 15-10 “Shared code placement” on page 15-11 “MAT-file logging” on page 15-12 “MAT-file variable name modifier” on page 15-13 “Code Format” on page 15-14 “StethoScope” on page 15-14 “Download to VxWorks target” on page 15-15 “Base task priority” on page 15-16 “Task stack size” on page 15-17 “External mode” on page 15-17 “Transport layer” on page 15-18 “MEX-file arguments” on page 15-19

In this section...
“Static memory allocation” on page 15-20 “Static memory buffer size” on page 15-20

Code Generation: Tornado Target Tab Overview

Control generated code for the Tornado target.

Configuration

This tab appears only if you specify `tornado.tlc` as the “System target file” on page 7-7.

See Also

- *Tornado User's Guide* from Wind River Systems
- *StethoScope User's Guide* from Wind River Systems
- “Asynchronous Support”
- “Code Generation Pane: Tornado Target” on page 15-8

Standard math library

Specify a standard math library for your model.

Settings

Default: `C99 (ISO)`

`C89/C90 (ANSI)`

Generates calls to the ISO/IEC 9899:1990 C standard math library.

`C99 (ISO)`

Generates calls to the ISO/IEC 9899:1999 C standard math library.

`C++03 (ISO)`

Generates calls to the ISO/IEC 14882:2003 C++ standard math library.

Tips

- The build process checks whether the specified standard math library and toolchain are compatible. If they are not compatible, a warning occurs during code generation and the build process continues.
- When you change the value of parameter **Language**, the standard math library updates to ISO/IEC 9899:1999 C (`C99 (ISO)`) for C and ISO/IEC 14882:2003 C++ (`C++03 (ISO)`) for C++.

Dependencies

The `C++03 (ISO)` math library is available for use only if you set parameter **Language** to `C++`.

Command-Line Information

Parameter: `TargetLangStandard`
Type: character vector

Value: `'C89/C90 (ANSI)'|'C99 (ISO)'|'C++03 (ISO)'`
Default: `'C99 (ISO)'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Valid library
Safety precaution	No impact

See Also

"Run-Time Environment Configuration"

Code replacement library

Specify a code replacement library the code generator uses when producing code for a model.

Settings

Default: `None`

`None`

Does not use a code replacement library.

`Named code replacement library`

Generates calls to a specific platform, compiler, or standards code replacement library. The list of named libraries depends on:

- Installed support packages.
- System target file, language, standard math library, and device vendor configuration.
- Whether you created and registered code replacement libraries, using the Embedded Coder product.

For more information about selections for this parameter, see "Code replacement library" on page 12-9.

Tip

Before setting this parameter, verify that your compiler supports the library you want to use. If you select a parameter value that your compiler does not support, compiler errors can occur.

Command-Line Information

Parameter: `CodeReplacementLibrary`
Type: character vector
Value: `'None'|'GNU C99 extensions'|'Intel IPP for x86-64 (Windows)'|'Intel IPP/SSE for x86-64 (Windows)'|'Intel IPP for x86-64 (Windows for MinGW compiler)'|'Intel IPP/SSE for x86-64 (Windows for MinGW compiler)'|'Intel IPP for x86/Pentium (Windows)'|'Intel IPP/SSE x86/Pentium (Windows)'|'Intel IPP for x86-64 (Linux)'|'Intel IPP/SSE with GNU99 extensions for x86-64 (Linux)'`

Default: `'None'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Valid library
Safety precaution	No impact

See Also

"Run-Time Environment Configuration"

Shared code placement

Specify the location for generating utility functions, exported data type definitions, and declarations of exported data with custom storage class.

Settings

Default: `Auto`

`Auto`

Operates as follows:

- When the model contains Model blocks, places utility code within the `slprj/target/_sharedutils` folder.
- When the model does not contain Model blocks, places utility code in the build folder (generally, in *`model`*`.c` or *`model`*`.cpp`).

`Shared location`

Directs code for utilities to be placed within the `slprj` folder in your working folder.

Command-Line Information

Parameter: `UtilityFuncGeneration`
Type: character vector
Value: `'Auto'` | `'Shared location'`
Default: `'Auto'`

Recommended Settings

Application	Setting
Debugging	`Shared location`
Traceability	`Shared location`
Efficiency	No impact (execution, RAM) `Shared location` (ROM)
Safety precaution	No impact

See Also

- “Run-Time Environment Configuration”
- “Sharing Utility Code”

MAT-file logging

Specify whether to enable MAT-file logging.

Settings

Default: off

 On

Enables MAT-file logging. When you select this option, the generated code saves to MAT-files simulation data specified in one of the following ways:

- Configuration Parameters dialog box, **Data Import/Export** pane (see “Model Configuration Parameters: Data Import/Export”)
- To Workspace blocks
- Scope blocks with block parameter **Log data to workspace** enabled

In simulation, this data would be written to the MATLAB workspace, as described in “Export Simulation Data” and “Configure Signal Data for Logging”. Setting MAT-file logging redirects the data to a MAT-file instead. The file is named *`model`*`.mat`, where *`model`* is the name of your model.

 Off

Disables MAT-file logging. Clearing this option has the following benefits:

- Eliminates overhead associated with supporting a file system, which typically is not required for embedded applications
- Eliminates extra code and memory usage required to initialize, update, and clean up logging variables
- Under certain conditions, eliminates code and storage associated with root output ports
- Omits the comparison between the current time and stop time in the *`model`*`_step`, allowing the generated program to run indefinitely, regardless of the stop time setting

Dependencies

Selecting this parameter enables parameter **MAT-file variable name modifier**.

Limitation

MAT-file logging does not support file-scoped data, for example, data items to which you apply the built-in custom storage class `FileScope`.

MAT-file logging does not work in a referenced model, and code is not generated to implement it.

Command-Line Information

Parameter: `MatFileLogging`
Type: character vector

Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact
Efficiency	Off
Safety precaution	Off

See Also

- “Log Program Execution Results”
- “Log Data for Analysis”
- “Virtualized Output Ports Optimization” (Embedded Coder)

MAT-file variable name modifier

Select the text to add to the MAT-file variable names.

Settings

Default: `rt_`

`rt_`
: Adds prefix text.

`_rt`
: Adds suffix text.

`none`
: Does not add text.

Dependency

If you have an Embedded Coder license, this parameter is enabled by parameter **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

Application	Setting
Safety precaution	No impact

See Also

- “Log Program Execution Results”
- “Log Data for Analysis”

Code Format

Specify the code format (generated code features).

Settings

Default: `RealTime`

`RealTime`
: Specifies the Real-Time code generation format.

`RealTimeMalloc`
: Specifies the Real-Time Malloc code generation format.

Command-Line Information
Parameter: `CodeFormat`
Type: character vector
Value: `'RealTime'` | `'RealTimeMalloc'`
Default: `'RealTime'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

“Compare System Target File Support Across Products”

StethoScope

Specify whether to enable StethoScope, an optional data acquisition and data monitoring tool.

Settings

Default: off

☑ On
: Enables StethoScope.

Off

Disables StethoScope.

Tips

You can optionally monitor and change the parameters of the executing real-time program using either StethoScope or Simulink External mode, but not both with the same compiled image.

Dependencies

Enabling parameter **StethoScope** disables parameter **External mode**, and vice versa.

Command-Line Information

Parameter: `StethoScope`
Type: character vector
Value: `'on' | 'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact
Efficiency	Off
Safety precaution	Off

See Also

- *Tornado User's Guide* from Wind River Systems
- *StethoScope User's Guide* from Wind River Systems

Download to VxWorks target

Specify whether to automatically download the generated program to the VxWorks target.

Settings

Default: off

On

Automatically downloads the generated program to VxWorks after each build.

Off

Does not automatically download to VxWorks, you must downloaded generated programs manually.

Tips

- Automatic download requires specifying the target name and host name in the makefile.

- Before every build, reset VxWorks by pressing **Ctrl+X** on the host console or power-cycling the VxWorks chassis. This clears dangling processes or stale data that exists in VxWorks when the automatic download occurs.

Command-Line Information
Parameter: `DownloadToVxWorks`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	Off

See Also

- *Tornado User's Guide* from Wind River Systems
- "Asynchronous Support"

Base task priority

Specify the priority with which the base rate task for the model is to be spawned.

Settings

Default: 30

Tips

- For a multirate, multitasking model, the code generator increments the priority of each subrate task by one.
- The value you specify for this option will be overridden by a base priority specified in a call to the `rt_main()` function spawned as a task.

Command-Line Information
Parameter: `BasePriority`
Type: integer
Value: valid value
Default: `30`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	Might impact efficiency, depending on other task's priorities
Safety precaution	No impact

See Also

- *Tornado User's Guide* from Wind River Systems
- “Asynchronous Support”

Task stack size

Stack size in bytes for each task that executes the model.

Settings

Default: 16384

Command-Line Information
Parameter: `TaskStackSize`
Type: integer
Value: valid value
Default: 16384

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Larger stack may waste space
Safety precaution	Larger stack reduces the possibility of overflow

See Also

- *Tornado User's Guide* from Wind River Systems
- “Asynchronous Support”

External mode

Specify whether to enable communication between the Simulink model and an application based on a client/server architecture.

Settings

Default: on

 On

Enables External mode. The client (Simulink model) transmits messages requesting the server (application) to accept parameter changes or to upload signal data. The server responds by executing the request.

Off

Disables External mode.

Dependencies

Selecting this parameter enables these parameters:

- **Transport layer**
- **MEX-file arguments**
- **Static memory allocation**

Command-Line Information

Parameter: `ExtMode`
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

"External Mode Simulations for Parameter Tuning and Signal Monitoring"

Transport layer

Specify the transport protocol for External mode communications.

Settings

Default: `tcpip`

`tcpip`

Applies a TCP/IP transport mechanism. The MEX-file name is `ext_comm`.

Tip

You cannot edit the value of the **MEX-file name** parameter displayed next to parameter **Transport layer**. For system target files provided by MathWorks, the value is specified in *`matlabroot`*`/toolbox/simulink/simulink/extmode_transports.m`.

Dependency

This parameter is enabled by parameter **External mode**.

Command-Line Information
Parameter: `ExtModeTransport`
Type: integer
Value: `0`
Default: `0`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

“External Mode Simulation with TCP/IP or Serial Communication”

MEX-file arguments

Specify arguments to pass to an External mode interface MEX-file for communicating with executing targets.

Settings

Default: `''`

For TCP/IP interfaces, `ext_comm` allows three optional arguments:

- Network name of your target (for example, `'myPuter'` or `'148.27.151.12'`)
- Verbosity level (`0` for no information or `1` for detailed information)
- TCP/IP server port number (an integer value between `256` and `65535`, with a default of `17725`)

Dependency

This parameter is enabled by parameter **External mode**.

Command-Line Information
Parameter: `ExtModeMexArgs`
Type: character vector
Value: valid arguments
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- “External Mode Simulation with TCP/IP or Serial Communication”
- “Choose Communication Protocol for Client and Server”

Static memory allocation

Control the memory buffer for External mode communication.

Settings

Default: off

 On

Enables parameter **Static memory buffer size** for allocating dynamic memory.

 Off

Uses a static memory buffer for External mode instead of allocating dynamic memory (calls to malloc).

Tip

To determine how much memory you need to allocate, select verbose mode on the target to display the amount of memory it tries to allocate and the amount of memory available.

Dependencies

- This parameter is enabled by parameter **External mode**.
- This parameter enables parameter **Static memory buffer size**.

Command-Line Information

Parameter: `ExtModeStaticAlloc`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

“Control Memory Allocation for Communication Buffers in Target”

Static memory buffer size

Specify the memory buffer size for External mode communication.

Settings

Default: `1000000`

Enter the number of bytes to preallocate for External mode communications buffers in the target.

Tips

- If you enter too small a value for your application, External mode issues an out-of-memory error.
- To determine how much memory you need to allocate, select verbose mode on the target to display the amount of memory it tries to allocate and the amount of memory available.

Dependency

This parameter is enabled by parameter **Static memory allocation**.

Command-Line Information

Parameter: `ExtModeStaticAllocSize`
Type: integer
Value: valid value
Default: `1000000`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

“Control Memory Allocation for Communication Buffers in Target”

Recommended Settings Summary for Model Configuration Parameters

The following table summarizes the impact of each configuration parameter on debugging, traceability, efficiency, and safety considerations, and indicates the factory default configuration settings for the GRT and ERT targets, unless otherwise specified.

For parameters that are available only when an ERT target is specified, see “Recommended Settings Summary for Model Configuration Parameters” (Embedded Coder).

For additional details, click the links in the Configuration Parameter column.

Mapping Application Requirements to the Solver Pane

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Start time"	No impact	No impact	No impact	0.0	0.0 seconds
"Stop time"	No impact	No impact	No impact	A positive value	10.0 seconds
"Type"	`Fixed-step`	`Fixed-step`	`Fixed-step`	`Fixed-step`	`Variable-step` (you must change to `Fixed-step` for code generation)
"Solver"	No impact	No impact	No impact	`Discrete (no continuous states)`	`ode3 (Bogacki-Shampine)`
"Periodic sample time constraint"	No impact	No impact	No impact	`Specified` or `Ensure sample time independent`	`Unconstrained`
"Sample time properties"	No impact	No impact	No impact	Period, offset, and priority of each sample time in the model; faster sample times must have higher priority than slower sample times	`''`
"Treat each discrete rate as a separate task"	No impact	No impact	No impact	No impact	On
"Automatically handle rate transition for data transfer"	No impact	No impact (for simulation and during development) Off (for production code generation)	No impact	Off	Off

Mapping Application Requirements to the Data Import/Export Pane

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
“Input”	No impact	No impact	No recommendation	No recommendation	Off
“Initial state”	No impact	No impact	No recommendation	No recommendation	Off
“Time”	No impact	No impact	No recommendation	No recommendation	On
“States”	No impact	No impact	No recommendation	No recommendation	Off
“Output”	No impact	No impact	No recommendation	No recommendation	On
“Final states”	No impact	No impact	No recommendation	No recommendation	Off
“Signal logging”	No impact	No impact	No recommendation	No recommendation	On
“Record logged workspace data in Simulation Data Inspector”	No impact	No impact	No recommendation	No recommendation	Off
“Limit data points”	No impact	No impact	No recommendation	No recommendation	On
“Decimation”	No impact	No impact	No recommendation	No recommendation	1
“Format”	No impact	No impact	No recommendation	No recommendation	`Array`
“Output options”	No impact	No impact	No recommendation	No recommendation	`Refine output`
“Refine factor”	No impact	No impact	No recommendation	No recommendation	1

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Output times"	No impact	No impact	No recommendation	No recommendation	`'[]'`

Mapping Application Requirements to the Math and Data Types Pane

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Default for underspecified data type"	No impact	No impact	`single`	No impact	`double`
"Use division for fixed-point net slope computation"	No impact	No impact	On (when target hardware supports efficient division) Off (otherwise)	No impact	`off`
"Application lifespan (days)"	No impact	No impact	Finite value	`inf`	`auto`
"Use floating-point multiplication to handle net slope corrections"	No impact	No impact	On (when target hardware supports efficient multiplication) Off (otherwise)	No recommendation	Off
*The command-line value is reverse of the listed value.					

Mapping Application Requirements to the Optimization Pane

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Default parameter behavior" on page 19-9	`Tunable` (GRT) `Inlined` (ERT)	`Inlined`	`Inlined`	No impact	`Tunable` (GRT) `Inlined` (ERT)
"Loop unrolling threshold" on page 19-16	No impact	No impact	>0	No impact	5
"Maximum stack size (bytes)" on page 19-17	No impact	No impact	No impact	No impact	`Inherit from target`
"Use memcpy for vector assignment" on page 19-13	No impact	No impact	On	No impact	On
"Memcpy threshold (bytes)" on page 19-15	No impact	No impact	Accept default or determine target-specific optimal value	No impact	64
"Inline invariant signals" on page 19-11	Off	Off	On	No impact	Off
"Remove code from floating-point to integer conversions that wraps out-of-range values" on page 19-6	Off	Off	On (execution, ROM) No impact (RAM)	No impact	Off
"Use bitsets for storing state configuration" on page 19-19	Off	Off	Off (execution, ROM) On (RAM)	No impact	Off
"Use bitsets for storing Boolean data" on page 19-21	Off	Off	Off (execution, ROM) On (RAM)	No impact	Off

Mapping Application Requirements to the Diagnostics Pane: Solver Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Algebraic loop"	`error`	No impact	No impact	`error`	`warning`
"Minimize algebraic loop"	No impact	No impact	No impact	`error`	`warning`
"Block priority violation"	No impact	No impact	No impact	`error`	`warning`
"Consecutive zero-crossings violation"	No impact	No impact	No impact	`warning` or `error`	`error`
"Unspecified inheritability of sample time"	No impact	No impact	No impact	`error`	`warning`
"Solver data inconsistency"	`warning`	No impact	`none`	No impact	`warning`
"Automatic solver parameter selection"	No impact	No impact	No impact	`error`	`warning`

Mapping Application Requirements to the Diagnostics Pane: Sample Time Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Source block specifies -1 sample time"	No impact	No impact	No impact	`error`	`none`
"Multitask data transfer"	No impact	No impact	No impact	`error`	`error`
"Single task data transfer"	No impact	No impact	No impact	`none` or `error`	`none`
"Multitask conditionally executed subsystem"	No impact	No impact	No impact	`error`	`error`
"Tasks with equal priority"	No impact	No impact	No impact	`none` or `error`	`warning`
"Enforce sample times specified by Signal Specification blocks"	No impact	No impact	No impact	`error`	`warning`

Mapping Application Requirements to the Diagnostics Pane: Data Validity Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Signal resolution"	No impact	No impact	No impact	`Explicit only`	`Explicit only`
"Division by singular matrix"	No impact	No impact	No impact	`error`	`none`
"Underspecified data types"	No impact	No impact	No impact	`error`	`none`
"Simulation range checking"	`warning` or `error`	`warning` or `error`	`none`	`error`	`none`
"Wrap on overflow"	No impact	No impact	No impact	`error`	`warning`
"Saturate on overflow"	No impact	No impact	No impact	`error`	`warning`
"Inf or NaN block output"	No impact	No impact	No impact	`error`	`none`
""rt" prefix for identifiers"	No impact	No impact	No impact	`error`	`error`
"Detect downcast"	No impact	No impact	No impact	`error`	`error`
"Detect overflow"	No impact	No impact	No impact	`error`	`error`
"Detect underflow"	No impact	No impact	No impact	`error`	`none`
"Detect precision loss"	No impact	No impact	No impact	`error`	`error`
"Detect loss of tunability"	No impact	No impact	No impact	`error`	`warning` for GRT-based targets `error` for ERT-based targets
"Detect read before write"	No impact	No impact	No impact	`error`	`Enable all as warnings`
"Detect write after read"	No impact	No impact	No impact	`error`	`Enable all as warning`
"Detect write after write"	No impact	No impact	No impact	`error`	`Enable all as errors`
"Multitask data store"	No impact	No impact	No impact	`error`	`warning`
"Duplicate data store names"	`warning`	No impact	`none`	No impact	`none`
"Check undefined subsystem initial output"	No impact	No impact	No impact	On	On

Mapping Application Requirements to the Diagnostics Pane: Type Conversion Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Unnecessary type conversions"	No impact	No impact	No impact	`warning`	`none`
"Vector/matrix block input conversion"	No impact	No impact	No impact	`error`	`none`
"32-bit integer to single precision float conversion"	No impact	No impact	No impact	`warning`	`warning`

Mapping Application Requirements to the Diagnostics Pane: Connectivity Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Signal label mismatch"	No impact	No impact	No impact	`error`	`none`
"Unconnected block input ports"	No impact	No impact	No impact	`error`	`warning`
"Unconnected block output ports"	No impact	No impact	No impact	`error`	`warning`
"Unconnected line"	No impact	No impact	No impact	`error`	`none`
"Unspecified bus object at root Outport block"	No impact	No impact	No impact	`error`	`warning`
"Element name mismatch"	No impact	No impact	No impact	`error`	`warning`
"Bus signal treated as vector"	No impact	No impact	No impact	`error`	`none`
"Context-dependent inputs"	No impact	No impact	No impact	`Enable all`	`Use local settings`

Mapping Application Requirements to the Diagnostics Pane: Compatibility Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"S-function upgrades needed"	No impact	No impact	No impact	`error`	`none`

Mapping Application Requirements to the Diagnostics Pane: Model Referencing Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Model block version mismatch"	No impact	No impact	No impact	No recommendation	`none`
"Port and parameter mismatch"	No impact	No impact	No impact	`error`	`none`
"Invalid root Inport/ Outport block connection"	No impact	No impact	No impact	`error`	`none`
"Unsupported data logging"	No impact	No impact	No impact	`error`	`warning`

Mapping Application Requirements to the Diagnostics Pane: Saving Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Block diagram contains disabled library links"	No impact	No impact	No impact	No impact	`warning`
"Block diagram contains parameterized library links"	No impact	No impact	No impact	No impact	`none`

Mapping Application Requirements to the Diagnostics Pane: Stateflow Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Unused data, events, messages, and functions"	`warning`	No impact	No impact (for simulation and during development) `none` (for production code generation)	`warning`	`warning`
"Unexpected backtracking"	`warning`	No impact	No impact	`error`	`warning`
"Invalid input data access in chart initialization"	`warning`	No impact	No impact	`error`	`warning`
"No unconditional default transitions"	`warning`	No impact	No impact (for simulation and during development) `none` (for production code generation)	`error`	`warning`
"Transition outside natural parent"	`warning`	No impact	No impact (for simulation and during development) `none` (for production code generation)	`error`	`warning`

Mapping Application Requirements to the Hardware Implementation Pane

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Device vendor"	No impact	No impact	No impact	Select your **Device vendor** and **Device type** if they are available in the drop-down list. If your **Device vendor** and **Device type** are not available, set device-specific values by using `Custom Processor`.	`None` if specified system target file is `ert.tlc`, `realtime.tlc`, or `autosar.tlc`. Otherwise, `Determine by Code Generation system target file`
"Device vendor"	No impact	No impact	No impact	Select your **Device vendor** and **Device type** if they are available in the drop-down list. If your **Device vendor** and **Device type** are not available, set device-specific values by using `Custom Processor`.	`Intel`
"Device type"	No impact	No impact	No impact	Select your **Device vendor** and **Device type** if they are available in the drop-down list. If your **Device vendor** and **Device type** are not available, set device-specific values by using `Custom Processor`.	`x86–64 (Windows64)`

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Number of bits: char"	No impact	No impact	Target specific	No recommendation for simulation without code generation. For simulation with code generation, select your **Device vendor** and **Device type** if they are available in the drop-down list. If your **Device vendor** and **Device type** are not available, set device-specific values by using `Custom Processor`.	`char` 8, `short` 16, `int` 32, `long` 32, `long long` 64, `float` 32, `double` 64, `native` 32, `pointer` 32
"Largest atomic size: integer"	No impact	No impact	Target specific	No recommendation for simulation without code generation. For simulation with code generation, select your **Device vendor** and **Device type** if they are available in the drop-down list. If your **Device vendor** and **Device type** are not available, set device-specific values by using `Custom Processor`.	integer `Char`, floating-point `Float`

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Byte ordering"	No impact	No impact	No impact	No recommendation for simulation without code generation. For simulation with code generation, select your **Device vendor** and **Device type** if they are available in the drop-down list. If your **Device vendor** and **Device type** are not available, set device-specific values by using `Custom Processor`.	`Little Endian`
"Signed integer division rounds to"	No impact for simulation and during development `Undefined` for production code generation	No impact for simulation and during development `Zero` or `Floor` for production code generation	No impact for simulation and during development `Zero` for production code generation	No recommendation for simulation without code generation. For simulation with code generation, select your **Device vendor** and **Device type** if they are available in the drop-down list. If your **Device vendor** and **Device type** are not available, set device-specific values by using `Custom Processor`.	`Zero`

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
“Shift right on a signed integer as arithmetic shift”	No impact	No impact	On	No recommendation for simulation without code generation. For simulation with code generation, select your **Device vendor** and **Device type** if they are available in the drop-down list. If your **Device vendor** and **Device type** are not available, set device-specific values by using `Custom Processor`.	On
“Support long long”	No impact	No impact	On (execution, ROM)	No recommendation for simulation without code generation. For simulation with code generation, select your **Device vendor** and **Device type** if they are available in the drop-down list. If your **Device vendor** and **Device type** are not available, set device-specific values by using `Custom Processor`.	Off

Mapping Application Requirements to the Model Referencing Pane

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Rebuild"	No impact	No impact	No impact	`If any changes detected` or `Never` If you use the `Never` setting, then set the **Never rebuild diagnostic** parameter to `Error if rebuild required`	`If any changes detected`
"Never rebuild diagnostic"	No impact	No impact	No impact	`error if rebuild required`	`error if rebuild required`
"Enable parallel model reference builds"	No impact	No impact	No impact	No impact	Off
"MATLAB worker initialization for builds"	No impact	No impact	No impact	No impact	`None`
"Total number of instances allowed per top model"	No impact	No impact	No impact	No recommendation	`Multiple`
"Pass fixed-size scalar root inputs by value for code generation"	No impact	No impact	No impact	No recommendation	Off
"Minimize algebraic loop occurrences"	No impact	No impact	No impact	No recommendation	Off
"Propagate sizes of variable-size signals"	No impact	No impact	No impact	No recommendation	`Infer from blocks in model`
"Model dependencies"	No impact	No impact	No impact	No recommendation	`''`

Mapping Application Requirements to the Simulation Target Pane: General Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Enable memory integrity checks"	On	No impact	No recommendation	On	On
"Echo expressions without semicolons"	On	No impact	Off	No impact	On
"Break on Ctrl+C"	On	No recommendation	No recommendation	No recommendation	On

Mapping Application Requirements to the Simulation Target Pane: Symbols Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Reserved names"	No impact	No impact	No impact	No recommendation	`{}`

Mapping Application Requirements to the Simulation Target Pane: Custom Code Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
“Import custom code”	On	No impact	No impact	On	On
“Source file”	No recommendation	No recommendation	No recommendation	No recommendation	' '
“Header file”	No recommendation	No recommendation	No recommendation	No recommendation	' '
“Initialize function”	No recommendation	No recommendation	No recommendation	No recommendation	' '
“Terminate function”	No recommendation	No recommendation	No recommendation	No recommendation	' '
“Include directories”	No impact	No impact	No impact	No recommendation	' '
“Source files”	No impact	No impact	No impact	No recommendation	' '
“Libraries”	No impact	No impact	No impact	No recommendation	' '
“Defines”	No impact	No impact	No impact	No recommendation	' '

Mapping Application Requirements to the Code Generation Pane: General Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
System target file on page 7-7	No impact	No impact	No impact	No impact (GRT) ERT based (ERT)	`grt.tlc`
“Language” on page 7-10	No impact	No impact	No impact	No impact	C
compexcep="155"“ Compiler optimization level” on page 7-19	`Optimizations off (faster builds)`	`Optimizations off (faster builds)`	`Optimizations on (faster runs)` (execution) No impact (ROM, RAM)	No impact	`Optimizations off (faster builds)`
“Custom compiler optimization flags” on page 7-21	`Optimizations off (faster builds)`	`Optimizations off (faster builds)`	`Optimizations on (faster runs)`	No impact	`Optimizations off (faster builds)`
“Generate makefile” on page 7-22	No impact	No impact	No impact	No impact	On
“Make command” on page 7-24	No impact	No impact	No impact	No recommendation	`make_rtw`
“Template makefile” on page 7-26	No impact	No impact	No impact	No impact	`grt_default_tmf`
“Select objective” on page 7-28	`Debugging`	Not applicable for GRT-based targets	`Execution efficiency`	No recommendation	`Unspecified`
“Check model before generating code” on page 7-34	`On (proceed with warnings)` or `On (stop for warnings)`	`On (proceed with warnings)` or `On (stop for warnings)`	`On (proceed with warnings)` or `On (stop for warnings)`	`On (proceed with warnings)` or `On (stop for warnings)`	`Off`
“Generate code only” on page 7-36	Off	No impact	No impact	No impact	Off
“Verbose build” on page 7-46	On	No impact	No impact	No recommendation	On
“Retain .rtw file” on page 7-47	On	No impact	No impact	No impact	Off
“Profile TLC” on page 7-48	On	No impact	No impact	No impact	Off
“Start TLC debugger when generating code” on page 7-49	On	No impact	No impact	No impact	Off

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Start TLC coverage when generating code" on page 7-50	On	No impact	No impact	No impact	Off
"Enable TLC assertion" on page 7-51	On	No impact	No impact	No recommendation	Off

Mapping Application Requirements to the Code Generation Pane: Report Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Create code generation report" on page 8-5	On	On	No impact	No recommendation	Off
"Open report automatically" on page 8-7	On	On	No impact	No impact	Off

Mapping Application Requirements to the Code Generation Pane: Comments Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Include comments" on page 9-6	On	On	No impact	No recommendation	On
"Simulink block comments" on page 9-8	On	On	No impact	No recommendation	On
"Stateflow object comments" on page 9-9	On	On	No impact	No recommendation	Off
"Show eliminated blocks" on page 9-13	On	On	No impact	No recommendation	On
"Verbose comments for 'Model default' storage class" on page 9-14	On	On	No impact	No recommendation	On
"Operator annotations" (Embedded Coder)	No impact	On	No impact	No recommendation	On

Mapping Application Requirements to the Code Generation Pane: Identifiers Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Maximum identifier length" on page 10-5	Valid value	>30	No impact	>30	31
"Use the same reserved names as Simulation Target" on page 10-7	No impact	No impact	No impact	No impact	Off
"Reserved names" on page 10-8	No impact	No impact	No impact	No impact	{}

Mapping Application Requirements to the Code Generation Pane: Custom Code Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Use the same custom code settings as Simulation Target" on page 11-4	No impact	No impact	No impact	No impact	Off
"Source file" on page 11-5	No impact	No impact	No impact	No impact	' '
"Header file" on page 11-6	No impact	No impact	No impact	No impact	' '
"Initialize function" on page 11-7	No impact	No impact	No impact	No impact	' '
"Terminate function" on page 11-8	No impact	No impact	No impact	No impact	' '
"Include directories" on page 11-9	No impact	No impact	No impact	No impact	' '
"Source files" on page 11-11	No impact	No impact	No impact	No impact	' '
"Libraries" on page 11-12	No impact	No impact	No impact	No impact	' '
"Defines" on page 11-13	No impact	No impact	No impact	No impact	' '

Mapping Application Requirements to the Code Generation Pane: Interface Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Standard math library" on page 12-48	No impact	No impact	Valid library	No impact	`C99 (ISO)`
"Code replacement library" on page 12-9	No impact	No impact	Valid library	No impact	`None`
"Shared code placement" on page 12-13	`Shared location` (GRT) No impact (ERT)	`Shared location` (GRT) No impact (ERT)	No impact (execution, RAM) `Shared location` (ROM)	No impact	`Auto`
"Support: non-finite numbers" on page 12-15	No impact	No impact	Off (Execution, ROM) No impact (RAM)	Norecommendation	On
"Code interface packaging" on page 12-17	No impact	No impact	`Reusable function` or `C++ class`	No impact	`Nonreusable function` if **Language** is set to `C`; `C++ class` if **Language** is set to `C++`
"Multi-instance code error diagnostic" on page 12-20	`Warning` or `Error`	No impact	`None`	No impact	`Error`
"Classic call interface" on page 12-52	No impact	Off	Off (execution, ROM), No impact (RAM)	No recommendation	Off (except On for GRT models created before R2012a)
"Single output/ update function" on page 12-54	On	On	On	No recommendation	On
"MAT-file logging" on page 12-56	On	No impact	Off	Off	On (GRT) Off (ERT)
"MAT-file variable name modifier" (Embedded Coder)	No impact	No impact	No impact	No impact	`rt_`

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Generate C API for: signals" on page 12-22	No impact	No impact	No impact	No impact (development) Off (production)	Off
"Generate C API for: parameters" on page 12-23	No impact	No impact	No impact	No impact (development) Off (production)	Off
"Generate C API for: states" on page 12-24	No impact	No impact	No impact	No impact (development) Off (production)	Off
"Generate C API for: root-level I/O" on page 12-25	No impact	No impact	No impact	No impact (development) Off (production)	Off
"ASAP2 interface" on page 12-26	No impact	No impact	No impact	No impact (development) Off (production)	Off
"External mode" on page 12-27	No impact	No impact	No impact	No impact (development) Off (production)	Off
"Transport layer" on page 12-28	No impact	No impact	No impact	No impact	`tcpip`
"MEX-file arguments" on page 12-30	No impact	No impact	No impact	No impact	`''`
"Static memory allocation" on page 12-32	No impact	No impact	No impact	No impact	Off
"Static memory buffer size" on page 15-20	No impact	No impact	No impact	No impact	1000000

Model Advisor Checks

- “Simulink Coder Checks” on page 16-2
- “Code Generation Advisor Checks” on page 16-21

Simulink Coder Checks

In this section...
“Simulink Coder Checks Overview” on page 16-2
“Check reuse of subsystem code” on page 16-2
“Identify blocks using one-based indexing” on page 16-3
“Check solver for code generation” on page 16-3
“Check for blocks not supported by code generation” on page 16-4
“Check and update model to use toolchain approach to build generated code” on page 16-5
“Check and update embedded target model to use ert.tlc system target file” on page 16-6
“Check and update models that are using targets that have changed significantly across different releases of MATLAB” on page 16-8
“Check for blocks that have constraints on tunable parameters” on page 16-8
“Check for model reference configuration mismatch” on page 16-9
“Check sample times and tasking mode” on page 16-10
“Check for code generation identifier formats used for model reference” on page 16-11
“Check for relative execution order change for Data Store Read and Data Store Write blocks” on page 16-11
“Available Checks for Code Generation Objectives” on page 16-12
“Identify questionable blocks within the specified system” on page 16-18
“Check model configuration settings against code generation objectives” on page 16-19

Simulink Coder Checks Overview

Use Simulink Coder Model Advisor checks to configure your model for code generation.

See Also

- “Run Model Advisor Checks”
- “Simulink Checks”
- “Embedded Coder Checks” (Embedded Coder)

Check reuse of subsystem code

Check ID: `mathworks.codegen.SubsysCodeReuse`

Identify CodeReuseSubsystem blocks that are not reusing code.

Results and Recommended Actions

Condition	Recommended Action
One or more CodeReuseSubsystem blocks in the model do not reuse code.	Modify CodeReuseSubsystem blocks in the model so that they reuse code.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Look under masks,
- Exclude blocks and charts from this check if you have a Simulink Check™ license.

See Also

- “Model Advisor Exclusion Overview” (Simulink Check)

Identify blocks using one-based indexing

Check ID: `mathworks.codegen.cgsl_0101`

Identify blocks using one-based indexing.

Description

Zero-based indexing is more efficient in the generated code than one-based indexing.

Using zero-based indexing increases execution efficiency of the generated code.

Available with Simulink Coder.

Results and Recommended Actions

Condition	Recommended Action
The model or subsystem contains blocks configured for one-based indexing.	Configure the blocks for zero-based indexing. Update the supporting blocks.

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

- “cgsl_0101: Zero-based indexing”.
- “Model Advisor Exclusion Overview” (Simulink Check)

Check solver for code generation

Check ID: `mathworks.codegen.SolverCodeGen`

Check model solver and sample time configuration settings.

Description

Incorrect configuration settings can stop the code generator from producing code. Underspecifying sample times can lead to undesired results. Avoid generating code that might corrupt data or produce unpredictable behavior.

Available with Simulink Coder.

Results and Recommended Actions

Condition	Recommended Action
The solver type is set incorrectly for model level code generation.	In the Configuration Parameters dialog box, on the **Solver** pane, set **Type** to `Fixed-step`.
Multitasking diagnostic options are not set to `error`.	In the Configuration Parameters dialog box, on the **Diagnostics** pane, set • **Sample Time** > **Multitask conditionally executed subsystem** to `error` • **Sample Time** > **Multitask rate transition** to `error` • **Data Validity** > **Multitask data store** to `error`

Tips

You do not have to modify the solver settings to generate code from a subsystem. The build process automatically changes **Solver type** to `fixed-step` when you right-click on the subsystem and select **C/C++ Code** > **Build This Subsystem** or **C/C++ Code** > **Generate S-Function** from the subsystem context menu.

See Also

- “Configure Time-Based Scheduling”
- “Execute Multitasking Models”

Check for blocks not supported by code generation

Check ID: `mathworks.codegen.codeGenSupport`

Identify blocks not supported by code generation.

Description

This check partially identifies model constructs that are not suited for code generation as identified in the Simulink Block Support tables for Simulink Coder and Embedded Coder. If you are using blocks with support notes for code generation, review the information and follow the given advice.

Available with Simulink Coder.

Results and Recommended Actions

Condition	Recommended Action
The model or subsystem contains blocks that should not be used for code generation.	Consider replacing the blocks listed in the results. Click an element from the list of questionable items to locate condition.

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.

See Also

- “Blocks and Products Supported for Code Generation”
- “Model Advisor Exclusion Overview” (Simulink Check)

Check and update model to use toolchain approach to build generated code

Check ID: `mathworks.codegen.toolchainInfoUpgradeAdvisor.check`

Check if model uses Toolchain settings to build generated code.

Description

Checks whether the model uses the template makefile approach or the toolchain approach to build the generated code.

Available with Simulink Coder.

When you open a model created before R2013b that has **System target file** set to `ert.tlc`, `ert_shrlib.tlc`, or `grt.tlc` the software automatically tries to upgrade the model from using the template makefile approach to using the toolchain approach.

If the software did not upgrade the model, this check determines the cause, and if available, recommends actions you can perform to upgrade the model.

To determine which approach your model is using, you can also look at the Code Generation pane in the Configuration Parameters dialog box. The toolchain approach uses the following parameters to build generated code:

- “Toolchain” on page 7-14
- “Build configuration” on page 7-16

The template makefile approach uses the following settings to build generated code:

- **Compiler optimization level**

- **Custom compiler optimization flags**
- **Generate makefile**
- **Template makefile**

Results and Recommended Actions

Condition	Recommended Action	Comment
Model is configured to use the toolchain approach.	No action.	The model was automatically upgraded.
Model is not configured to use the toolchain approach.	Model cannot be automatically upgraded to use the toolchain approach.	The system target file is not toolchain-compliant. Set **System target file** to a toolchain-compliant target, such as `ert.tlc`, `ert_shrlib.tlc`, or `grt.tlc`.
Model is not configured to use the toolchain approach. (Parameter values are not the default values.)	Model can be automatically upgraded to use the toolchain approach. Click **Update Model**.	The parameters are set to their default values, except **Compiler Optimization Level**, which is set `Optimizations on (faster runs)`. Clicking **Update Model** sets **Compiler Optimization Level** to its default value, `Optimizations off (faster builds)`, and then upgrades the model. The upgraded model has **Build Configuration** set to `Faster Builds`. Saving the model makes these changes permanent.
Model is not configured to use the toolchain approach. (Parameter values are not the default values.)	Model cannot be automatically upgraded to use the toolchain approach.	One or more of the following parameters is not set to its default value: • **Generate makefile** (default: Enabled) • **Template makefile** (default: Target-specific default TMF) • **Compiler optimization level** (default: `Optimizations off (faster builds)`) • **Make command** (default: `make_rtw` without arguments) See “Upgrade Model to Use Toolchain Approach”

Action Results

Clicking **Update model** upgrades the model to use the toolchain approach to build generated code.

See Also

- “Upgrade Model to Use Toolchain Approach”

Check and update embedded target model to use ert.tlc system target file

Check ID: `mathworks.codegen.codertarget.check`

Check and update the embedded target model to use ert.tlc system target file.

Description

Check and update models whose **System target file** is set to a file other than `ert.tlc` and whose target hardware is one of the supported Texas Instruments C2000™ processors to use `ert.tlc` and similar settings.

Available with Simulink Coder.

Results and Recommended Actions

Condition	Recommended Action
System target file is set to `ert.tlc - Embedded Coder`.	No action
System target file is set to a file other than `ert.tlc` and **Hardware board** parameter is set to a processor that is supported by the Embedded Coder Support Package for Texas Instruments C2000 Processors.	Run the corresponding check in Upgrade Advisor: 1 In the **Modeling** tab of Simulink Editor, click **Model Advisor**. 2 Open the **Upgrade Advisor** pane and select check **Check and update embedded target model to use ert.tlc system target file**. 3 Right-click on the check and select **Run This Check**. 4 After the check passes, open the Configuration Parameters dialog box, go to **Hardware Implementation** pane and confirm that the correct **Hardware board** is selected.

Action Results

Clicking **Run This Check** automatically sets the following parameters on the **Code Generation** pane in the model Configuration Parameters dialog box:

- **System target file** parameter to `ert.tlc`.
- **Toolchain** parameter to match the previous toolchain.
- **Build configuration** parameter to match the build configuration.

Capabilities and Limitations

The new workflow uses the toolchain approach, which relies on enhanced makefiles to build generated code. It does not provide an equivalent to setting the **Build format** parameter to `Project` in the previous configuration. Therefore, the new workflow cannot automatically generate IDE projects within the CCS 3.3 IDE.

See Also

“Toolchain Configuration”

Check and update models that are using targets that have changed significantly across different releases of MATLAB

Check ID: `mathworks.codegen.realtime2CoderTargetInfoUpgradeAdvisor.check`

Check and update models with Simulink targets that have changed significantly across different releases of MATLAB.

Description

Save a model that you have updated to work with the current installation of MATLAB.

Available with Simulink Coder.

Results and Recommended Actions

Condition	Recommended Action
Model uses a target that has changed significantly since the release of MATLAB in which it was originally saved.	Save model
Model does not use a Simulink target or is using the latest version of the target.	No action
Model is automatically updated.	Save model
Invalid external mode configuration.	In the **Configuration Parameters** > **Interface** pane, update the external mode parameter settings to match characteristics of your host-target connection.
Model is corrupted.	Close and reopen the model. If the issue persists, reset **Configuration Parameters > Hardware Implementation > Hardware board**.

Action Results

Clicking **Save model** updates the model to work with the current installation of MATLAB and saves the model.

See Also

“Configure Production and Test Hardware”

Check for blocks that have constraints on tunable parameters

Check ID: `mathworks.codegen.ConstraintsTunableParam`

Identify blocks with constraints on tunable parameters.

Description

Lookup Table blocks have strict constraints when they are tunable. If you violate lookup table block restrictions, the generated code produces incorrect answers.

Available with Simulink Coder.

Results and Recommended Actions

Condition	Recommended Action
Lookup Table blocks have tunable parameters.	When tuning parameters during simulation or when running the generated code, you must: • Preserve monotonicity of the setting for the **Vector of input values** parameter. • Preserve the number and location of zero values that you specify for **Vector of input values** and **Vector of output values** parameters if you specify multiple zero values for the **Vector of input values** parameter.
Lookup Table (2-D) blocks have tunable parameters.	When tuning parameters during simulation or when running the generated code, you must: • Preserve monotonicity of the setting for the **Row index input values** and **Column index of input values** parameters. • Preserve the number and location of zero values that you specify for **Row index input values**, **Column index of input values**, and **Vector of output values** parameters if you specify multiple zero values for the **Row index input values** or **Column index of input values** parameters.
Lookup Table (n-D) blocks have tunable parameters.	When tuning parameters during simulation or when running the generated code, you must preserve the increasing monotonicity of the breakpoint values for each table dimension **Breakpoints n**.

Capabilities and Limitations

If you have a Simulink Check license, you can exclude blocks and charts from this check.

See Also

- 1-D Lookup Table
- 2-D Lookup Table
- “Model Advisor Exclusion Overview” (Simulink Check)

Check for model reference configuration mismatch

Check ID: `mathworks.codegen.MdlrefConfigMismatch`

Identify referenced model configuration parameter settings that do not match the top model configuration parameter settings.

Description

The code generator cannot create code for top models that contain referenced models with different, incompatible configuration parameter settings.

Available with Simulink Coder.

Results and Recommended Actions

Condition	Recommended Action
The top model and the referenced model have inconsistent model configuration parameter settings.	Modify the specified model configuration settings.

See Also

- “Model Reference Basics”
- “Set Configuration Parameters for Model Hierarchies”

Check sample times and tasking mode

Check ID: `mathworks.codegen.SampleTimesTaskingMode`

Set up the sample time and tasking mode for your system.

Description

Incorrect tasking mode can result in inefficient code execution or incorrect generated code.

Available with Simulink Coder.

Results and Recommended Actions

Condition	Recommended Action
The model represents a multirate system but is not configured for multitasking.	Select model configuration parameter **“Treat each discrete rate as a separate task”**. When you select the parameter, multitasking execution is applied for a multirate model. For more information, see “Time-Based Scheduling and Code Generation”.
The model is configured for multitasking, but multitasking is not desirable on the target hardware. For example, the operating system does not support multiprocessing or the target hardware is bare metal (is not running an operating system) and the application does not provide for a multitasking execution scheme.	Clear model configuration parameter **“Treat each discrete rate as a separate task”**. When you clear the parameter, single-tasking execution is applied. For more information, see “Time-Based Scheduling and Code Generation”.

See Also

“Time-Based Scheduling and Code Generation”

Check for code generation identifier formats used for model reference

Check ID: `mathworks.codegen.ModelRefRTWConfigCompliance`

Checks for referenced models in a model referencing hierarchy for which code generation changes configuration parameter settings that involve identifier formats.

Description

In referenced models, if the following **Configuration Parameters > Code Generation > Identifiers** parameters have settings that do not contain a `$R` token (which represents the name of the reference model), code generation prepends the `$R` token to the identifier format.

- **Global variables**
- **Global types**
- **Subsystem methods**
- **Constant macros**

Available with Simulink Coder.

Results and Recommended Actions

Condition	Recommended Action
A script that operates on generated code uses model names that code generation changes.	Update the script to use the generated name (which includes an appended `$R` token).

Check for relative execution order change for Data Store Read and Data Store Write blocks

Check ID: `com.mathworks.sorting.datastoresimrtwcmp`

Checks that the execution order of Data Store Read and Data Store Write blocks does not change when a model is compiled for code generation.

Description

The execution order defines the sequence in which the Data Store Read and Data Store Write blocks access the Data Store Memory block. The Model Advisor check compares the execution order from prior to running the check (normal simulation mode) to the execution order after compiling the check (code generation mode). The check passes when the execution order is the same. When there are differences, the check issues a **Warning** and identifies the discrepancies in the results.

Available with Simulink Coder.

Results and Recommended Actions

Condition	Recommended Action
The execution order of the Data Store Read and Data Store Write blocks is different between simulation mode and code generation mode.	In the Model Advisor results, under **Action**, click **Modify block priorities**. The Model Advisor modifies the blocks so the execution order in simulation mode is the same as that in code generation mode.

See Also

- “Control and Display Execution Order”
- “Data Store Basics”
- “Order Data Store Access”

Available Checks for Code Generation Objectives

Code generation objectives checks facilitate designing and troubleshooting Simulink models and subsystems that you want to use to generate code.

The Code Generation Advisor includes the following checks from Simulink, Simulink Coder, and Embedded Coder for each of the code generation objectives. Two checks unique to the Code Generation Advisor are included below the list.

Check	Execution efficiency (all targets)	ROM efficiency (ERT-based targets)	RAM efficiency (ERT-based targets)	Safety precaution (ERT-based targets)	Traceability (ERT-based targets)	Debugging (all targets)	MISRA C:2012 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Check model configuration settings against code generation objectives” on page 16-19	Included	Included	Included	Included	Included	Included	Included (see **Note** below)	Included
“Check for optimal bus virtuality”	Included	Included	Included	N/A	N/A	N/A	N/A	N/A

Check	Execution efficiency (all targets)	ROM efficiency (ERT-based targets)	RAM efficiency (ERT-based targets)	Safety precaution (ERT-based targets)	Traceability (ERT-based targets)	Debugging (all targets)	MISRA C:2012 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Identify questionable blocks within the specified system” on page 16-18	Included	Included	Included	N/A	N/A	N/A	N/A	N/A
“Check the hardware implementation” (Embedded Coder)	Included if Embedded Coder is available	Included if Embedded Coder is available	N/A	N/A	N/A	N/A	N/A	N/A
“Identify questionable software environment specifications” (Embedded Coder)	Included when Traceability is not a higher priority and Embedded Coder is available	Included when Traceability is not a higher priority and Embedded Coder is available	N/A	N/A	N/A	N/A	N/A	N/A
“Identify questionable code instrumentation (data I/O)” (Embedded Coder)	Included when Traceability or Debugging are not higher priorities and Embedded Coder is available	Included when Traceability or Debugging are not higher priorities and Embedded Coder is available	Included when Traceability or Debugging are not higher priorities and Embedded Coder is available	N/A	N/A	N/A	N/A	N/A
“Identify questionable subsystem settings” (Embedded Coder)	N/A	Included if Embedded Coder is available	Included if Embedded Coder is available	N/A	N/A	N/A	N/A	N/A

Check	Execution efficiency (all targets)	ROM efficiency (ERT-based targets)	RAM efficiency (ERT-based targets)	Safety precaution (ERT-based targets)	Traceability (ERT-based targets)	Debugging (all targets)	MISRA C:2012 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Identify blocks that generate expensive rounding code” (Embedded Coder)	Included if Embedded Coder is available	Included if Embedded Coder is available	N/A	N/A	N/A	N/A	N/A	N/A
“Identify questionable fixed-point operations” (Embedded Coder)	Included if Embedded Coder or Fixed-Point Designer™ is available	Included if Embedded Coder or Fixed-Point Designer is available	N/A	N/A	N/A	N/A	N/A	N/A
“Identify blocks using one-based indexing” on page 16-3	Included	Included	N/A	N/A	N/A	N/A	N/A	N/A
“Identify lookup table blocks that generate expensive out-of-range checking code” (Embedded Coder)	Included if Embedded Coder is available	Included if Embedded Coder is available	N/A	N/A	N/A	N/A	N/A	N/A
“Check output types of logic blocks” (Embedded Coder)	Included if Embedded Coder is available	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Check	Execution efficiency (all targets)	ROM efficiency (ERT-based targets)	RAM efficiency (ERT-based targets)	Safety precaution (ERT-based targets)	Traceability (ERT-based targets)	Debugging (all targets)	MISRA C:2012 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Identify unconnected lines, input ports, and output ports”	N/A	N/A	N/A	Included	N/A	N/A	N/A	N/A
“Check Data Store Memory blocks for multitasking, strong typing, and shadowing issues”	N/A	N/A	N/A	Included	N/A	N/A	N/A	N/A
“Identify block output signals with continuous sample time and non-floating point data type”	N/A	N/A	N/A	Included	N/A	N/A	N/A	N/A
“Check for blocks that have constraints on tunable parameters ” on page 16-8	N/A	N/A	N/A	Included	N/A	N/A	N/A	N/A
“Check if read/write diagnostics are enabled for data store blocks”	N/A	N/A	N/A	Included	N/A	N/A	N/A	N/A

Check	Execution efficiency (all targets)	ROM efficiency (ERT-based targets)	RAM efficiency (ERT-based targets)	Safety precaution (ERT-based targets)	Traceability (ERT-based targets)	Debugging (all targets)	MISRA C:2012 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Check structure parameter usage with bus signals”	N/A	N/A	N/A	Included	N/A	N/A	N/A	N/A
“Check data store block sample times for modeling errors”	N/A	N/A	N/A	Included	N/A	N/A	N/A	N/A
“Check for potential ordering issues involving data store access”	N/A	N/A	N/A	Included	N/A	N/A	N/A	N/A
“Check for blocks not recommended for C/C++ production code deployment” (Embedded Coder)	N/A	N/A	N/A	N/A	N/A	N/A	Included if Embedded Coder is available	N/A
“Check for blocks not recommended for MISRA C:2012” (Embedded Coder)	N/A	N/A	N/A	N/A	N/A	N/A	Included if Embedded Coder is available	N/A
“Check for unsupported block names” (Embedded Coder)	N/A	N/A	N/A	N/A	N/A	N/A	Included if Embedded Coder is available	N/A

Check	Execution efficiency (all targets)	ROM efficiency (ERT-based targets)	RAM efficiency (ERT-based targets)	Safety precaution (ERT-based targets)	Traceability (ERT-based targets)	Debugging (all targets)	MISRA C:2012 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Check usage of Assignment blocks” (Embedded Coder)	N/A	N/A	N/A	N/A	N/A	N/A	Included if Embedded Coder is available	N/A
“Check for bitwise operations on signed integers” (Embedded Coder)	N/A	N/A	N/A	N/A	N/A	N/A	Included if Embedded Coder is available	N/A
“Check for recursive function calls” (Embedded Coder)	N/A	N/A	N/A	N/A	N/A	N/A	Included if Embedded Coder is available	N/A
“Check for equality and inequality operations on floating-point values” (Embedded Coder)	N/A	N/A	N/A	N/A	N/A	N/A	Included if Embedded Coder is available	N/A
“Check for switch case expressions without a default case” (Embedded Coder)	N/A	N/A	N/A	N/A	N/A	N/A	Included if Embedded Coder is available	N/A

Check	Execution efficiency (all targets)	ROM efficiency (ERT-based targets)	RAM efficiency (ERT-based targets)	Safety precaution (ERT-based targets)	Traceability (ERT-based targets)	Debugging (all targets)	MISRA C:2012 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Check configuration parameters for generation of inefficient saturation code” (Embedded Coder)	Included if Embedded Coder is available	Included if Embedded Coder is available	N/A	Included if Embedded Coder is available	N/A	N/A	N/A	N/A

Note When the Code Generation Advisor checks your model against the MISRA C:2012 guidelines objective, the tool does not consider all of the configuration parameter settings that are checked by the MISRA C:2012 guidelines checks in the Model Advisor. For a complete check of configuration parameter settings:

1. Open the Model Advisor.
2. Navigate to **By Task > Modeling Guidelines for MISRA C:2012**.
3. Run the checks in the folder.

For more information on using the Model Advisor, see “Check Your Model Using the Model Advisor”.

See Also

- “Application Objectives Using Code Generation Advisor”
- “Configure Model for Code Generation Objectives by Using Code Generation Advisor” (Embedded Coder)
- “Run Model Advisor Checks”
- “Simulink Checks”
- “Simulink Coder Checks” on page 16-2
- “Simulink Check Checks” (Simulink Check)

Identify questionable blocks within the specified system

Identify blocks not supported by code generation or not recommended for deployment.

Description

The code generator creates code only for the blocks that it supports. Some blocks are not recommended for production code deployment.

Results and Recommended Actions

Condition	Recommended Action
A block is not supported by the code generator.	Remove the specified block from the model or replace the block with the recommended block.
A block is not recommended for production code deployment.	Remove the specified block from the model or replace the block with the recommended block.
Check for Gain blocks whose value equals 1.	Replace Gain blocks with Signal Conversion blocks.

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

“Blocks and Products Supported for Code Generation”

“Model Advisor Exclusion Overview” (Simulink Check)

Check model configuration settings against code generation objectives

Check the configuration parameter settings for the model against the code generation objectives.

Description

Each parameter in the Configuration Parameters dialog box might have different recommended settings for code generation based on your objectives. This check helps you identify the recommended setting for each parameter so that you can achieve optimized code based on your objective.

Results and Recommended Actions

Condition	Recommended Action
Parameters are set to values other than the value recommended for the specified objectives.	Set the parameters to the recommended values. **Note** A change to one parameter value can impact other parameters. Passing the check might take multiple iterations.

Action Results

Clicking **Modify Parameters** changes the parameter values to the recommended values.

See Also

- “Recommended Settings Summary for Model Configuration Parameters” (Embedded Coder)
- “Application Objectives Using Code Generation Advisor”
- “Configure Model for Code Generation Objectives by Using Code Generation Advisor” (Embedded Coder)

Code Generation Advisor Checks

In this section...
“Available Checks for Code Generation Objectives” on page 16-21
“Identify questionable blocks within the specified system” on page 16-27
“Check model configuration settings against code generation objectives” on page 16-28

Available Checks for Code Generation Objectives

Code generation objectives checks facilitate designing and troubleshooting Simulink models and subsystems that you want to use to generate code.

The Code Generation Advisor includes the following checks from Simulink, Simulink Coder, and Embedded Coder for each of the code generation objectives. Two checks unique to the Code Generation Advisor are included below the list.

Check	Execution efficiency (all targets)	ROM efficiency (ERT-based targets)	RAM efficiency (ERT-based targets)	Safety precaution (ERT-based targets)	Traceability (ERT-based targets)	Debugging (all targets)	MISRA C:2012 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Check model configuration settings against code generation objectives” on page 16-28	Included	Included	Included	Included	Included	Included	Included (see **Note** below)	Included
“Check for optimal bus virtuality”	Included	Included	Included	N/A	N/A	N/A	N/A	N/A
“Identify questionable blocks within the specified system” on page 16-27	Included	Included	Included	N/A	N/A	N/A	N/A	N/A

Check	Execution efficiency (all targets)	ROM efficiency (ERT-based targets)	RAM efficiency (ERT-based targets)	Safety precaution (ERT-based targets)	Traceability (ERT-based targets)	Debugging (all targets)	MISRA C:2012 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Check the hardware implementation” (Embedded Coder)	Included if Embedded Coder is available	Included if Embedded Coder is available	N/A	N/A	N/A	N/A	N/A	N/A
“Identify questionable software environment specifications” (Embedded Coder)	Included when Traceability is not a higher priority and Embedded Coder is available	Included when Traceability is not a higher priority and Embedded Coder is available	N/A	N/A	N/A	N/A	N/A	N/A
“Identify questionable code instrumentation (data I/O)” (Embedded Coder)	Included when Traceability or Debugging are not higher priorities and Embedded Coder is available	Included when Traceability or Debugging are not higher priorities and Embedded Coder is available	Included when Traceability or Debugging are not higher priorities and Embedded Coder is available	N/A	N/A	N/A	N/A	N/A
“Identify questionable subsystem settings” (Embedded Coder)	N/A	Included if Embedded Coder is available	Included if Embedded Coder is available	N/A	N/A	N/A	N/A	N/A
“Identify blocks that generate expensive rounding code” (Embedded Coder)	Included if Embedded Coder is available	Included if Embedded Coder is available	N/A	N/A	N/A	N/A	N/A	N/A

Check	Execution efficiency (all targets)	ROM efficiency (ERT-based targets)	RAM efficiency (ERT-based targets)	Safety precaution (ERT-based targets)	Traceability (ERT-based targets)	Debugging (all targets)	MISRA C:2012 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Identify questionable fixed-point operations” (Embedded Coder)	Included if Embedded Coder or Fixed-Point Designer is available	Included if Embedded Coder or Fixed-Point Designer is available	N/A	N/A	N/A	N/A	N/A	N/A
“Identify blocks using one-based indexing” on page 16-3	Included	Included	N/A	N/A	N/A	N/A	N/A	N/A
“Identify lookup table blocks that generate expensive out-of-range checking code” (Embedded Coder)	Included if Embedded Coder is available	Included if Embedded Coder is available	N/A	N/A	N/A	N/A	N/A	N/A
“Check output types of logic blocks” (Embedded Coder)	Included if Embedded Coder is available	N/A	N/A	N/A	N/A	N/A	N/A	N/A
“Identify unconnected lines, input ports, and output ports”	N/A	N/A	N/A	Included	N/A	N/A	N/A	N/A

Check	Execution efficiency (all targets)	ROM efficiency (ERT-based targets)	RAM efficiency (ERT-based targets)	Safety precaution (ERT-based targets)	Traceability (ERT-based targets)	Debugging (all targets)	MISRA C:2012 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Check Data Store Memory blocks for multitasking, strong typing, and shadowing issues”	N/A	N/A	N/A	Included	N/A	N/A	N/A	N/A
“Identify block output signals with continuous sample time and non-floating point data type”	N/A	N/A	N/A	Included	N/A	N/A	N/A	N/A
“Check for blocks that have constraints on tunable parameters ” on page 16-8	N/A	N/A	N/A	Included	N/A	N/A	N/A	N/A
“Check if read/write diagnostics are enabled for data store blocks”	N/A	N/A	N/A	Included	N/A	N/A	N/A	N/A
“Check structure parameter usage with bus signals”	N/A	N/A	N/A	Included	N/A	N/A	N/A	N/A

Check	Execution efficiency (all targets)	ROM efficiency (ERT-based targets)	RAM efficiency (ERT-based targets)	Safety precaution (ERT-based targets)	Traceability (ERT-based targets)	Debugging (all targets)	MISRA C:2012 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Check data store block sample times for modeling errors”	N/A	N/A	N/A	Included	N/A	N/A	N/A	N/A
“Check for potential ordering issues involving data store access”	N/A	N/A	N/A	Included	N/A	N/A	N/A	N/A
“Check for blocks not recommended for C/C++ production code deployment” (Embedded Coder)	N/A	N/A	N/A	N/A	N/A	N/A	Included if Embedded Coder is available	N/A
“Check for blocks not recommended for MISRA C:2012” (Embedded Coder)	N/A	N/A	N/A	N/A	N/A	N/A	Included if Embedded Coder is available	N/A
“Check for unsupported block names” (Embedded Coder)	N/A	N/A	N/A	N/A	N/A	N/A	Included if Embedded Coder is available	N/A

Check	Execution efficiency (all targets)	ROM efficiency (ERT-based targets)	RAM efficiency (ERT-based targets)	Safety precaution (ERT-based targets)	Traceability (ERT-based targets)	Debugging (all targets)	MISRA C:2012 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Check usage of Assignment blocks” (Embedded Coder)	N/A	N/A	N/A	N/A	N/A	N/A	Included if Embedded Coder is available	N/A
“Check for bitwise operations on signed integers” (Embedded Coder)	N/A	N/A	N/A	N/A	N/A	N/A	Included if Embedded Coder is available	N/A
“Check for recursive function calls” (Embedded Coder)	N/A	N/A	N/A	N/A	N/A	N/A	Included if Embedded Coder is available	N/A
“Check for equality and inequality operations on floating-point values” (Embedded Coder)	N/A	N/A	N/A	N/A	N/A	N/A	Included if Embedded Coder is available	N/A
“Check for switch case expressions without a default case” (Embedded Coder)	N/A	N/A	N/A	N/A	N/A	N/A	Included if Embedded Coder is available	N/A

Check	Execution efficiency (all targets)	ROM efficiency (ERT-based targets)	RAM efficiency (ERT-based targets)	Safety precaution (ERT-based targets)	Traceability (ERT-based targets)	Debugging (all targets)	MISRA C:2012 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Check configuration parameters for generation of inefficient saturation code” (Embedded Coder)	Included if Embedded Coder is available	Included if Embedded Coder is available	N/A	Included if Embedded Coder is available	N/A	N/A	N/A	N/A

Note When the Code Generation Advisor checks your model against the MISRA C:2012 guidelines objective, the tool does not consider all of the configuration parameter settings that are checked by the MISRA C:2012 guidelines checks in the Model Advisor. For a complete check of configuration parameter settings:

1. Open the Model Advisor.
2. Navigate to **By Task > Modeling Guidelines for MISRA C:2012**.
3. Run the checks in the folder.

For more information on using the Model Advisor, see “Check Your Model Using the Model Advisor”.

See Also

- “Application Objectives Using Code Generation Advisor”
- “Configure Model for Code Generation Objectives by Using Code Generation Advisor” (Embedded Coder)
- “Run Model Advisor Checks”
- “Simulink Checks”
- “Simulink Coder Checks” on page 16-2
- “Simulink Check Checks” (Simulink Check)

Identify questionable blocks within the specified system

Identify blocks not supported by code generation or not recommended for deployment.

Description

The code generator creates code only for the blocks that it supports. Some blocks are not recommended for production code deployment.

Results and Recommended Actions

Condition	Recommended Action
A block is not supported by the code generator.	Remove the specified block from the model or replace the block with the recommended block.
A block is not recommended for production code deployment.	Remove the specified block from the model or replace the block with the recommended block.
Check for Gain blocks whose value equals 1.	Replace Gain blocks with Signal Conversion blocks.

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

“Blocks and Products Supported for Code Generation”

“Model Advisor Exclusion Overview” (Simulink Check)

Check model configuration settings against code generation objectives

Check the configuration parameter settings for the model against the code generation objectives.

Description

Each parameter in the Configuration Parameters dialog box might have different recommended settings for code generation based on your objectives. This check helps you identify the recommended setting for each parameter so that you can achieve optimized code based on your objective.

Results and Recommended Actions

Condition	Recommended Action
Parameters are set to values other than the value recommended for the specified objectives.	Set the parameters to the recommended values. **Note** A change to one parameter value can impact other parameters. Passing the check might take multiple iterations.

Action Results

Clicking **Modify Parameters** changes the parameter values to the recommended values.

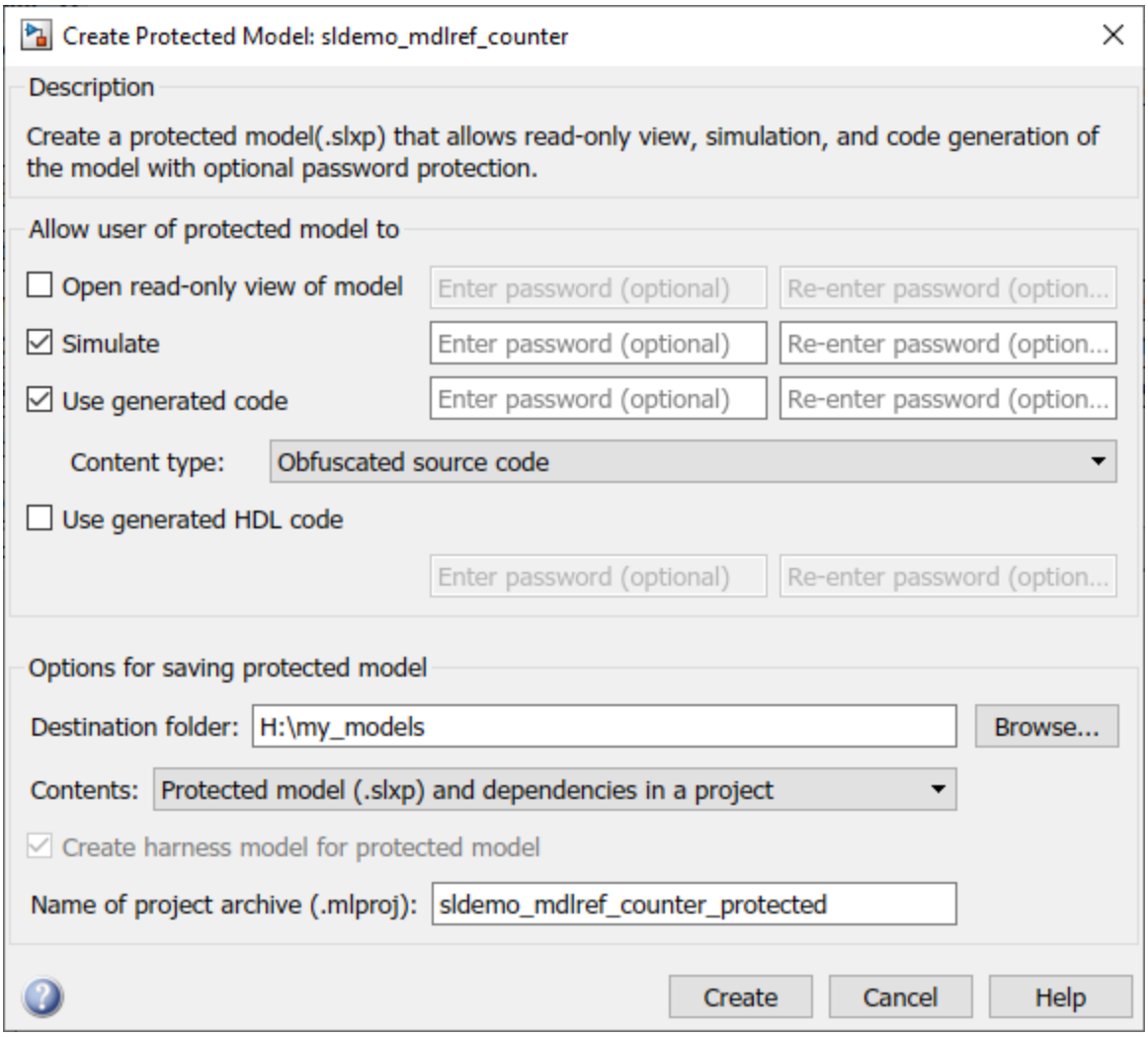
See Also

- “Recommended Settings Summary for Model Configuration Parameters” (Embedded Coder)
- “Application Objectives Using Code Generation Advisor”
- “Configure Model for Code Generation Objectives by Using Code Generation Advisor” (Embedded Coder)

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” (Embedded Coder)

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” (Embedded Coder)

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” (Embedded Coder)
- “Protect Models to Conceal Contents” (Embedded Coder)

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” (Embedded Coder)
- “Protect Models to Conceal Contents” (Embedded Coder)

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” (Embedded Coder)

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” (Embedded Coder)
- “Protect Models to Conceal Contents” (Embedded Coder)

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” (Embedded Coder)

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” (Embedded Coder)

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” (Embedded Coder)

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” (Embedded Coder)

Simulink Coder Tools

Code Mappings Editor – C

Associate model elements with code definitions

Description

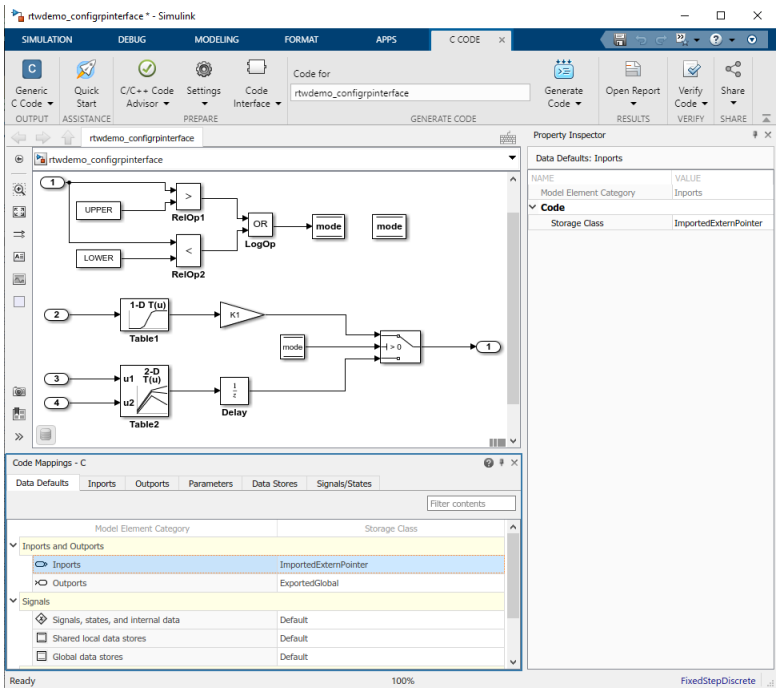
The Code Mappings editor is a graphical interface where you can configure data elements in a model, excluding referenced models, for code generation. Each model in a model reference hierarchy has its own code mappings. Associate each category of model data element with a specific storage class throughout a model. Then, override those settings, as needed, for specific data elements.

A storage class defines properties such as appearance and location, which the code generator uses when producing code for associated data.

To configure data elements and functions for code generation, use the tables in the Code Mappings editor display:

- **Data Defaults**
- **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 property settings for the selected data element.



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 Simulink Coder app. On the **C Code** tab, select **Code Interface > Default Code Mappings** or **Code Interface > Individual Element Code Mappings**.
- Open the Simulink 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-Level Inport and Outport Blocks

Configure code generation for the root-level 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 file `exDblFloat.h` into a writable folder.

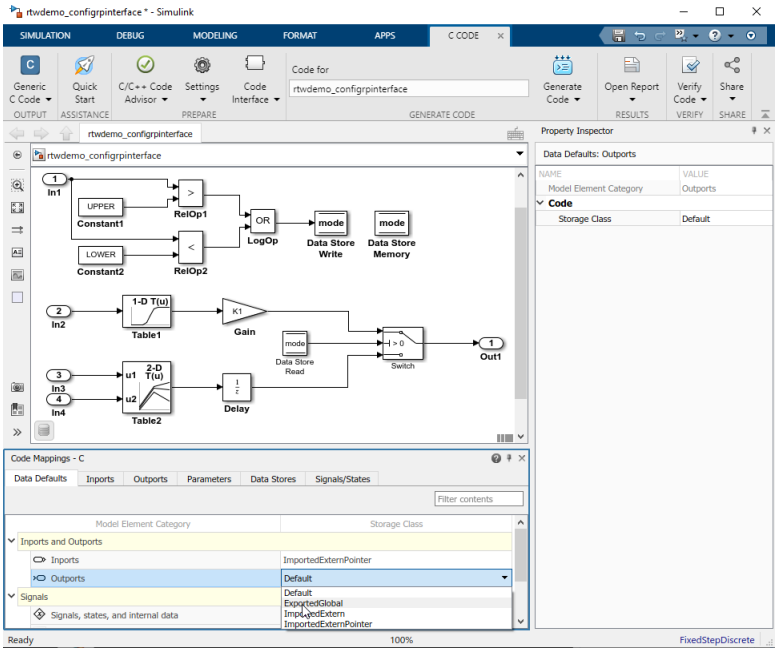
```
copyfile(fullfile(matlabroot,'toolbox','rtw','rtwdemos','exDblFloat.h'));
```

2. Open model `rtwdemo_configrpinterface`. Save a copy of the model into the same location as file `exDblFloat.h`.
3. Open the Simulink Coder app. The **C Code** tab includes the Code Mappings editor.

Configure Default Mappings

Configure the code generator to declare and define global variables for inports and outports in generated files `rtwdemo_configrpinterface.h` and `rtwdemo_configrpinterface.c`.

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 `ImportedExternPointer`. Set the storage class for **Outports** to `ExportedGlobal`. The editor updates the default storage class setting for the two selected data element categories.



Configure Individual Inports and Outports 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 model default configuration.
2. Force the code generator to use the default configuration for inports, which is storage class `ImportedExternPointer`. Press the **Ctrl** key and select the inports. For one of the selected inports, set the storage class to `Model default: ImportedExternPointer`. The editor updates the storage class setting for the selected inports.
3. Force the code generator to use storage class `ExportedGlobal` for the model root outport. Click the **Outports** tab. Select the row for `Out2`. Then, set the storage class to `Model default: ExportedGlobal`.

Configure Individual Data Elements

To configure properties for individual data elements, for example, if you need to override default configuration settings, use the tabs for the different data element types. For this example, override the default storage class setting for Inport block `In1`.

By default, the code generator names inport and outport variables based on the Inport or Outport block name in the model. When you configure data elements with a storage class setting other than `Auto`, you can override that default setting for individual elements by setting storage class property **Identifier**. This property enables you to specify an identifier for the code without modifying the model design. For this example, set **Identifier** for the Inport and Outport blocks.

1. In the Code Mappings editor, click the **Inports** tab.
2. For `In1`, set the storage class to `ImportedExtern`.
3. For each inport, select the row. Then, in the Property Inspector, set the **Identifier** property as follows:

- Set `In1` to `input1`.
- Set `In2` to `input2`.
- Set `In3` to `input3`.
- Set `In4` to `input4`.

4 Click **Outports**.

5 Select outport `Out1`. Then, in the Property Inspector, set the **Identifier** property to `output`.

Generate and Verify Code

Generate code and verify that the code generated for the Inport and Output blocks appears as you expect. For example:

- `rtwdemo_configrpinterface_private.h` includes these declarations:

```
/* Exported data declaration */

/* Data with Imported storage */
extern real_T input1;               /* '<Root>/In1' */
extern real_T input2;               /* '<Root>/In2' */
extern real_T input3;               /* '<Root>/In3' */
extern real_T input4;               /* '<Root>/In4' */
```

- `rtwdemo_configrpinterface.h` lists `output` as field in output structure `ExtY_rtwdemo_configrpinterfac_T`.

```
/* External outputs (root outports fed by signals with default storage) */
typedef struct {
  real_T output;                   /* '<Root>/Out1' */
} ExtY_rtwdemo_configrpinterfac_T;
```

- This code fragment shows the variable that represents `In2`, `input2`, being used in the generated entry-point step function for sample rate of 1 second.

```
/* Model step function for TID1 */
void rtwdemo_configrpinterface_step1(void) /* Sample time: [1.0s, 0.0s] */
{
   /* Lookup_n-D: '<Root>/Table1D' incorporates:
    * Inport: '<Root>/In2'
    */
   rtwdemo_configrpinterface_B.Table1D = look1_binlcpw(input2,
   rtwdemo_configrpinterfac_ConstP.Table1D_bp01Data,
   rtwdemo_configrpinterfac_ConstP.Table1D_tableData, 10U);
.
.
.
```

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 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.

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` (requires Embedded Coder) .
- Storage class defined in an Embedded Coder Dictionary (requires Embedded Coder) .

`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. Valid settings are `Default`, `ExportedGlobal`, `ImportedExtern`, and `ImportedExternPointer`. See “Choose Storage Class for Controlling Data Representation in Generated Code”.

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. Valid settings are `Auto`, `Model default`, `ExportedGlobal`, `ImportedExtern`, and `ImportedExternPointer`. 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. Valid settings are `Auto`, `Model default`, `ExportedGlobal`, `ImportedExtern`, and `ImportedExternPointer`. 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	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. Valid settings are `Auto`, `Model default`, `ExportedGlobal`, `ImportedExtern`, and `ImportedExternPointer`. 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.

Type of Data Store Element	Description
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. Valid settings are `Auto`, `Model default`, `ExportedGlobal`, `ImportedExtern`, and `ImportedExternPointer`. 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. Valid settings are `Auto`, `Model default`, `ExportedGlobal`, `ImportedExtern`, and `ImportedExternPointer`. 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

Topics
“C Code Generation Configuration for Model Interface Elements”
“Choose Data Configuration Approach”
“Choose Storage Class for Controlling Data Representation in Generated Code”
“Configure C Code Generation for Model Entry-Point Functions”

Introduced in R2020b

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

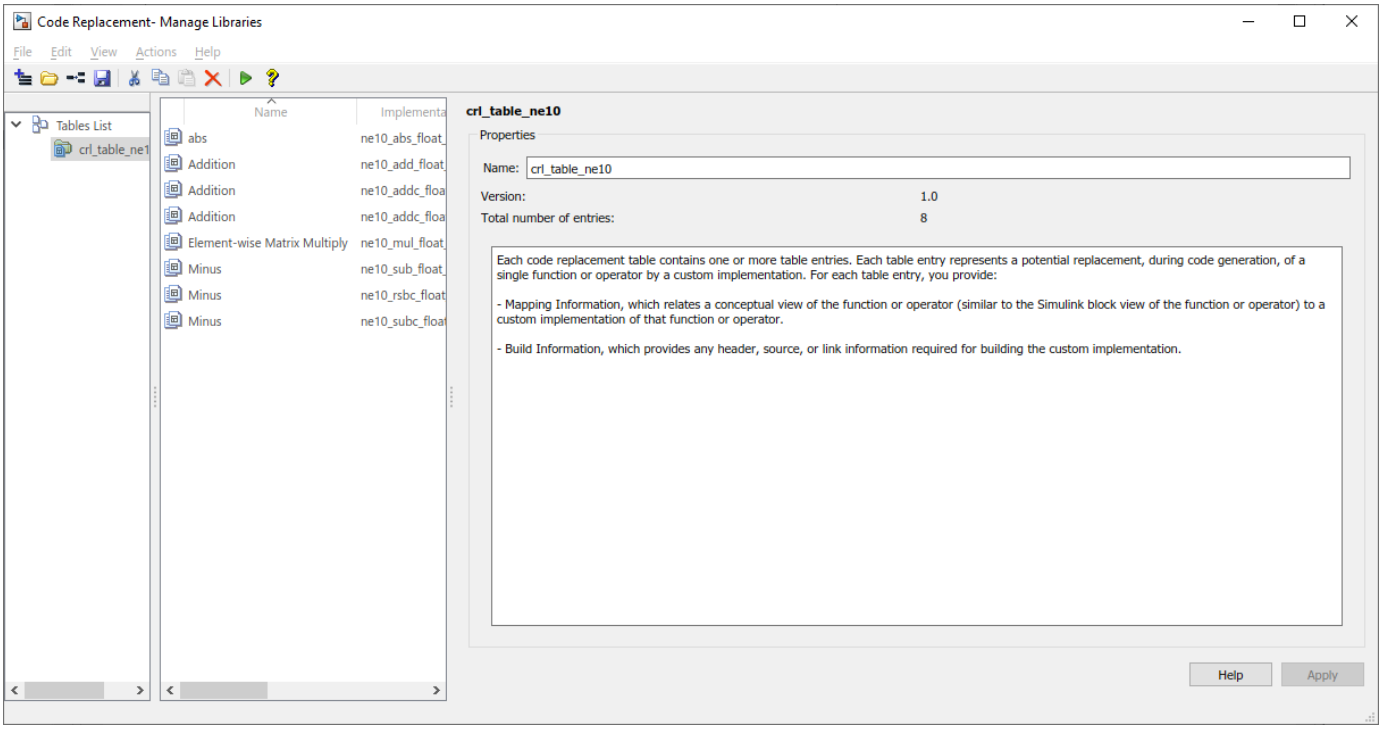
At the command prompt, type `crtool`.

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')
```



- “Quick Start Code Replacement Library Development - Simulink®” (Embedded Coder)

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.

`Out`*n*`Type` — 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
“Quick Start Code Replacement Library Development - Simulink®” (Embedded Coder)

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” (Embedded Coder).

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

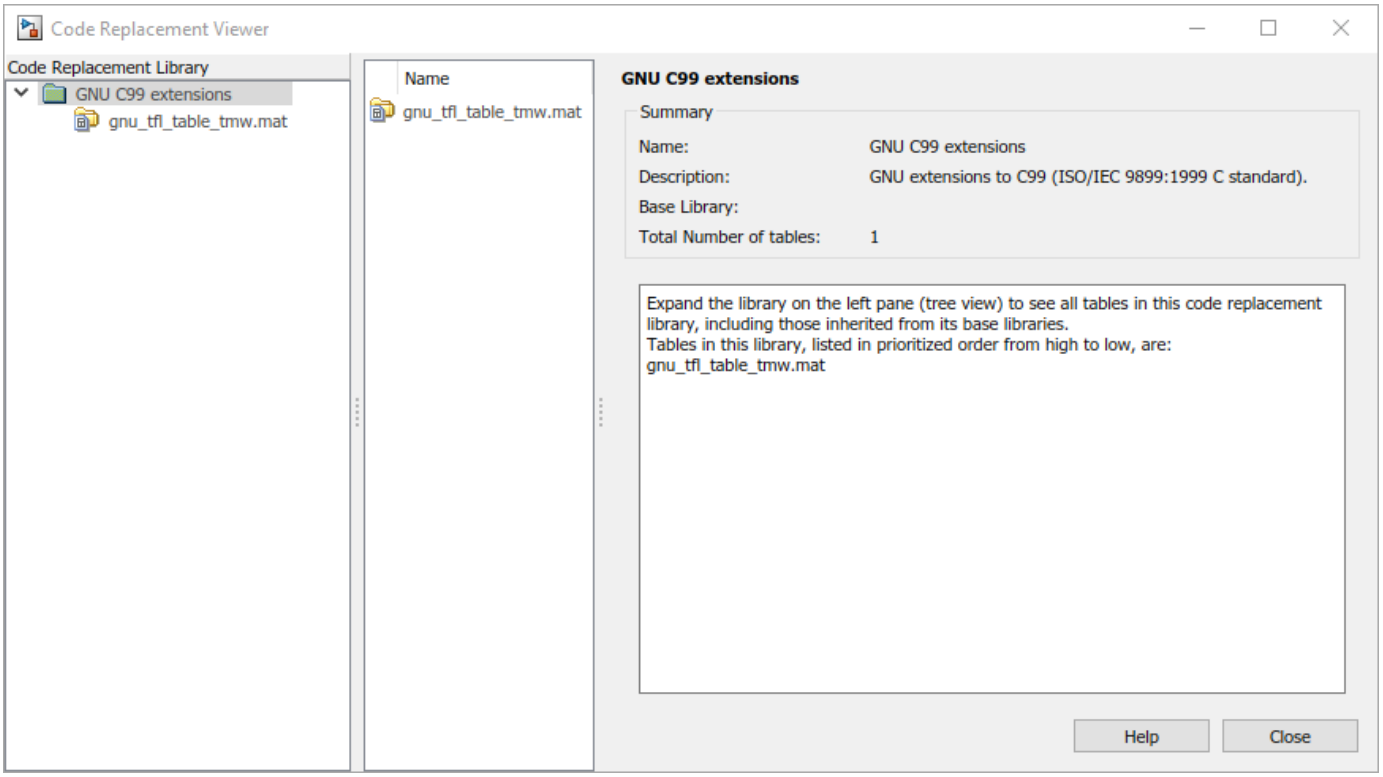
Open from the MATLAB command prompt using `crviewer`.

Examples

Display Contents of Code Replacement Library

This example opens the registered code replacement library `GNU C99 extensions`.

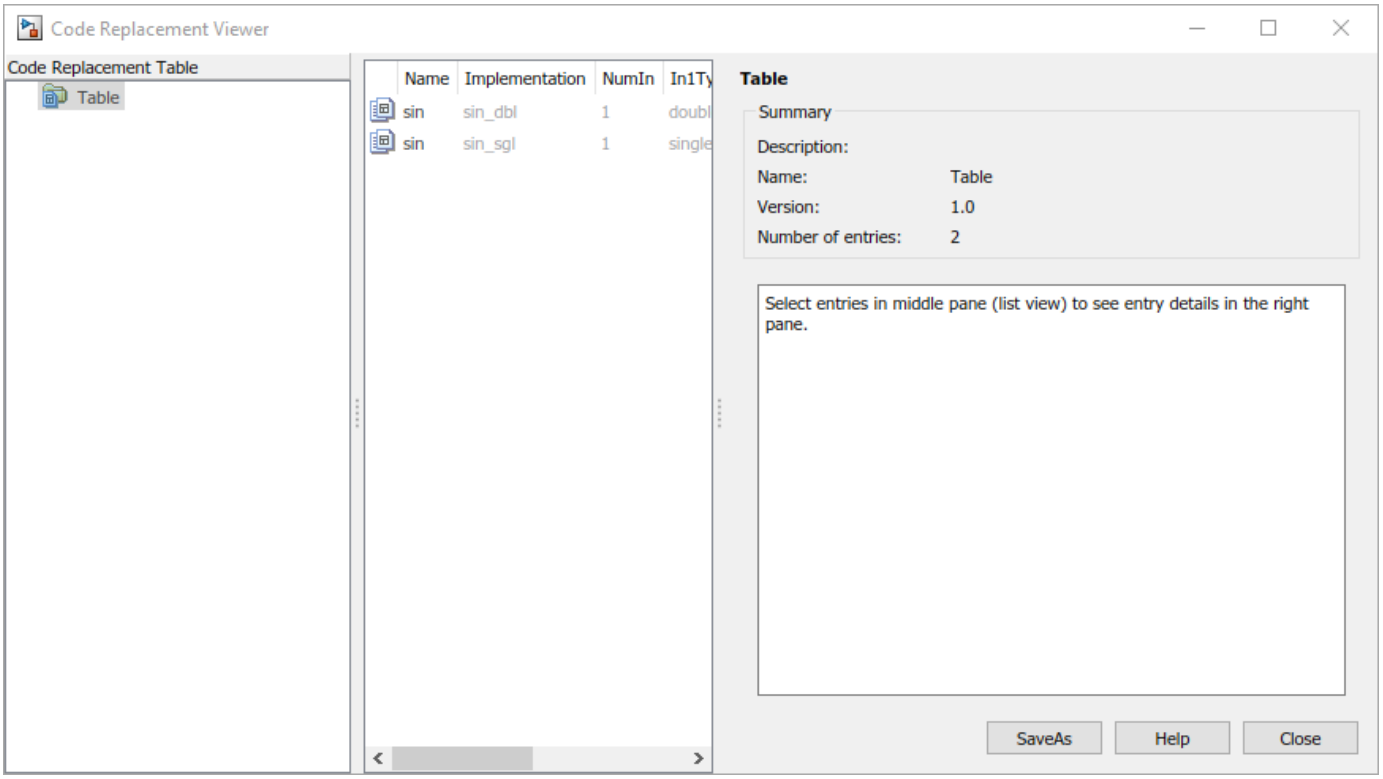
```
crviewer('GNU C99 extensions')
```



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” (Embedded Coder).

```
crviewer(crl_table_sinfcn)
```



- “Choose a 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”
“What Is Code Replacement?”
“Code Replacement Libraries”
“Code Replacement Terminology”

Introduced in R2014b

Optimization Parameters

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” on page 19-9	Transform numeric block parameters into constant inlined values in the generated code.
“Pass reusable subsystem outputs as” (Embedded Coder)	Specify how a reusable subsystem passes outputs.
“Remove root level I/O zero initialization” (Embedded Coder)	Specify whether to generate initialization code for root-level inports and outports set to zero.
“Remove internal data zero initialization” (Embedded Coder)	Specify whether to generate initialization code for internal work structures, such as block states and block outputs, to zero.
“Level” (Embedded Coder)	Choose the optimization level that you want to apply to the generated code.
“Priority” (Embedded Coder)	Optimize the generated code for increased execution efficiency, decreased RAM consumption, or a balance between the two.
“Specify custom optimizations” (Embedded Coder)	Instead of applying an optimization level, select this parameter to select the optimization parameters in the **Details** section.
“Use memcpy for vector assignment” on page 19-13	Optimize code generated for vector assignment by replacing `for` loops with `memcpy`.
“Memcpy threshold (bytes)” on page 19-15	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” on page 19-25	Specify whether block signals are declared locally or globally.
“Reuse local block outputs” on page 19-27	Specify whether Simulink Coder software reuses signal memory.
“Eliminate superfluous local variables (Expression folding)” on page 19-29	Collapse block computations into single expressions.
“Reuse global block outputs” (Embedded Coder)	Reuse global memory for block outputs.
“Perform in-place updates for Assignment and Bus Assignment blocks” (Embedded Coder)	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” (Embedded Coder)	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” (Embedded Coder)	Replace multiply operations in array indices when accessing arrays in a loop.
“Pack Boolean data into bitfields” (Embedded Coder)	Specify whether Boolean signals are stored as one–bit bitfields or as a Boolean data type.
“Bitfield declarator type specifier” (Embedded Coder)	Specify the bitfield type when selecting configuration parameter “Pack Boolean data into bitfields” (Embedded Coder).
“Reuse buffers of different sizes and dimensions” (Embedded Coder)	Reduce memory consumption by reusing buffers to store data of different sizes and dimensions.
“Optimize global data access” (Embedded Coder)	Select global variable optimization.
“Optimize block operation order in the generated code” (Embedded Coder)	Reorder block operations in the generated code for improved code execution speed.
“Use bitsets for storing state configuration” on page 19-19	Use bitsets to reduce the amount of memory required to store state configuration variables.
“Use bitsets for storing Boolean data” on page 19-21	Use bitsets to reduce the amount of memory required to store Boolean data.
“Maximum stack size (bytes)” on page 19-17	Specify the maximum stack size in bytes for your model.
“Loop unrolling threshold” on page 19-16	Specify the minimum signal or parameter width for which a `for` loop is generated.
“Optimize using the specified minimum and maximum values” (Embedded Coder)	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” on page 19-11	Transform symbolic names of invariant signals into constant values.
“Remove code from floating-point to integer conversions with saturation that maps NaN to zero” on page 19-31	Remove code that handles floating-point to integer conversion results for `NaN` values.
“Use memset to initialize floats and doubles to 0.0” on page 19-33	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” on page 19-6	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” (Embedded Coder)	Remove wrapping code of tunable parameters.
“Remove code that protects against division arithmetic exceptions” (Embedded Coder)	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” on page 19-8	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” on page 19-23	Set the storage type and size for enumerations created with active state output.
“Use signal labels to guide buffer reuse” (Embedded Coder)	For signals with the same label, the code generator attempts to use the same signal memory.
“Generate parallel for-loops” (Embedded Coder)	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” on page 19-35	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” (Embedded Coder)	Specify whether the generated code contains bitwise or logical operators or both.

See Also

Related Examples

- “Performance”

Optimization Pane: Tab Overview

Set up optimizations for a model's active configuration set. Optimizations are set for both simulation and code generation.

Tips

- To open the Optimization pane For ERT-based targets, in the Simulink Editor, select **Apps > Embedded Coder > Settings > Configuration Parameters > Optimization**.
- To open the Optimization pane For GRT-based targets, in the Simulink Editor, select **Apps > Simulink Coder > Settings > Configuration Parameters > Optimization**.
- Simulink Coder optimizations appear only when the Simulink Coder product is installed on your system. Selecting a GRT-based or ERT-based system target file changes the available options. ERT-based target optimizations require a Embedded Coder license when generating code. See the **Dependencies** sections below for licensing information for each parameter.

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 Optimization” on page 19-2
- “Perform Acceleration”
- “Performance”

Remove code from floating-point to integer conversions that wraps out-of-range values

Description

Remove wrapping code that handles out-of-range floating-point to integer conversion results.

Category: Optimization

Settings

Default: Off

On

Removes code when out-of-range conversions occur. Select this check box if code efficiency is critical to your application and the following conditions are true for at least one block in the model:

- Computing the outputs or parameters of a block involves converting floating-point data to integer or fixed-point data.
- The **Saturate on integer overflow** check box is cleared in the Block Parameters dialog box.

Caution Execution of generated code might not produce the same results as simulation.

Off

Results for simulation and execution of generated code match when out-of-range conversions occur. The generated code is larger than when you select this check box.

Tips

- Selecting this check box reduces the size and increases the speed of the generated code at the cost of potentially producing results that do not match simulation in the case of out-of-range values.
- Selecting this check box affects code generation results only for out-of-range values and cannot cause code generation results to differ from simulation results for in-range values.

Dependency

This parameter requires a Simulink Coder license.

Command-Line Information

Parameter: `EfficientFloat2IntCast`
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	Off
Traceability	Off
Efficiency	On (execution, ROM), No impact (RAM)
Safety precaution	No impact

See Also

Related Examples

- “Remove Code From Floating-Point to Integer Conversions That Wraps Out-of-Range Values”
- “Model Configuration Parameters: Code Generation Optimization” on page 19-2

Buffer for reusable subsystems

Description

Specify whether to generate code that has buffers to enhance the reuse of subsystems.

Category: Optimization

Settings

Default: On

 On

Generates code that has extra buffers that enable the reuse of subsystems in the generated code which enhances ROM efficiency.

 Off

Generates code that does not have extra buffers that could have potentially increased the reuse of a subsystem in the generated code. This code generation reduces ROM efficiency. The absence of the extra buffers might avoid extra data copies in the generated code, which improves code execution speed.

Command-Line Information

Parameter: `BufferReusableBoundary`
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No Impact
Traceability	No Impact
Efficiency	Off (Execution Speed), On (ROM)
Safety precaution	No Impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Optimization” on page 19-2
- “Specify Buffer Reuse by Using Simulink.Signal Objects” (Embedded Coder)

Default parameter behavior

Description

Transform numeric block parameters into constant inlined values in the generated code.

Category: Optimization

Settings

Default: `Tunable` for GRT targets | `Inlined` for ERT targets

`Inlined`

Set **Default parameter behavior** to `Inlined` to reduce global RAM usage and increase efficiency of the generated code. The code does not allocate memory to represent numeric block parameters such as the **Gain** parameter of a Gain block. Instead, the code inlines the literal numeric values of these block parameters.

`Tunable`

Set **Default parameter behavior** to `Tunable` to enable tunability of numeric block parameters in the generated code. The code represents numeric block parameters and variables that use the storage class `Auto`, including numeric MATLAB variables, as tunable fields of a global parameters structure.

Tips

- Whether you set **Default parameter behavior** to `Inlined` or to `Tunable`, create parameter data objects to preserve tunability for block parameters. For more information, see “Create Tunable Calibration Parameter in the Generated Code”.
- When you switch from a system target file that is not ERT-based to one that is ERT-based, **Default parameter behavior** sets to `Inlined` by default. However, you can change the setting of **Default parameter behavior** later.
- When a top model uses referenced models or if a model is referenced by another model:
  - Referenced models must set **Default parameter behavior** to `Inlined` if the top model has **Default parameter behavior** set to `Inlined`.
  - The top model can specify **Default parameter behavior** as `Tunable` or `Inlined`.
- If your model contains an Environment Controller block, you can suppress code generation for the branch connected to the Sim port if you set **Default parameter behavior** to `Inlined` and the branch does not contain external signals.

Dependencies

When you set **Default parameter behavior** to `Inlined`, you enable “Inline invariant signals” on page 19-11 configuration parameter.

Command-Line Information

Parameter: `DefaultParameterBehavior`
Type: character vector
Value: `'Inlined'` | `'Tunable'`
Default: `'Tunable'` for GRT targets | `'Inlined'` for ERT targets

Recommended Settings

Application	Setting
Debugging	`Tunable` during development `Inlined` for production code generation
Traceability	`Inlined`
Efficiency	`Inlined`
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Optimization” on page 19-2
- “Inline Numeric Values of Block Parameters”
- “How Generated Code Stores Internal Signal, State, and Parameter Data”

Inline invariant signals

Description

Transform symbolic names of invariant signals into constant values.

Category: Optimization

Settings

Default: Off

On

Simulink Coder software uses the numerical values of model parameters, instead of their symbolic names, in generated code. An invariant signal is not inline if it is nonscalar, complex, or the block inport the signal is attached to takes the address of the signal.

Off

Uses symbolic names of model parameters in generated code.

Dependencies

- This parameter requires a Simulink Coder license.
- This parameter is enabled when you set **Default parameter behavior** to `Inlined`.

Command-Line Information

Parameter: `InlineInvariantSignals`
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 19-2
- “Inline Invariant Signals”

- “Performance”

Use memcpy for vector assignment

Description

Optimize code generated for vector assignment by replacing `for` loops with `memcpy`.

Category: Optimization

Settings

Default: On

On

Enables use of `memcpy` for vector assignment based on the associated threshold parameter **Memcpy threshold (bytes)**. `memcpy` is used in the generated code if the number of array elements times the number of bytes per element is greater than or equal to the specified value for **Memcpy threshold (bytes)**. One byte equals the width of a character in this context.

Off

Disables use of `memcpy` for vector assignment.

Dependencies

- This parameter requires a Simulink Coder license.
- When selected, this parameter enables the associated parameter **Memcpy threshold (bytes)**.

Command-Line Information

Parameter: `EnableMemcpy`
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 Generation Optimization” on page 19-2
- “Use memcpy Function to Optimize Generated Code for Vector Assignments”

- “Performance”

Memcpy threshold (bytes)

Description

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.

Category: Optimization

Settings

Default: `64`

Dependencies

- This parameter requires a Simulink Coder license.
- For the `memcpy` optimization, this parameter is enabled when you select **Use memcpy for vector assignment**.

Command-Line Information

Parameter: `MemcpyThreshold`
Type: integer
Value: valid quantity of bytes
Default: `64`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Accept default or determine target-specific optimal value
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Optimization” on page 19-2
- “Use memcpy Function to Optimize Generated Code for Vector Assignments”
- “Performance”

Loop unrolling threshold

Description

Specify the minimum array size width for which a `for` loop is generated.

Category: Optimization

Settings

Default: 5

Specify the array size at which the code generator begins to use a `for` loop instead of separate assignment statements to assign values..

When there are perfectly nested loops, the code generator uses a for loop if the product of the loop counts for loops in the perfect loop nest is greater than or equal to the threshold.

Dependency

This parameter requires a Simulink Coder license.

Command-Line Information

Parameter: `RollThreshold`
Type: character vector
Value: valid value
Default: `'5'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	>0
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Optimization” on page 19-2
- “Configure Loop Unrolling Threshold”
- “Performance”

Maximum stack size (bytes)

Description

Specify the maximum stack size in bytes for your model.

Category: Optimization

Settings

Default:`Inherit from target`

`Inherit from target`

The Simulink Coder software assigns the maximum stack size to the smaller value of the following:

- The default value (200,000 bytes) set by the Simulink Coder software
- Value of the TLC variable `MaxStackSize` in the system target file

`<Specify a value>`

Specify a positive integer value. Simulink Coder software assigns the maximum stack size to the specified value.

Note If you specify a maximum stack size for a model, the estimated required stack size of a referenced model must be less than the specified maximum stack size of the parent model.

Tips

- If you specify the maximum stack size to be zero, then the generated code implements all variables as global data.
- If you specify the maximum stack to be `inf`, then the generated code contains the least number of global variables.
- If your model contains a variable that is larger than 4096 bytes, the code generator implements it in global memory by default. You can increase the size of variables that the code generator places in local memory by changing the value of the TLC variable `MaxStackVariableSize`. You can change this value by typing the following command in MATLAB Command Window: `set_param(modelName,'TLCOptions','-aMaxStackVariableSize=N')`

Command-Line Information

Parameter: `MaxStackSize`
Type: `int`
Value: valid value
Default: `Inherit from target`

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 Optimization” on page 19-2
- “Customize Stack Space Allocation”
- “Performance”

Use bitsets for storing state configuration

Description

Use bitsets to reduce the amount of memory required to store state configuration variables.

Category: Optimization

Settings

Default: Off

 On

Stores state configuration variables in bitsets. Potentially reduces the amount of memory required to store the variables. Potentially requires more instructions to access state configuration, which can result in less optimal code.

 Off

Stores state configuration variables in unsigned bytes. Potentially increases the amount of memory required to store the variables. Potentially requires fewer instructions to access state configuration, which can result in more optimal code.

Tips

- Selecting this check box can significantly reduce the amount of memory required to store the variables. However, it can increase the amount of memory required to store target code if the target processor does not include instructions for manipulating bitsets.
- Select this check box for Stateflow charts that have a large number of sibling states at a given level of the hierarchy.
- Clear this check box for Stateflow charts with a small number of sibling states at a given level of the hierarchy.

Dependency

This parameter requires a Simulink Coder license.

Command-Line Information

Parameter: `StateBitsets`
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	Off
Traceability	Off

Application	Setting
Efficiency	Off (execution, ROM), On (RAM)
Safety precaution	No impact

See Also

Related Examples

- “Reduce Memory Usage for Boolean and State Configuration Variables”
- “Design Tips for Optimizing Generated Code for Stateflow Objects”

Use bitsets for storing Boolean data

Description

Use bitsets to reduce the amount of memory required to store Boolean data.

Category: Optimization

Settings

Default: Off

 On

Stores Boolean data in bitsets. Potentially reduces the amount of memory required to store the data. Potentially requires more instructions to access the data, which can result in less optimal code.

 Off

Stores Boolean data in unsigned bytes. Potentially increases the amount of memory required to store the data. Potentially requires fewer instructions to access the data, which can result in more optimal code.

Tips

- Select this check box for Stateflow charts that reference Boolean data infrequently.
- Clear this check box for Stateflow charts that reference Boolean data frequently.

Dependency

This parameter requires a Simulink Coder license.

Command-Line Information

Parameter: `DataBitsets`
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	Off
Traceability	Off
Efficiency	Off (execution, ROM), On (RAM)
Safety precaution	No impact

See Also

Related Examples

- “Reduce Memory Usage for Boolean and State Configuration Variables”
- “Design Tips for Optimizing Generated Code for Stateflow Objects”

Base storage type for automatically created enumerations

Description

Set the storage type and size for enumerations created with active state output.

Category: Optimization

Settings

Default: `'Native Integer'`

`'Native Integer'`
: Default target integer type

`int32`
: 32 bit signed integer type

`int16`
: 16 bit signed integer type

`int8`
: 8 bit signed integer type

`uint16`
: 16 bit unsigned integer type

`uint8`
: 8 bit unsigned integer type

Tips

- The default `'Native Integer'` is recommended for most models.
- If you need a smaller memory footprint for the generated enumerations, set the storage type to a smaller size. The size must be large enough to hold the number of states in the chart.

Dependency

This parameter requires a Simulink Coder license.

Command-Line Information

Parameter: `ActiveStateOutputEnumStorageType`
Value: `'Native Integer'` | `'int32'` | `'int16'` | `'int8'` | `'uint16'` | `'uint8'`
Default: `'Native Integer'`

See Also

Related Examples

- “Monitor State Activity Through Active State Data” (Stateflow)

- “Design Tips for Optimizing Generated Code for Stateflow Objects”

Enable local block outputs

Description

Specify whether block signals are declared locally or globally.

Category: Optimization

Settings

Default: On

On

Block signals are declared locally in functions.

Off

Block signals are declared globally.

Tips

- If it is not possible to declare an output as a local variable, the generated code declares the output as a global variable.
- If you are constrained by limited stack space, you can turn **Enable local block outputs** off and still benefit from memory reuse.

Dependencies

- This parameter requires a Simulink Coder license.
- This parameter is enabled by **Signal storage reuse**.

Command-Line Information

Parameter: `LocalBlockOutputs`
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	Off
Traceability	Off
Efficiency	On
Safety precaution	No impact

See Also

Related Examples

- “Enable and Reuse Local Block Outputs in Generated Code”
- “Performance”
- “Model Configuration Parameters: Code Generation Optimization” on page 19-2

Reuse local block outputs

Description

Specify whether Simulink Coder software reuses signal memory.

Category: Optimization

Settings

Default: On

On

- Simulink Coder software tries to reuse signal memory, reducing stack size where signals are being buffered in local variables.
- 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 a Simulink Coder license.

Command-Line Information

Parameter: `BufferReuse`
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	Off
Traceability	Off
Efficiency	On
Safety precaution	No impact

See Also

Related Examples

- “Enable and Reuse Local Block Outputs in Generated Code”
- “Performance”
- “Model Configuration Parameters: Code Generation Optimization” on page 19-2

Eliminate superfluous local variables (Expression folding)

Description

Collapse block computations into single expressions.

Category: Optimization

Settings

Default: On

On

- Enables expression folding.
- Eliminates local variables, incorporating the information into the main code statement.
- Improves code readability and efficiency.

Off

Disables expression folding.

Dependencies

- This parameter requires a Simulink Coder license.
- This parameter is enabled by **Signal storage reuse**.

Command-Line Information

Parameter: `ExpressionFolding`
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	Off
Traceability	No impact for simulation or during development Off for production code generation
Efficiency	On
Safety precaution	No impact

See Also

Related Examples

- “Minimize Computations and Storage for Intermediate Results at Block Outputs”

- “Performance”
- “Model Configuration Parameters: Code Generation Optimization” on page 19-2

Remove code from floating-point to integer conversions with saturation that maps NaN to zero

Description

Remove code that handles floating-point to integer conversion results for `NaN` values.

Category: Optimization

Settings

Default: On

On

Removes code when mapping from `NaN` to integer zero occurs. Select this check box if code efficiency is critical to your application and the following conditions are true for at least one block in the model:

- Computing outputs or parameters of a block involves converting floating-point data to integer or fixed-point data.
- The **Saturate on integer overflow** check box is selected in the Block Parameters dialog box.

Caution Execution of generated code might not produce the same results as simulation.

Off

Results for simulation and execution of generated code match when mapping from `NaN` to integer zero occurs. The generated code is larger than when you select this check box.

Tips

- Selecting this check box reduces the size and increases the speed of the generated code at the cost of producing results that do not match simulation in the case of `NaN` values.
- Selecting this check box affects code generation results only for `NaN` values and cannot cause code generation results to differ from simulation results for other values.

Dependencies

- This parameter requires a Simulink Coder license.
- For ERT-based targets, this parameter is enabled when you select the **floating-point numbers** and **non-finite numbers** check boxes in the **Code Generation > Interface** pane.

Command-Line Information

Parameter: `EfficientMapNaN2IntZero`
Value: `'on'` | `'off'`

Default: `'on'`

Recommended Settings

Application	Setting
Debugging	Off
Traceability	Off
Efficiency	On
Safety precaution	No recommendation

See Also

Related Examples

- “Remove Code That Maps NaN to Integer Zero”
- “Model Configuration Parameters: Code Generation Optimization” on page 19-2

Use memset to initialize floats and doubles to 0.0

Description

Specify whether to generate code that explicitly initializes floating-point data to 0.0.

Category: Optimization

Settings

Default: On (GUI), `'off'` (command line)

 On

Uses `memset` to clear internal storage for floating-point data to integer bit pattern 0 (all bits 0), regardless of type. If your compiler and target CPU both represent floating-point zero with the integer bit pattern `0`, use this parameter to gain execution and ROM efficiency.

This parameter requires that you turn on the configuration parameter **Memcpy threshold** to enable the memset functionality. Check that the value of the threshold is set high enough.

 Off

Generates code to explicitly initialize storage for data of types `float` and `double` to 0.0. The resulting code is slightly less efficient than code generated when you select the option.

Dependency

This parameter requires a Simulink Coder license.

Command-Line Information

Parameter: `InitFltsAndDblsToZero`
Value: `'on'` | `'off'`
Default: `'off'`

Note The command-line values are reverse of the settings values. Therefore, `'on'` in the command line corresponds to the description of “Off” in the settings section, and `'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 impact

See Also

Related Examples

- “Optimize Generated Code Using memset Function”
- “Model Configuration Parameters: Code Generation Optimization” on page 19-2

Signal storage reuse

Description

Reuse signal memory.

Category: Optimization

Settings

Default: On

On

Simulink software reuses memory buffers allocated to store block input and output signals, reducing the memory requirement of your real-time program.

Off

Simulink software allocates a separate memory buffer for each block's outputs. This makes all block outputs global and unique, which in many cases significantly increases RAM and ROM usage.

Tips

- This option applies only to signals with storage class `Auto`.
- Signal storage reuse can occur only among signals that have the same data type.
- Clearing this option can substantially increase the amount of memory required to simulate large models.
- Clear this option if you need to:
  - Debug a C-MEX S-function
  - Use a Floating Scope or a Floating Scope block with the **Floating display** option selected to inspect signals in a model that you are debugging
- Simulink software opens an error dialog if **Signal storage reuse** is enabled and you attempt to use a Floating Scope or floating Display block to display a signal whose buffer has been reused.

Dependencies

This parameter enables:

- **Enable local block outputs** on page 19-25
- **Reuse local block outputs** on page 19-27
- **Eliminate superfluous local variables (Expression folding)** on page 19-29

If you have an Embedded Coder license, this parameter enables:

- **Optimize global data access** (Embedded Coder)
- **Perform in-place updates for Assignment and Bus Assignment blocks** (Embedded Coder)

- **Reuse global block outputs** (Embedded Coder)
- **Optimize block operation order in the generated code** (Embedded Coder)
- **Reuse buffers for Data Store Read and Data Store Write blocks** (Embedded Coder)

Command-Line Information

Parameter:`OptimizeBlockIOStorage`
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	Off
Traceability	Off
Efficiency	On
Safety precaution	No impact

See Also

Related Examples

- “Minimize Computations and Storage for Intermediate Results at Block Outputs”
- “Performance”
- “Model Configuration Parameters: Code Generation Optimization” on page 19-2