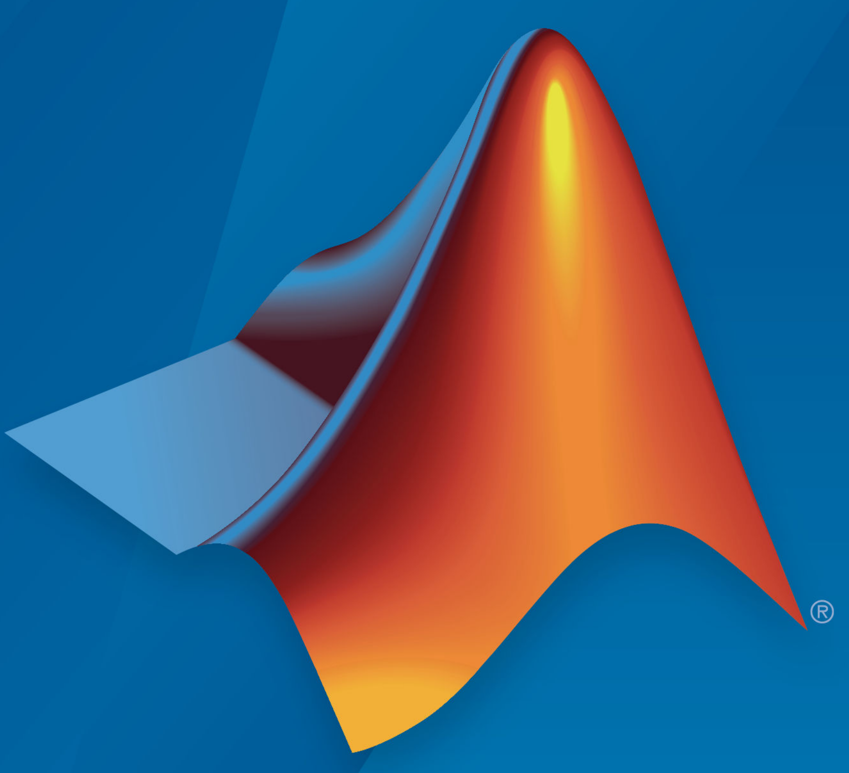
# Embedded Coder®

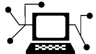
## Reference



MATLAB® & SIMULINK®

R2018a

# 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.
3 Apple Hill Drive
Natick, MA 01760-2098


*Embedded Coder® Reference*


© COPYRIGHT 2011–2018 by The MathWorks, Inc.

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

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

**Trademarks**

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

**Patents**

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

**Revision History**

| | | |
|---|---|---|
| April 2011 | Online only | New for Version 6.0 (Release 2011a) |
| September 2011 | Online only | Revised for Version 6.1 (Release 2011b) |
| March 2012 | Online only | Revised for Version 6.2 (Release 2012a) |
| September 2012 | Online only | Revised for Version 6.3 (Release 2012b) |
| March 2013 | Online only | Revised for Version 6.4 (Release 2013a) |
| September 2013 | Online only | Revised for Version 6.5 (Release 2013b) |
| March 2014 | Online only | Revised for Version 6.6 (Release 2014a) |
| October 2014 | Online only | Revised for Version 6.7 (Release 2014b) |
| March 2015 | Online only | Revised for Version 6.8 (Release 2015a) |
| September 2015 | Online only | Revised for Version 6.9 (Release 2015b) |
| October 2015 | Online only | Rereleased for Version 6.8.1 (Release 2015aSP1) |
| March 2016 | Online only | Revised for Version 6.10 (Release 2016a) |
| September 2016 | Online only | Revised for Version 6.11 (Release 2016b) |
| March 2017 | Online only | Revised for Version 6.12 (Release 2017a) |
| September 2017 | Online only | Revised for Version 6.13 (Release 2017b) |
| March 2018 | Online only | Revised for Version 7.0 (Release 2018a) |

# 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/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

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** Functions in Embedded Coder—Alphabetical List

**2** Functions in Simulink Coder—Alphabetical List

**3** Blocks in Embedded Coder—Alphabetical List

**4** Blocks in Simulink Coder—Alphabetical List

**5** Embedded Coder Parameters: Advanced Parameters

**Create block** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5-2
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5-2
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5-2
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . . . . 5-3
Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5-3

**Existing shared code** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5-4
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5-4

Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **5-4**
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . . **5-4**
Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . **5-4**

**Use only existing shared code . . . . . . . . . . . . . . . . . . . . . . . . . 5-6**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **5-6**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **5-6**
Dependency . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **5-6**
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . . **5-6**
Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . **5-6**

**Use Embedded Coder Features . . . . . . . . . . . . . . . . . . . . . . . . . 5-8**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **5-8**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **5-8**
Dependencies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **5-8**
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . . **5-8**

**Remove reset function . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5-10**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **5-10**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **5-10**
Dependencies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **5-10**
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . . **5-10**

**Remove disable function . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5-12**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **5-12**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **5-12**
Dependencies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **5-12**
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . . **5-12**

## 6 Code Generation Parameters: AUTOSAR

**Model Configuration Parameters: Code Generation AUTOSAR . . . . . . . . . . . . . . . . . . . . . . . . . . 6-2**

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

**Generate XML file for schema version** . . . . . . . . . . . . . . . . . . . **6-4**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **6-4**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **6-4**
Tip . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **6-4**
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . . **6-5**

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

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

**Support root-level matrix I/O using one-dimensional arrays** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **6-9**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **6-9**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **6-9**
Tip . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **6-9**
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . . **6-9**

# 7 Code Generation Parameters: Code Placement

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

**Code Generation: Code Placement Tab Overview** . . . . . . . . . . . **7-4**
Configuration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-4**
To get help on an option . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-4**

**Data definition** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-5**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-5**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-5**
Dependencies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-5**
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . . **7-5**

Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-6**

**Data definition filename** . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-7**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-7**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-7**
Dependency . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-7**
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-7**
Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-8**

**Data declaration** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-9**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-9**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-9**
Dependencies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-9**
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-9**
Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-10**

**Data declaration filename** . . . . . . . . . . . . . . . . . . . . . . . . . . **7-11**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-11**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-11**
Dependency . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-11**
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . . . . **7-11**
Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-11**

**Use owner from data object for data definition placement** . . **7-13**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-13**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-13**
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . . . . **7-13**
Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-13**

**#include file delimiter** . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-15**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-15**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-15**
Dependency . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-15**
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . . . . **7-15**
Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-15**

**Signal display level** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-17**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-17**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-17**
Dependency . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-17**
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . . . . **7-17**
Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-17**

**Parameter tune level** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-19**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-19**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-19**
Dependency . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-19**
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . . **7-19**
Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . . **7-19**

**File packaging format** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-21**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-21**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-21**
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . . **7-22**
Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . . **7-22**

**Header files** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-24**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-24**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-24**
Dependency . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-25**
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . . **7-25**
Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . . **7-25**

**Source files** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-26**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-26**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-26**
Dependency . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-27**
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . . **7-27**
Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . . **7-27**

**Data files** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-28**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-28**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-28**
Dependency . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-28**
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . . **7-29**
Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . . **7-29**

**Rate Transition block code** . . . . . . . . . . . . . . . . . . . . . . . . . **7-30**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-30**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-30**
Dependencies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7-30**
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . . **7-30**
Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . . **7-31**

# 8 Code Generation Parameters: Code Style

**Model Configuration Parameters: Code Generation Code Style** . . . . . . . . **8-2**

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

**Parentheses level** . . . . . . . . **8-5**
Description . . . . . . . . **8-5**
Settings . . . . . . . . **8-5**
Command-Line Information . . . . . . . . **8-5**
Recommended Settings . . . . . . . . **8-6**

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

**Preserve condition expression in if statement** . . . . . . . . **8-9**
Description . . . . . . . . **8-9**
Settings . . . . . . . . **8-9**
Command-Line Information . . . . . . . . **8-9**
Recommended Settings . . . . . . . . **8-10**

**Convert if-elseif-else patterns to switch-case statements** . . . **8-11**
Description . . . . . . . . **8-11**
Settings . . . . . . . . **8-11**
Command-Line Information . . . . . . . . **8-12**
Recommended Settings . . . . . . . . **8-12**

**Preserve extern keyword in function declarations** . . . . . . . . **8-13**
Description . . . . . . . . **8-13**
Settings . . . . . . . . **8-13**
Command-Line Information . . . . . . . . **8-13**
Recommended Settings . . . . . . . . **8-14**

**Preserve static keyword in function declarations** . . . . . . . . **8-15**
Description . . . . . . . . **8-15**

Settings . . . . . . . . . . 8-15
Dependency . . . . . . . . . . 8-15
Command-Line Information . . . . . . . . . . 8-16
Recommended Settings . . . . . . . . . . 8-16

**Suppress generation of default cases for Stateflow switch statements if unreachable . . . . . . . . . . 8-17**
Description . . . . . . . . . . 8-17
Settings . . . . . . . . . . 8-17
Command-Line Information . . . . . . . . . . 8-17
Recommended Settings . . . . . . . . . . 8-18

**Replace multiplications by powers of two with signed bitwise shifts . . . . . . . . . . 8-19**
Description . . . . . . . . . . 8-19
Settings . . . . . . . . . . 8-19
Command-Line Information . . . . . . . . . . 8-20
Recommended Settings . . . . . . . . . . 8-20

**Allow right shifts on signed integers . . . . . . . . . . 8-21**
Description . . . . . . . . . . 8-21
Settings . . . . . . . . . . 8-21
Command-Line Information . . . . . . . . . . 8-21
Recommended Settings . . . . . . . . . . 8-22

**Casting modes . . . . . . . . . . 8-23**
Description . . . . . . . . . . 8-23
Settings . . . . . . . . . . 8-23
Command-Line Information . . . . . . . . . . 8-24
Recommended Settings . . . . . . . . . . 8-24

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

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

**Newline style** . . . . . . . . . . 8-29
Description . . . . . . . . . . 8-29
Settings . . . . . . . . . . 8-29
Command-Line Information . . . . . . . . . . 8-29
Recommended Settings . . . . . . . . . . 8-29

## 9 Code Generation Parameters: Data Type Replacement

**Model Configuration Parameters: Code Generation Data Type Replacement** . . . . . . . . . . 9-2
Configure Data Type Replacements Programmatically . . . . . . 9-3

**Code Generation: Data Type Replacement Tab** . . . . . . . . . . 9-4
Configuration . . . . . . . . . . 9-4
To get help on an option . . . . . . . . . . 9-4

**Replace data type names in the generated code** . . . . . . . . . . 9-5
Description . . . . . . . . . . 9-5
Settings . . . . . . . . . . 9-5
Dependencies . . . . . . . . . . 9-6
Command-Line Information . . . . . . . . . . 9-6
Recommended Settings . . . . . . . . . . 9-6

**Replacement Name: double** . . . . . . . . . . 9-7
Description . . . . . . . . . . 9-7
Settings . . . . . . . . . . 9-7
Dependency . . . . . . . . . . 9-8
Command-Line Information . . . . . . . . . . 9-8
Recommended Settings . . . . . . . . . . 9-8

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

**Replacement Name: int32** . . . . . . . . . . 9-11
Description . . . . . . . . . . 9-11

Settings . . . . . . . . . . 9-11
Dependency . . . . . . . . . . 9-11
Command-Line Information . . . . . . . . . . 9-12
Recommended Settings . . . . . . . . . . 9-12

**Replacement Name: int16** . . . . . . . . . . **9-13**
Description . . . . . . . . . . 9-13
Settings . . . . . . . . . . 9-13
Dependency . . . . . . . . . . 9-13
Command-Line Information . . . . . . . . . . 9-14
Recommended Settings . . . . . . . . . . 9-14

**Replacement Name: int8** . . . . . . . . . . **9-15**
Description . . . . . . . . . . 9-15
Settings . . . . . . . . . . 9-15
Dependency . . . . . . . . . . 9-15
Command-Line Information . . . . . . . . . . 9-15
Recommended Settings . . . . . . . . . . 9-16

**Replacement Name: uint32** . . . . . . . . . . **9-17**
Description . . . . . . . . . . 9-17
Settings . . . . . . . . . . 9-17
Dependency . . . . . . . . . . 9-17
Command-Line Information . . . . . . . . . . 9-18
Recommended Settings . . . . . . . . . . 9-18

**Replacement Name: uint16** . . . . . . . . . . **9-19**
Description . . . . . . . . . . 9-19
Settings . . . . . . . . . . 9-19
Dependency . . . . . . . . . . 9-19
Command-Line Information . . . . . . . . . . 9-20
Recommended Settings . . . . . . . . . . 9-20

**Replacement Name: uint8** . . . . . . . . . . **9-21**
Description . . . . . . . . . . 9-21
Settings . . . . . . . . . . 9-21
Dependency . . . . . . . . . . 9-21
Command-Line Information . . . . . . . . . . 9-22
Recommended Settings . . . . . . . . . . 9-22

**Replacement Name: boolean** . . . . . . . . . . **9-23**
Description . . . . . . . . . . 9-23
Settings . . . . . . . . . . 9-23

Dependency . . . . . . . . . . 9-24
Command-Line Information . . . . . . . . . . 9-24
Recommended Settings . . . . . . . . . . 9-24

**Replacement Name: int** . . . . . . . . . . **9-26**
Description . . . . . . . . . . 9-26
Settings . . . . . . . . . . 9-26
Dependency . . . . . . . . . . 9-27
Command-Line Information . . . . . . . . . . 9-27
Recommended Settings . . . . . . . . . . 9-27

**Replacement Name: uint** . . . . . . . . . . **9-28**
Description . . . . . . . . . . 9-28
Settings . . . . . . . . . . 9-28
Dependency . . . . . . . . . . 9-29
Command-Line Information . . . . . . . . . . 9-29
Recommended Settings . . . . . . . . . . 9-29

**Replacement Name: char** . . . . . . . . . . **9-30**
Description . . . . . . . . . . 9-30
Settings . . . . . . . . . . 9-30
Dependency . . . . . . . . . . 9-30
Command-Line Information . . . . . . . . . . 9-30
Recommended Settings . . . . . . . . . . 9-31

## 10 Memory Sections Parameters on the Code Generation Pane

**Code Generation: Memory Sections Tab Overview** . . . . . . . . . . **10-2**
Configuration . . . . . . . . . . 10-2
To get help on an option . . . . . . . . . . 10-2

**Package** . . . . . . . . . . **10-3**
Description . . . . . . . . . . 10-3
Settings . . . . . . . . . . 10-3
Tip . . . . . . . . . . 10-3
Command-Line Information . . . . . . . . . . 10-3
Recommended Settings . . . . . . . . . . 10-4

**Refresh package list** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **10-5**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **10-5**
Tip . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **10-5**

**Initialize/Terminate** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **10-6**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **10-6**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **10-6**
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . . **10-6**
Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . . **10-6**

**Execution** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **10-8**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **10-8**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **10-8**
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . . **10-8**
Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . . **10-8**

**Shared utility** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **10-10**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **10-10**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **10-10**
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . **10-10**
Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . **10-10**

**Constants** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **10-12**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **10-12**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **10-12**
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . **10-12**
Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . **10-13**

**Inputs/Outputs** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **10-14**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **10-14**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **10-14**
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . **10-14**
Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . **10-15**

**Internal data** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **10-16**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **10-16**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **10-16**
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . **10-16**
Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . **10-17**

**Parameters** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **10-18**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **10-18**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **10-18**

Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . . **10-18**
Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . . **10-19**

**Validation results** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **10-20**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **10-20**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **10-20**
Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . . **10-20**

# 11 Code Generation Parameters: Templates

**Model Configuration Parameters: Code Generation Templates** . . . . . . . . . . . . . . . . . . . . . . . . . . . . **11-2**

**Code Generation: Templates Tab Overview** . . . . . . . . . . . . . . . **11-4**
Configuration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **11-4**
To get help on an option . . . . . . . . . . . . . . . . . . . . . . . . . . **11-4**

**Code templates: Source file (*.c) template** . . . . . . . . . . . . . . **11-5**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **11-5**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **11-5**
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . . **11-5**
Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . **11-5**

**Code templates: Header file (*.h) template** . . . . . . . . . . . . . . **11-7**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **11-7**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **11-7**
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . . **11-7**
Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . **11-7**

**Data templates: Source file (*.c) template** . . . . . . . . . . . . . . . **11-9**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **11-9**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **11-9**
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . . **11-9**
Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . **11-9**

**Data templates: Header file (*.h) template** . . . . . . . . . . . . . **11-11**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **11-11**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **11-11**
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . **11-11**

Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . 11-11

**File customization template** . . . . . . . . . . . . . . . . . . . . . . . . . 11-13
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-13
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-13
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . . 11-13
Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . 11-13

**Generate an example main program** . . . . . . . . . . . . . . . . . . . 11-15
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-15
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-15
Tips . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-15
Dependencies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-16
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . . 11-16
Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . 11-16

**Target operating system** . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-18
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-18
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-18
Dependencies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-18
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . . 11-18
Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . 11-19

## 12 Code Generation Parameters: Verification

**Model Configuration Parameters: Code Generation Verification** . . . . . . . . . . . . . 12-2

**Code Generation: Verification Tab Overview** . . . . . . . . . . . . . . 12-4
Configuration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12-4
To get help on an option . . . . . . . . . . . . . . . . . . . . . . . . . 12-4

**Measure task execution time** . . . . . . . . . . . . . . . . . . . . . . . . 12-5
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12-5
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12-5
Dependencies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12-5
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . . 12-5
Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . 12-6

**Measure function execution times** . . . . . . . . . . . . . . . . . . . . **12-7**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **12-7**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **12-7**
Dependencies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **12-7**
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . . **12-7**
Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . **12-8**

**Workspace variable** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **12-9**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **12-9**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **12-9**
Dependency . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **12-9**
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . . **12-9**
Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . **12-9**

**Save options** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **12-11**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **12-11**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **12-11**
Dependency . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **12-11**
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . **12-11**
Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . . **12-12**

**Third-party tool** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **12-13**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **12-13**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **12-13**
Dependencies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **12-13**
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . **12-13**
Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . . **12-14**

**Enable portable word sizes** . . . . . . . . . . . . . . . . . . . . . . . . **12-15**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **12-15**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **12-15**
Dependencies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **12-15**
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . **12-15**
Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . . **12-16**

**Enable source-level debugging for SIL** . . . . . . . . . . . . . . . . **12-17**
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **12-17**
Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **12-17**
Command-Line Information . . . . . . . . . . . . . . . . . . . . . . . **12-17**
Recommended Settings . . . . . . . . . . . . . . . . . . . . . . . . . . **12-17**

# 13 Configuration Parameters

**Code Generation: Coder Target Pane** . . . . . . . . . . . . . . . . . . . . **13-2**
Code Generation: Coder Target Pane Overview (previously “IDE Link Tab Overview”) . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-3**
Coder Target: Tool Chain Automation Tab Overview . . . . . . . **13-4**
Build format . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-5**
Build action . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-6**
Overrun notification . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-9**
Function name . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-10**
Configuration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-11**
Compiler options string . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-12**
Linker options string . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-13**
System stack size (MAUs) . . . . . . . . . . . . . . . . . . . . . . . . . . **13-14**
System heap size (MAUs) . . . . . . . . . . . . . . . . . . . . . . . . . . **13-16**
Profile real-time execution . . . . . . . . . . . . . . . . . . . . . . . . . **13-17**
Profile by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-18**
Number of profiling samples to collect . . . . . . . . . . . . . . . . **13-19**
Maximum time allowed to build project (s) . . . . . . . . . . . . . **13-20**
Maximum time allowed to complete IDE operation (s) . . . . . **13-22**
Export IDE link handle to base workspace . . . . . . . . . . . . . **13-23**
IDE link handle name . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-24**
Source file replacement . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-25**

**Code Generation: Target Hardware Resources Pane** . . . . . . **13-27**
Code Generation: Coder Target Pane Overview . . . . . . . . . . **13-28**
(Target Hardware Resources) . . . . . . . . . . . . . . . . . . . . . . . **13-28**
Coder Target: Target Hardware Resources Tab Overview . . **13-29**
IDE/Tool Chain . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-29**
Target Hardware Resources: Board Tab . . . . . . . . . . . . . . . **13-30**
Target Hardware Resources: Memory Tab . . . . . . . . . . . . . **13-33**
Target Hardware Resources: Section Tab . . . . . . . . . . . . . . **13-36**
Target Hardware Resources: DSP/BIOS Tab . . . . . . . . . . . . **13-38**
Target Hardware Resources: Peripherals Tab . . . . . . . . . . . **13-41**
C28x-Clocking . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-44**
C28x-ADC . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-47**
C28-COMP . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-50**
C28x-eCAN_A, C28x-eCAN_B . . . . . . . . . . . . . . . . . . . . . . **13-51**
C28x-eCAP . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-54**
C28x-ePWM . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-57**
C28x-I2C . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-60**
C28x-SCI_A, C28x-SCI_B, C28x-SCI_C . . . . . . . . . . . . . . . . **13-66**

C28x-SPI_A, C28x-SPI_B, C28x-SPI_C, C28x-SPI_D . . . . . . . **13-69**
C28x-eQEP . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-72**
C28x-Watchdog . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-74**
C28x-GPIO . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-76**
C28x-DMA_ch[#] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-81**
C28x-LIN . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-90**
Add Processor Dialog Box . . . . . . . . . . . . . . . . . . . . . . . **13-96**
Target Hardware Resources Tab: Linux, VxWorks, or Windows . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-97**

**Hardware Implementation Pane: Altera Cyclone V SoC development kit, Arrow SoCKit development board** . . . . . **13-99**
Hardware Implementation Pane Overview . . . . . . . . . . . **13-100**
Operating system options . . . . . . . . . . . . . . . . . . . . . . . **13-100**
Clocking . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-100**
Build Options . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-100**
External mode . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-101**

**Hardware Implementation Pane: ARM Cortex-A9 (QEMU)** **13-102**
Hardware Implementation Pane Overview . . . . . . . . . . . **13-102**
Operating system options . . . . . . . . . . . . . . . . . . . . . . . **13-103**
Clocking . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-103**
Build Options . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-103**
External mode . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-104**

**Hardware Implementation Pane: ARM Cortex-M3 (QEMU)** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-106**
Hardware Implementation Pane Overview . . . . . . . . . . . **13-106**
Clocking . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-107**
External mode . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-107**

**Hardware Implementation Pane** . . . . . . . . . . . . . . . . . . . . **13-108**
Hardware Implementation Pane Overview . . . . . . . . . . . **13-108**
Build options . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-110**
Clocking . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-111**
DAC . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-112**
UART0, UART1, UART2, and UART3 . . . . . . . . . . . . . . . . **13-113**
Ethernet . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-116**
External mode . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-118**

**Hardware Implementation Pane: Freescale FRDM-KL25Z** **13-120**
Code Generation Pane . . . . . . . . . . . . . . . . . . . . . . . . . **13-121**
Scheduler options . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-121**

Build Options . . . . . . . . . . 13-121
Clocking . . . . . . . . . . 13-122
I2C0 . . . . . . . . . . 13-122
I2C1 . . . . . . . . . . 13-123
Timer/PWM . . . . . . . . . . 13-123
UART0, UART1, and UART2 . . . . . . . . . . 13-123
PIL . . . . . . . . . . 13-124
External mode . . . . . . . . . . 13-125

**Hardware Implementation Pane: BeagleBone Black** . . . . . **13-127**
Hardware Implementation Pane Overview . . . . . . . . . . 13-128
Board Parameters . . . . . . . . . . 13-128
Build Options . . . . . . . . . . 13-128
Clocking . . . . . . . . . . 13-129
Operating system options . . . . . . . . . . 13-129
External mode . . . . . . . . . . 13-130

**Hardware Implementation Pane: STMicroelectronics Discovery Boards** . . . . . . . . . . **13-131**
Hardware Implementation Pane Overview . . . . . . . . . . 13-132
Embedded Coder Support Package for STMicroelectronics Discovery Boards Hardware Settings . . . . . . . . . . 13-132
Operating system options . . . . . . . . . . 13-134
Scheduler options . . . . . . . . . . 13-135
Build options . . . . . . . . . . 13-137
Clocking . . . . . . . . . . 13-139
PIL . . . . . . . . . . 13-140
ADC Common . . . . . . . . . . 13-142
ADC 1, ADC 2, ADC 3 . . . . . . . . . . 13-144
GPIO A, GPIO B, GPIO C, GPIO D, GPIO E, GPIO F, GPIO G, GPIO H, GPIO I . . . . . . . . . . 13-146
External mode . . . . . . . . . . 13-147

**Hardware Implementation Pane** . . . . . . . . . . **13-0**
Hardware Implementation Pane Overview . . . . . . . . . . 13-148
Build options . . . . . . . . . . 13-149
Clocking . . . . . . . . . . 13-151
Ethernet . . . . . . . . . . 13-152
PIL . . . . . . . . . . 13-154
External mode . . . . . . . . . . 13-156

**Hardware Implementation Pane** . . . . . . . . . . **13-0**
Hardware Implementation Pane Overview . . . . . . . . . . 13-148
Build options . . . . . . . . . . 13-159

Clocking . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-160**
I2C . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-161**
PIL . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-162**
SPI . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-163**
External mode . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-164**

**Hardware Implementation Pane: Texas Instruments C2000** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-165**
Hardware Implementation Pane Overview . . . . . . . . . . . . **13-167**
Texas Instruments C2000 Settings . . . . . . . . . . . . . . . . . **13-167**
C28x-Scheduler options . . . . . . . . . . . . . . . . . . . . . . . . **13-169**
C28x-Build options . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-171**
C28x-Clocking . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-175**
C28x-ADC . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-178**
C28x-DAC . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-181**
C28-COMP . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-182**
C28x-eCAN_A, C28x-eCAN_B . . . . . . . . . . . . . . . . . . . **13-183**
C28x-eCAP . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-186**
C28x-ePWM . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-189**
C28x-I2C . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-192**
C28x-SCI_A, C28x-SCI_B, C28x-SCI_C . . . . . . . . . . . . . . **13-198**
C28x-SPI_A, C28x-SPI_B, C28x-SPI_C, C28x-SPI_D . . . . . . **13-201**
C28x-eQEP . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-204**
C28x-Watchdog . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-206**
C28x-GPIO . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-208**
C28x-Flash_loader . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-213**
C28x-DMA_ch[#] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-215**
C28x-LIN . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-224**
External mode . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-230**
Execution profiling . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-231**

**Hardware Implementation Pane: Texas Instruments Concerto** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-232**
Hardware Implementation Pane Overview . . . . . . . . . . . . **13-233**
M3x-Scheduler options . . . . . . . . . . . . . . . . . . . . . . . . . **13-233**
C28x / ARM Cortex-M3 - Build options . . . . . . . . . . . . . . **13-235**
M3x-Clocking . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-238**
M3x-GPIO A–D . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-240**
M3x-UART0–4 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-241**
M3x-Ethernet . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-243**
M3x-PIL . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-244**
External mode . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-246**
C28x-Clocking . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-247**
C28x-ADC . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13-250**

C28x-eCAP . . . . . 13-253
C28x-ePWM . . . . . 13-256
C28x-I2C . . . . . 13-259
C28x-SCI_A, C28x-SCI_B, C28x-SCI_C . . . . . 13-265
C28x-SPI_A, C28x-SPI_B, C28x-SPI_C, C28x-SPI_D . . . . . 13-268
C28x-eQEP . . . . . 13-271
C28x-GPIO . . . . . 13-273
C28x-DMA_ch[#] . . . . . 13-278
External mode . . . . . 13-287

**Hardware Implementation Pane: Xilinx Zynq ZC702/ZC706 Evaluation Kits, ZedBoard** . . . . . **13-289**
Hardware Implementation Pane Overview . . . . . 13-289
Operating system settings . . . . . 13-289
Clocking . . . . . 13-290
Build Options . . . . . 13-290
External mode . . . . . 13-291

**Recommended Settings Summary for Model Configuration Parameters** . . . . . **13-292**

# 14 Parameters for Creating Protected Models

**Create Protected Model** . . . . . **14-2**
Create Protected Model: Overview . . . . . 14-2
Open read-only view of model . . . . . 14-3
Simulate . . . . . 14-3
Use generated code . . . . . 14-4
Code interface . . . . . 14-5
Content type . . . . . 14-6
Create protected model in . . . . . 14-7
Create harness model for protected model . . . . . 14-7

# 15 Model Advisor Checks


**Embedded Coder Checks** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **15-2**
Embedded Coder Checks Overview . . . . . . . . . . . . . . . . . . . . **15-3**
Check for blocks not recommended for C/C++ production code deployment . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **15-3**
Identify lookup table blocks that generate expensive out-of-range checking code . . . . . . . . . . . . . . . . . . . . . . . . . . . . **15-4**
Check output types of logic blocks . . . . . . . . . . . . . . . . . . . . **15-6**
Check the hardware implementation . . . . . . . . . . . . . . . . . . **15-7**
Identify questionable software environment specifications . . **15-8**
Identify questionable code instrumentation (data I/O) . . . . . **15-10**
Check configuration parameters for MISRA C:2012 . . . . . . **15-11**
Check for blocks not recommended for MISRA C:2012 . . . . **15-14**
Check for unsupported block names . . . . . . . . . . . . . . . . . **15-16**
Check usage of Assignment blocks . . . . . . . . . . . . . . . . . . . **15-16**
Check for switch case expressions without a default case . . **15-18**
Check for missing error ports for AUTOSAR receiver interfaces . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **15-19**
Check bus object names that are used as element names . . **15-20**
Check configuration parameters for secure coding standards . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **15-20**
Check for blocks not recommended for secure coding standards . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **15-23**
Identify questionable subsystem settings . . . . . . . . . . . . . **15-24**
Identify blocks that generate expensive fixed-point and saturation code . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **15-25**
Check for missing const qualifiers in model functions . . . . . **15-27**
Identify questionable fixed-point operations . . . . . . . . . . . **15-28**
Identify blocks that generate expensive rounding code . . . . **15-31**
Check for bitwise operations on signed integers . . . . . . . . . **15-32**
Check for recursive function calls . . . . . . . . . . . . . . . . . . . **15-33**
Check for equality and inequality operations on floating-point values . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **15-33**
Check integer word length . . . . . . . . . . . . . . . . . . . . . . . . **15-34**
Check block names . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **15-35**

## 16 Tools in Embedded Coder—Alphabetical List

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

# Functions in Embedded Coder — Alphabetical List

# activate

Mark file, project, or build configuration as active

## Syntax

`activate(IDE_Obj,'objectname','type')`

## IDEs

This function supports the following IDEs:

- Analog Devices® VisualDSP++®
- Texas Instruments™ Code Composer Studio™ v3

## Description

Use the `activate(IDE_Obj,'objectname','type')` method to make a project file or build configuration active in the MATLAB® session.

When you make a project, file, or build configuration active, methods you invoke on the IDE handle object apply to that project, file, or build configuration.

## Input Arguments

### IDE_Obj

For *IDE_Obj*, enter the name of the IDE handle object you created using a constructor function.

### objectname

For *objectname*, enter the name of the project file or build configuration to make active.

For project files, enter the full file name including the extension.

For build configurations, enter `'Debug'`, `'Release'`, or `'Custom'`. Before using the `activate` method on a build configuration, activate the project that contains the build configuration. For more information about configurations, see "Configuration" on page 13-11.

**type**

For *type*, enter the type of object to make active. If you omit the *type* argument, *type* defaults to `'project'`. Enter one of the following character vectors for *type*:

- `'project'` — Makes a specified project active.
- `'buildcfg'` — Make a specified build configuration active

**IDE support for *type***

| | CCS | VisualDSP++ |
|---|---|---|
| 'project' | Yes | Yes |
| 'buildcfg' | Yes | Yes |

## Examples

After using a constructor to create the IDE handle object, h, open several projects, make the first one active, and build the project:

```
h.open('c:\temp\myproj1')
h.open('c:\temp\myproj2')
h.open('c:\temp\myproj3')
h.activate('c:\temp\myproj1', 'project')
h.build
```

After making a project active, make the `'debug'` configuration active:

```
h.activate('debug','buildcfg')
```

## See Also

build | new | remove

**Introduced in R2011a**

# activateConfigSet

**Class:** cgv.CGV
**Package:** cgv

Activate configuration set of model

## Syntax

*cgvObj*.activateConfigSet(configSetName)

## Description

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

## Examples

Before calling cgv.CGV.run on a cgv.CGV object for a model, the model must already contain the named configuration set. After creating the cgv.CGV object for a model, you can use cgv.CGV.activateConfigSet to activate a configuration set in the model when the cgv.CGV object simulates the model.

```
configObj = Simulink.ConfigSet;
attachConfigSet('rtwdemo_cgv', configObj);
cgvObj = cgv.CGV('rtwdemo_cgv');
cgvObj.activateConfigSet(configObj.Name);
```

## See Also

### Topics

“About Model Configurations” (Simulink)
“Programmatic Code Generation Verification”

# add

Add files to active project in IDE

## Syntax

`add(IDE_Obj,filename,filetype)`

## IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

## Description

Use `add(IDE_Obj,filename,filetype)` to add an existing file to the active project in the IDE. Using the `add` function is equivalent to selecting **Project > Add Files to Project** in the IDE.

Before using `add`:

- Use the constructor function for your IDE to create an IDE handle object, such as *IDE_Obj*.
- Create or open a project using the `new` or `open` methods.
- Make the project active in the IDE using the `activate` method.

You can add file types your IDE supports to your project. Consult the documentation for your IDE for detailed information about supported file types.

**Supported File Types and Extensions**

| File Type | Extensions Supported | CCS IDE Project Folder |
|---|---|---|
| C/C++ source files | `.c`, `.cpp`, `.cc`, `.cxx`, `.sa`, `.h`, `.hpp`, `.hxx` | `Source` |
| Assembly source files | `.a*`, `.s*` (excluding `.sa`), `.dsp` | `Source` |
| Object and library files | `.o*`, `.lib`, `.doj`, `.dlb` | `Libraries` |
| Linker command file | `.cmd`, `.ldf` | `Project Name` |
| VDK support file | `.vdk` | Not applicable |
| DSP/BIOS file (only with CCS IDE) | `.tcf` | `DSP/BIOS Config` |

**Note** CCS IDE drops files in the project folder, indicated in the right-most column of the preceding table.

# Input Arguments

`add` places the file specified by *`filename`* in the active project in the IDE.

### IDE_Obj

*`IDE_Obj`* is a handle for an instance of the IDE. Before using a method, the constructor function for your IDE to create *`IDE_Obj`*.

### filename

*`filename`* is the name of the file to add to the active IDE project.

If you supply a filename without a path or relative path, the code generator searches the IDE working folder first. It then searches the folders on your MATLAB path. Add supported file types shown in the preceding table.

### filetype

*`filetype`* is an optional argument that specifies the file type. For example, `'lib'`, `'src'`, `'header'`.

## Examples

Start by creating an IDE handle object, such as `IDE_Obj` using the constructor for your IDE. Then enter the following commands:

```
new(IDE_Obj,'myproject','project');  % Create a new project.
```

```
add(IDE_Obj,'sourcefile.c'); % Add a C source file.
```

## See Also

`activate` | `cd` | `new` | `open` | `remove`

**Introduced in R2011a**

# addAdditionalHeaderFile

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

## Syntax

`addAdditionalHeaderFile(hEntry,headerFile)`

## Description

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

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

## Examples

### Specify Additional Header and Source Files

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

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

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

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

## Input Arguments

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

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

Example: op_entry

**headerFile — Name of additional header file**
character vector

*headerFile* is a character vector that specifies an additional header file.

Example: 'all_additions.h'

## See Also

addAdditionalIncludePath | addAdditionalSourceFile | addAdditionalSourcePath

### Topics

"Specify Build Information for Replacement Code"
"Define Code Replacement Mappings"

**Introduced in R2007b**

# addAdditionalIncludePath

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

## Syntax

`addAdditionalIncludePath(hEntry,path)`

## Description

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

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

## Examples

### Specify Path to Additional Header and Source Files

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

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

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

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

## Input Arguments

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

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

Example: `op_entry`

**`path` — Path to an additional header file**
character vector

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

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

## See Also

`addAdditionalHeaderFile` | `addAdditionalSourceFile` | `addAdditionalSourcePath`

### Topics

“Specify Build Information for Replacement Code”
“Define Code Replacement Mappings”

**Introduced in R2007b**

# addAdditionalLinkObj

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

## Syntax

`addAdditionalLinkObj(hEntry,linkObj)`

## Description

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

## Examples

### Specify an Additional Link Object

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

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

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

## Input Arguments

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

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

Example: op_entry

**linkObj — Name of an additional link object**
character vector

The *linkObj* is a character vector that specifies an additional link object.

Example: 'addition.o'

## See Also

addAdditionalLinkObjPath

### Topics

"Specify Build Information for Replacement Code"
"Define Code Replacement Mappings"

**Introduced in R2007b**

# addAdditionalLinkObjPath

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

## Syntax

`addAdditionalLinkObjPath(hEntry,path)`

## Description

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

## Examples

### Specify Path to Additional Link Object

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

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

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

## Input Arguments

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

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

Example: op_entry

**path — Path to an additional link object**
character vector

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

Example: op_entry

# See Also

addAdditionalLinkObj

## Topics

"Specify Build Information for Replacement Code"
"Define Code Replacement Mappings"

**Introduced in R2008a**

# addAdditionalSourceFile

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

## Syntax

`addAdditionalSourceFile(hEntry,sourceFile)`

## Description

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

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

## Examples

### Specify Additional Header and Source Files

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

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

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

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

## Input Arguments

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

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

Example: op_entry

**sourceFile — Name of an additional source file**
character vector

The *sourceFile* is a character vector specifying an additional source file.

Example: 'all_additions.c'

## See Also

addAdditionalHeaderFile | addAdditionalIncludePath | addAdditionalSourcePath

### Topics

"Specify Build Information for Replacement Code"
"Define Code Replacement Mappings"

**Introduced in R2007b**

# addAdditionalSourcePath

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

## Syntax

`addAdditionalSourcePath(hEntry,path)`

## Description

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

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

## Examples

### Specify Path to Additional Header and Source Files

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

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

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

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

## Input Arguments

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

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

Example: `op_entry`

**`path` — Path to an additional source file**
character vector

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

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

## See Also

`addAdditionalHeaderFile` | `addAdditionalIncludePath` | `addAdditionalSourceFile`

### Topics

"Specify Build Information for Replacement Code"
"Define Code Replacement Mappings"

**Introduced in R2007b**

# addAlgorithmProperty

Add algorithm properties for code replacement table entry

## Syntax

`addAlgorithmProperty(hEntry, name-value)`

## Arguments

*hEntry*

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

*name-value*

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

| Name | Values |
| --- | --- |
| 'ExtrapMethod' | 'Clip' \| 'Linear' |
| 'IndexSearchMethod' | 'Evenly spaced points' \| 'Linear search' \| 'Binary search' |
| 'InterpMethod' | 'Flat' \| 'Linear' \| 'Cubic spline' |
| 'IndexSearchMethod' | 'Evenly spaced points' \| 'Linear search' \| 'Binary search' |
| 'NumberOfTableDimensions' | '1' \| '2' \| '3' \| '4' \|\| '5' |
| 'RemoveProectionInput' | 'off' \| 'on' |
| 'RndMeth' | 'Ceiling' \| 'Convergent' \| 'Floor' \| 'Nearest' \| 'Round' \| 'Simplest' \| 'Zero' |

| Name | Values |
| --- | --- |
| 'SaturateOnIntegerOverflow' | 'off' \| 'on' |
| 'UseLastBreakpoint' | 'off' \| 'on' |
| 'UseLastTableValue' | 'off' \| 'on' |
| 'ValidIndexMayReachLast' | 'off' \| 'on' |

## Description

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

## Examples

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

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

```
hLib = RTW.TflTable;

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

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

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

```
hEnt.addConceptualArg(arg);

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

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

hEnt.addAlgorithmProperty('ExtrapMethod', 'Clip');
hEnt.addAlgorithmProperty('InterpMethod', Linear');
hEnt.addAlgorithmProperty('IndexSearchMethod', {'Linear search', ...
                                                 'Binary search'});
```

# See Also

`getTflArgFromString`

## Topics

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

**Introduced in R2014b**

# addArgConf

**Class:** RTW.ModelSpecificCPrototype
**Package:** RTW

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

## Syntax

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

## Description

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

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

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

## Input Arguments

| | |
|---|---|
| *`obj`* | Handle to a model-specific C prototype function control object previously returned by *`obj`*` = RTW.ModelSpecificCPrototype` or *`obj`*` = RTW.getFunctionSpecification(`*`modelName`*`)`. |
| *`portName`* | Character vector specifying the unqualified name of an inport or outport in your Simulink model. |

| | |
|---|---|
| *category* | Character vector specifying the argument category, either `'Value'` or `'Pointer'`. |
| *argName* | Character vector specifying a valid C identifier. |
| *qualifier* | Character vector specifying the argument type qualifier: `'none'`, `'const'`, `'const *'`, or `'const * const'`. |

## Examples

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

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

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

%% Add argument configuration information for Input and Output ports
addArgConf(a,'Input','Pointer','inputArg','const *')
addArgConf(a,'Output','Pointer','outputArg','none')

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

## Alternatives

You can specify the argument configuration information in the Model Interface dialog box. See “Customize Generated C Function Interfaces”.

## See Also

`RTW.ModelSpecificCPrototype.attachToModel`

### Topics

“Customize Generated C Function Interfaces”

# addBaseline

**Class:** cgv.CGV
**Package:** cgv

Add baseline file for comparison

## Syntax

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

## Description

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

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

## Input Arguments

**`inputName`**

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

**`baselineFile`**

A MAT-file containing baseline data

**`toleranceFile`**

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

## Examples

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

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

## See Also

`cgv.CGV.addInputData` | `cgv.CGV.createToleranceFile` | `cgv.CGV.getStatus` | `cgv.CGV.run`

### Topics

"Verify Numerical Equivalence with CGV"

# addHeaderReportFcn

**Class:** cgv.CGV
**Package:** cgv

Add callback function to execute before executing input data in object

## Syntax

*cgvObj*.addHeaderReportFcn(CallbackFcn)

## Description

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

```
CallbackFcn(cgvObj)
```

## Examples

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

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

where *HeaderReportFcn* is defined as:

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

## See Also

cgv.CGV.run

## Topics

"Callbacks for Customized Model Behavior" (Simulink)

# addPostExecFcn

**Class:** cgv.CGV
**Package:** cgv

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

## Syntax

*cgvObj*.addPostExecFcn(CallbackFcn)

## Description

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

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

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

## Examples

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

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

where *PostExecutionFcn* is defined as:

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

## See Also

cgv.CGV.run

## Topics

"Callbacks for Customized Model Behavior" (Simulink)

# addPostExecReportFcn

**Class:** cgv.CGV
**Package:** cgv

Add callback function to execute after each input data file executes

## Syntax

*cgvObj*.addPostExecReportFcn(CallbackFcn)

## Description

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

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

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

## Examples

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

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

where *PostExecutionReportFcn* is defined as:

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

## See Also

cgv.CGV.run

## Topics

“Callbacks for Customized Model Behavior” (Simulink)

# addPreExecFcn

**Class:** cgv.CGV
**Package:** cgv

Add callback function to execute before each input data file executes

## Syntax

*cgvObj*.addPreExecFcn(CallbackFcn)

## Description

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

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

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

## Examples

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

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

where *PreExecutionFcn* is defined as:

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

## See Also

cgv.CGV.run

## Topics

“Callbacks for Customized Model Behavior” (Simulink)

# addPreExecReportFcn

**Class:** cgv.CGV
**Package:** cgv

Add callback function to execute before each input data file executes

## Syntax

*cgvObj*.addPreExecReportFcn(CallbackFcn)

## Description

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

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

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

## Examples

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

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

where *PreExecutionReportFcn* is defined as:

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

## See Also

cgv.CGV.run

## Topics

“Callbacks for Customized Model Behavior” (Simulink)

# addTrailerReportFcn

**Class:** cgv.CGV
**Package:** cgv

Add callback function to execute after the input data executes

## Syntax

*cgvObj*.addTrailerReportFcn(CallbackFcn)

## Description

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

```
CallbackFcn(cgvObj)
```

## Examples

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

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

where *TrailerReportFcn* is defined as:

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

## See Also

cgv.CGV.run

## Topics

"Callbacks for Customized Model Behavior" (Simulink)

# addCheck

**Class:** rtw.codegenObjectives.Objective
**Package:** rtw.codegenObjectives

Add checks

## Syntax

`addCheck(obj, checkID)`

## Description

`addCheck(obj, checkID)` includes the check, *checkID*, in the Code Generation Advisor. When a user selects the objective, the Code Generation Advisor includes the check, unless another objective with a higher priority excludes the check.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a code generation objective object previously created. |
| *checkID* | Unique identifier of the check that you add to the new objective. |

## Examples

Add the **Identify questionable code instrumentation (data I/O)** check to the objective.

```
addCheck(obj, 'mathworks.codegen.CodeInstrumentation');
```

## See Also

`Simulink.ModelAdvisor`

## Topics

"Create Custom Code Generation Objectives"
"About IDs" (Simulink)

# addComplexTypeAlignment

Specify alignment boundary of a complex type

## Syntax

`addComplexTypeAlignment(hDataAlign,baseType,alignment)`

## Description

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

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

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

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

## Examples

### Specify Alignment Boundary for Complex Types

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

```
function rtwTargetInfo(cm)
% rtwTargetInfo function to register a code replacement library (CRL)
```

```
% for use with  code generation

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

end % End of RTWTARGETINFO

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

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

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

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

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

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

end % End of LOCCRLREGFCN
```

# Input Arguments

### `hDataAlign` — Handle to a data alignment object

handle

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

Example: `da`

### `baseType` — Specifies a built-in data type

character vector

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

Example: `'int8'`

**`alignment` — Specifies the alignment boundary**
positive integer

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

Example: `16`

# See Also

## Topics

“Data Alignment for Code Replacement”
“Define Code Replacement Mappings”
“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”

**Introduced in R2014a**

# addConceptualArg

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

## Syntax

`addConceptualArg(hEntry,arg)`

## Description

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

## Examples

### Add Conceptual Arguments for Ports

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

```
hLib = RTW.TflTable;

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

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

```
op_entry.addConceptualArg( arg );

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

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

op_entry.copyConceptualArgsToImplementation();

hLib.addEntry( op_entry );
```

## Input Arguments

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

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

Example: `op_entry`

**`arg` — Argument added to the array of conceptual arguments**
character vector

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

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

## See Also

`getTflArgFromString`

### Topics

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

**Introduced in R2007b**

# addDWorkArg

Add DWork argument for semaphore entry in code replacement table

## Syntax

`addDWorkArg(hEntry,arg)`

## Description

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

## Examples

### Add a DWork Argument

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

```
hLib = RTW.TflTable;

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

% specify conceptual operands and result
```

```
arg = hLib.getTflArgFromString('y1', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
hEnt.addConceptualArg(arg);
arg = hLib.getTflArgFromString('u1', 'void');
hEnt.addConceptualArg(arg);

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

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

hLib.addEntry( hEnt );
```

## Input Arguments

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

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

Example: `sem_entry`

**`arg` — Argument added to the arguments for the table entry**
character vector

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

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

## See Also

`getTflDWorkFromString`

## Topics

"Semaphore and Mutex Function Replacement"
"Define Code Replacement Mappings"
"Code You Can Replace from MATLAB Code"
"Code You Can Replace From Simulink Models"

**Introduced in R2011b**

# addConfigSet

**Class:** cgv.CGV
**Package:** cgv

Add configuration set

## Syntax

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

## Description

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

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

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

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

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

## See Also

### Topics

"About Model Configurations" (Simulink)
"Programmatic Code Generation Verification"

# addEntry

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

## Syntax

`addEntry(hTable,entry)`

## Description

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

## Examples

### Add Operator Entry to Code Replacement Table

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

```
hLib = RTW.TflTable;

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

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

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

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

op_entry.copyConceptualArgsToImplementation();

addEntry(hLib, op_entry);
```

## Input Arguments

**`hTable` — Handle to a code replacement table**
handle

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

Example: `hLib`

**`entry` — Handle to a function or operator entry**
handle

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

Example: `op_entry`

## See Also

### Topics
“Define Code Replacement Mappings”
“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”

**Introduced in R2007b**

# addInputData

**Class:** cgv.CGV
**Package:** cgv

Add input data

## Syntax

`cgvObj.addInputData(inputName, inputDataFile)`

## Description

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

## Input Arguments

### inputName

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

### inputDataFile

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

## Tips

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

## See Also

`cgv.CGV.run`

### Topics

"Verify Numerical Equivalence with CGV"

# addParam

**Class:** rtw.codegenObjectives.Objective
**Package:** rtw.codegenObjectives

Add parameters

## Syntax

`addParam(obj, paramName, value)`

## Description

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

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a code generation objective object previously created. |
| *paramName* | Parameter that you add to the objective. |
| *value* | Value of the parameter. |

## Examples

Add `DefaultParameterBehavior` to the objective, and specify the parameter value as `Inlined`.

```
addParam(obj, 'DefaultParameterBehavior', 'Inlined');
```

## See Also

`get_param`

## Topics

"Create Custom Code Generation Objectives"

# addPostLoadFiles

**Class:** cgv.CGV
**Package:** cgv

Add files required by model

## Syntax

`cgvObj.addPostLoadfiles({FileList})`

## Description

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

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

## See Also

### Topics

“Verify Numerical Equivalence with CGV”
“Callbacks for Customized Model Behavior” (Simulink)

# address

Memory address and page value of symbol in IDE

## Syntax

*a* = `address(`*IDE_Obj*`,`*symbol*`,`*scope*`)`

## IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

## Description

The *a* = `address(`*IDE_Obj*`,`*symbol*`,`*scope*`)` method returns the memory address of the first matching symbol in the symbol table of the most recently loaded program.

Because the `address` method returns the `address` and `page` values as a structure, your programs can use the values directly. For example, the `read` and `write` can use `a` as an input.

If the address method does not find the symbol in the symbol table, it generates a warning and returns a null value.

## Input Arguments

### a

Use *a* as a variable to capture the return values from the `address` method.

**IDE_Obj**

*IDE_Obj* is a handle for an instance of the IDE. Before using a method, use the constructor function for your IDE to create *IDE_Obj*.

**symbol**

*symbol* is the name of the symbol for which you are getting the memory address and page values.

Symbol names are case sensitive.

For `address` to return an address, the symbol must be a valid entry in the symbol table. If the `address` method does not find the symbol, it generates a warning and leaves `a` empty.

**scope**

Optionally, you set the scope of the address method. Enter `'local'` or `'global'`. Use `'local'` when the current scope of the program is the desired function scope. If you omit the *scope* argument, the `address` method uses `'local'` by default.

# Output Arguments

If the `address` method does not find the symbol, it generates a warning and does not return a value for `a`.

The `address` method only returns address information for the first matching symbol in the symbol table.

## For Code Composer Studio

The return value, `a`, is a numeric array with the symbol's address offset, `a(1)`, and page, `a(2)`.

With TI C6000™ processors, the memory page value is 0.

## For VisualDSP++

With VisualDSP++, `address` requires a linker command file (lcf) in your project.

The return value a is a numeric array with the symbol's start address, a(1), and memory type, a(2).

## Examples

After you load a program to your processor, address lets you read and write to specific entries in the symbol table for the program. For example, the following function reads the value of symbol '*ddat*' from the symbol table in the IDE.

```
ddatv = read(IDE_Obj,address(IDE_Obj,'ddat'),'double',4)
```

*ddat* is an entry in the current symbol table. address searches for the string *ddat* and returns a value when it finds a match. read returns *ddat* to MATLAB software as a double-precision value as specified by the string 'double'.

To change values in the symbol table, use address with write:

```
write(IDE_Obj,address(IDE_Obj,'ddat'),double([pi 12.3 exp(-1)...
sin(pi/4)]))
```

After executing this write operation, *ddat* contains double-precision values for π, 12.3, $e^{-1}$, and sin(π/4). Use read to verify the contents of *ddat*:

```
ddatv = read(IDE_Obj,address(IDE_Obj,'ddat'),'double',4)
```

MATLAB software returns

```
ddatv  =

    3.1416   12.3   0.3679   0.7071
```

## See Also

load | read | symbol | write

**Introduced in R2011a**

# adivdsp

Create handle object to interact with VisualDSP++ IDE

## Syntax

```
IDE_Obj = adivdsp
IDE_Obj = adivdsp('propname1',propvalue1,'propname2',propvalue2,…
,'timeout',value)
IDE_Obj = adivdsp('my_session')
```

## IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++

## Description

If the IDE is not running, `IDE_Obj = adivdsp` opens the VisualDSP++ software for the most recent active session. After that, it creates an object, `IDE_Obj`, that references the newly opened session. If the IDE is running, `adivdsp` returns object `IDE_Obj` that connects to the active session in the IDE.

**Note** The output object name (left side argument) you provide for `adivdsp` cannot begin with an underscore, such as `_IDE_Obj`.

`adivdsp` creates an interface between MATLAB software and Analog Devices VisualDSP ++ software. The first time you use `adivdsp`, supply a session name as an input argument (refer to the next syntax).

`IDE_Obj = adivdsp('sessionname','name','procnum','number',...)` returns an object handle `IDE_Obj` that you use to interact with a processor in the IDE from MATLAB.

Use the debug methods with this object to access memory and control the execution of the processor.

The `adivdsp` function interprets input arguments as object property definitions. Each property definition consists of a property name followed by the desired property value (often called a *PV*, or *property name/property value*, pair). Although you can define a number of `adivdsp` object properties when you create the object, there are several important properties that you must provide during object construction. These properties must be delineated when you create the object. The required input arguments are as follows:

- `sessionname` — Specifies the session to connect to. This session must exist in the session list. `adivdsp` does not create new sessions. The resulting object refers to a processor in `sessionname`. To see the list of sessions, use `listsessions` at the MATLAB command prompt.
- `procnum`— Specifies the processor to connect to in `sessionname`. The adivdsp object only supports connecting to processor 0. As such, the default value for `procnum` is 0 for the first processor on the board. If you omit the `procnum` argument, `adivdsp` connects to the first processor.

After you build the `adivdsp` object `IDE_Obj`, you can review the object property values with `get`, but you cannot modify the `sessionname` and `procnum` property values.

To connect to the active session in IDE, omit the `sessionname` property in the syntax. If you do not pass `sessionname` as an input argument, the object defaults to the active session in the IDE.

Use `listsessions` to determine the number for the desired DSP processor. If your IDE session is single processor or to connect to processor zero, you can omit the `procnum` property definition. If you omit the *`procnum`* argument, *`procnum`* defaults to `0` (zero-based).

`IDE_Obj = adivdsp('propname1',propvalue1,'propname2',propvalue2,… ,'timeout',value)` sets the global time-out value to `value` in `IDE_Obj`. MATLAB waits for the specified time-out value to get a response from the IDE application. If the IDE does not respond within the allotted time-out period, MATLAB exits from the evaluation of this function.

If the session exists in the session list and the IDE is not already running, `IDE_Obj = adivdsp('my_session')` connects to `my_session`. In this case, MATLAB starts VisualDSP++ IDE for the session named `my_session`.

The following list shows some other possible cases and results of using `adivdsp` to construct an object that refers to `my_session`.

- If `my_session` does not exist in the session list and the IDE is not already running, MATLAB returns an error stating that `my_session` does not exist in the session list.
- When `my_session` is the current active session and the IDE is already running, MATLAB connects to the IDE for this session.
- If `my_session` is not the current active session, but exists in the session list, and the IDE is already running, MATLAB displays a dialog box asking if you want to switch to `my_session`. If you choose to switch to `my_session`, the existing handles you have to other sessions in the IDE become invalid. To connect to the other sessions you use `adivdsp` to recreate the objects for those sessions.
- If `my_session` does not exist in the session list and the IDE is already running, MATLAB returns an error, explaining that the session `my_session` does not exist in the session list.

## Examples

These examples show some of the operation of `adivdsp`.

```
IDE_Obj = adivdsp('sessionname','my_session','procnum',0);
```

returns a handle to the first DSP processor for session `my_session`.

`IDE_Obj = adivdsp` without input arguments constructs the object `IDE_Obj` with the default property values, returning a handle to the first DSP processor for the active session in the IDE.

`IDE_Obj = adivdsp('sessionname','my_session');` returns a handle to the first DSP processor for the session `my_session`.

## See Also

`listsessions`

**Introduced in R2011a**

# adivdspsetup

Configure the code generator to interact with VisualDSP++ IDE

## Syntax

`adivdspsetup`

## IDEs

This function supports the following IDEs:

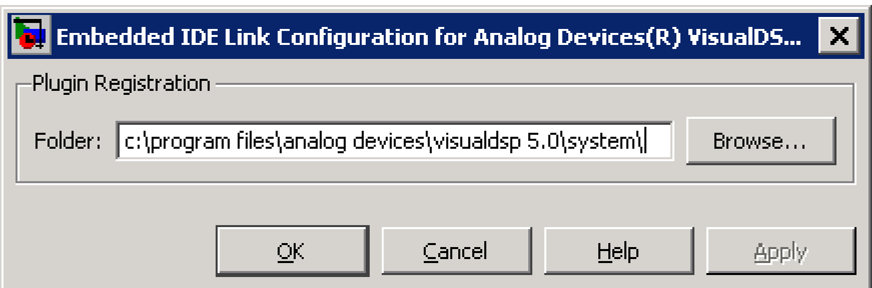
- Analog Devices VisualDSP++

## Description

Enter `adivdspsetup` at the MATLAB command line when you are setting up the code generator to interact with VisualDSP++ for the first time. This action displays a dialog box to specify where to install a plug-in for VisualDSP++. The default value for **Folder** is the VisualDSP++ `system` folder. You can specify folders for which you have write access. When you click **OK**, the software adds the plug-in to the folder and registers the plug-in with the VisualDSP++ IDE.

## Examples

1. At the MATLAB command line, enter: `adivdspsetup`. This action opens the following dialog box:




2 Click **Browse**, locate the `system` folder for VisualDSP++, and click **OK**. This action registers the MathWorks plugin to the VisualDSP++ IDE.

## See Also

`adivdsp`

**Introduced in R2011a**

# animate

Run application on processor to breakpoint

## Syntax

`animate(IDE_Obj)`

## IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

## Description

`animate(IDE_Obj)` starts the processor application, which runs until it encounters a breakpoint in the code. At the breakpoint, application execution halts and CCS Debugger returns data to the IDE to update the windows not connected to probe points. After updating the display, the application resumes execution and runs until it encounters another breakpoint. The run-break-resume process continues until you stop the application from MATLAB software with the `halt` function or from the IDE.

While running scripts or files in MATLAB software, you can use `animate` to update the IDE with information as your script or program runs.

### Using animate with Multiprocessor Boards

When you use `animate` with a `ticcs` object *IDE_Obj* that comprises more than one processor, such as an OMAP processor, the method applies to each processor in your *IDE_Obj* object. This action causes each processor to run a loaded program just as it does for the single processor case.

## See Also

`halt` | `restart` | `run`

**Introduced in R2011a**

# annotate

Color profiled model components or open model with profiled components colored

## Syntax

`annotate(executionProfile)`

## Description

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

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

## See Also

`report`

### Topics

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

**Introduced in R2016b**

# attachToModel

**Class:** RTW.ModelCPPClass
**Package:** RTW

Attach model-specific C++ class interface to loaded ERT-based Simulink model

## Syntax

`attachToModel(obj, modelName)`

## Description

`attachToModel(obj, modelName)` attaches a model-specific C++ class interface to a loaded ERT-based Simulink model.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a model-specific C++ class interface control object, such as a handle previously returned by `obj = RTW.ModelCPPArgsClass on page 1-534` or `obj = RTW.ModelCPPDefaultClass on page 1-540`. |
| *modelName* | Character vector specifying the name of a loaded ERT-based Simulink model to which the object is going to be attached. |

## Alternatives

The **Configure C++ Class Interface** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Configure C++ class interface dialog box, where you can flexibly control the C++ class interfaces that are generated for your model. Once you validate and apply your changes, you can generate code based on your C ++ class interface modifications. See “Customize C++ Class Interfaces Using Graphical Interfaces”.

## See Also

### Topics
“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

# attachToModel

**Class:** RTW.ModelSpecificCPrototype
**Package:** RTW

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

## Syntax

`attachToModel(obj, modelName)`

## Description

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

## Input Arguments

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

## Alternatives

Click the **Configure Model Functions** button on the **Code Generation > Interface** pane of the Configuration Parameters dialog box for flexible control over the model function prototypes that are generated for your model. Once you validate and apply your changes, you can generate code based on your function prototype modifications. See “Customize Generated C Function Interfaces”.

## See Also

### Topics

“Customize Generated C Function Interfaces”

# build

Build or rebuild current project

## Syntax

```
[result,numwarns] = build(IDE_Obj,timeout)
build(IDE_Obj,'all')
```

## IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

## Description

`[result,numwarns] = build(IDE_Obj,timeout)` incrementally builds the active project. Incremental builds recompile only source files in your project that you changed or added after the most recent build. `build` uses the file time stamp to determine whether to recompile a file. After recompiling the source files, `build` links the object files to make a new program file.

The value of `result` is 1 when the build process completes. The value of `numwarns` is the number of compilation warnings generated from the build process.

The *`timeout`* argument defines the number of seconds MATLAB waits for the IDE to complete the build process. If the IDE exceeds the timeout period, this method returns a timeout error immediately. The timeout error does not terminate the build process in the IDE. The IDE continues the build process. The timeout error indicates that the build process did not complete before the specified timeout period expired. If you omit the *`timeout`* argument, the `build` method uses a default value of `1000` seconds.

`build(IDE_Obj,'all')` rebuilds the files in the active project.

## See Also

`isrunning` | `open`

**Introduced in R2011a**

# ccsboardinfo

Information about boards and simulators known to IDE

## Syntax

```
ccsboardinfo
boards = ccsboardinfo
```

## IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

## Description

`ccsboardinfo` returns configuration information about each board and processor installed and recognized by CCS. When you issue the function, `ccsboardinfo` returns the following information about each board or simulator.

| Installed Board Configuration Data | Configuration Item Name | Description |
|---|---|---|
| Board number | `boardnum` | The number CCS assigns to the board or simulator. Board numbering starts at 0 for the first board. You also use `boardnum` when you create a link to the IDE. |
| Board name | `boardname` | The name assigned to the board or simulator. Usually, the name is the board model name, such as F28335 XDS100 USB Emulator. If you are using a simulator, the name tells you which processor the simulator matches, such as F28335 Device Simulator. If you renamed the board during setup, this item displays the board name. |

| Installed Board Configuration Data | Configuration Item Name | Description |
|---|---|---|
| Processor number | `procnum` | The number assigned by CCS to the processor on the board or simulator. When the board contains more than one processor, CCS assigns a number to each processor, numbering from 0 for the first processor on the first board. For example, when you have two boards, the first processor on the first board is `procnum` = 0, and the first and second processors on the second board are `procnum` = 1 and `procnum` = 2. You also use this property when you create a link to the IDE. |
| Processor name | `procname` | Provides the name of the processor. Usually the name is CPU, unless you assign a different name. |
| Processor type | `proctype` | Gives the processor model, such as TMS320C2800 for the C28x series processors. |

Each row in the table that you see displayed represents one digital signal processor, either on a board or simulator. As a consequence, you use the information in the table in the function `ticcs` to identify a selected board in your PC.

`boards = ccsboardinfo` returns the configuration information about your installed boards in a slightly different manner. Rather return the table of the information, the method returns a list of board names and numbers. In that list, each board has an structure named `proc` that contains processor information. For example

```
boards = ccsboardinfo
```

returns

```
boards =

    number: 0
      name: 'F28335 Device Simulator'
      proc: [1x1 struct]
```

where the structure `proc` contains the processor information for the F28335 device simulator:

```
boards.proc
```

```
ans =

    number: 0
      name: 'CPU'
      type: 'TMS320F283xx'
```

Reviewing the output from both function syntaxes shows that the configuration information is the same.

To connect with a specific board when you create an IDE handle object, combine this syntax with the dot notation for accessing elements in a structure. Use the `boardnum` and `procnum` properties in the `boards` structure. For example, when you enter

```
boards = ccsboardinfo;
```

`boards(1).name` returns the name of your second installed board and `boards(1).proc(2).name` returns the name of the second processor on the second board. To create a link to the second processor on the second board, use

```
IDE_Obj = ticcs('boardnum',boards(1).number,'procnum',...
boards(1).proc(2).name);
```

## Examples

On a PC with both F28335 simulator and F28027 with XDS100 USB emulator are installed,

```
ccsboardinfo
```

returns something like the following table. Your display may differ slightly based on what you called your boards when you configured them in CCS Setup Utility:

```
Board Board                                           Proc     Processor                           Processor
 Num  Name                                            Num      Name                                Type
 ---      ---------------------------------               ---      ---------------------------------      -
 0        F28027 XDS100 USB Emulator     0        TMS320C2800_0               TMS320C2800
 1        F28335 Device Simulator             0         CPU                                 TMS320F283xx
```

## See Also

info | ticcs

**Introduced in R2011a**

# cd

Set working folder in IDE

## Syntax

```
wd = cd(IDE_Obj)
cd(IDE_Obj,folder)
```

## IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

## Description

`wd = cd(IDE_Obj)` assigns the IDE working folder to the variable, `wd`. which you reference via the IDE handle object, `IDE_Obj`.

`cd(IDE_Obj,folder)` sets the IDE working folder to `'folder'`. `'folder'` can be a path relative to your working folder, or an absolute path. The intended folder must exist. `cd` does not create a folder. Setting the IDE folder does not change your MATLAB Current Folder.

`cd` alters the default folder for `open` and `load`. Loading a new workspace file also changes the working folder for the IDE.

## See Also

`dir` | `load` | `open`

**Introduced in R2011a**

# cgv.CGV class

**Package:** cgv

Verify numerical equivalence of results

## Description

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

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

## Construction

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

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

### Input Arguments

**`model_name`**

Name of the model that you are verifying.

Optional comma-separated pairs of `Name,Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`).

You can specify several name-value pair arguments in a variety of orders, such as `Name1,Value1,…,NameN,ValueN`.

**`ComponentType`**

Define the SIL or PIL approach

| Value | Description |
| --- | --- |
| `topmodel` (default) | Top-model SIL or PIL simulation and standalone code interface mode. |
| `modelblock` | Model block SIL or PIL simulation and model reference target code interface mode. |

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

**Default:** topmodel

**`Connectivity`**

Specify mode of execution

| Value | Description |
| --- | --- |
| `sim` or `normal` (default) | Mode of execution is Normal simulation. |
| `sil` | Mode of execution is SIL. |
| `pil` | Mode of execution is PIL. |

## Properties

**`Description`**

Specify a description of the object.

**Default:** `' '` (null character vector)

**`Name`**

Specify a name for the object.

**Default:** `' '` (null character vector)

## Methods

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

## Copy Semantics

Handle. To learn how handle classes change copy operations, see Copying Objects (MATLAB) in the MATLAB Programming Fundamentals documentation.

## Examples

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

1. Create a `cgv.CGV` object, `cgvObj`, for each mode of execution and use the `cgv.CGV` set up methods to configure the model for each execution. The set up methods are:
   - `addInputData`
   - `addPostLoadFiles`
   - `setOutputDir`
   - `setOutputFile`
   - `addCallBack`
   - `addConfigSet`
2. Run the model for each mode of execution using the `cgvObj.run` method.
3. Use the `cgv.CGV` access methods to get and evaluate the data. The access methods are:
   - `getOutputData`
   - `getSavedSignals`
   - `plot`
   - `compare`

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

---

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

---

## See Also

`cgv.Config`

### Topics

“Verify Numerical Equivalence with CGV”
“Using Code Generation Verification API”

# cgv.Config class

**Package:** cgv

Check and modify model configuration parameter values

## Description

Creates a handle to a `cgv.Config` object that supports checking and optionally modifying models for compatibility with various modes of execution that use generated code, such as, Software-In-the-Loop (SIL) or Processor-In-the-Loop (PIL).

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

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

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

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

## Construction

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

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

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

## Properties

### CheckOutports

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

| Value | Description |
| --- | --- |
| `on` (default) | Compile the model and check the model outports configuration |
| `off` | Do not compile the model or check the model outports configuration |

### ComponentType

Define the SIL or PIL approach

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

| Value | Description |
| --- | --- |
| `topmodel` (default) | Top-model SIL or PIL simulation and standalone code interface mode. |
| `modelblock` | Model block SIL or PIL simulation and model reference target code interface mode. |

### Connectivity

Specify mode of execution

| Value | Description |
|---|---|
| sim (default) | Mode of execution is simulation. Recommends changes to a subset of the configuration parameters that SIL and PIL targets require. |
| sil | Mode of execution is SIL. Requires that the system target file is set to 'ert.tlc' and that you do not use your own external target. Recommends changes to the configuration parameters that SIL targets require. |
| pil | Mode of execution is PIL with custom connectivity that you provide using the PIL Connectivity API. Recommends changes to the configuration parameters that PIL targets with custom connectivity require. |

### LogMode

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

| Value | Description |
|---|---|
| SignalLogging | Log signal data to a MATLAB workspace variable during execution.<br><br>This parameter selects the **Data Import/ Export** > **Signal logging** parameter in the Configuration Parameters dialog box. |

| Value | Description |
| --- | --- |
| `SaveOutput` | Save output data to a MATLAB workspace variable during execution.<br><br>This parameter selects **Data Import/ Export** > **Output** parameter in the Configuration Parameters dialog box.<br><br>The **Output** parameter does not save bus outputs. |

**`ReportOnly`**

The `ReportOnly` property specifies whether `cgv.Config` modifies the recommended values of the configuration parameters of the model.

If you set `ReportOnly` to `on`, `SaveModel` must be `off`.

| Value | Description |
| --- | --- |
| `off` (default) | `cgv.Config` automatically modifies the configuration parameter values that it recommends for the model. |
| `on` | `cgv.Config` does not modify the configuration parameter values that it recommends for the model. |

**`SaveModel`**

Specify whether to save the model with the configuration parameter values recommended by `cgv.Config`.

If you set `SaveModel` to `'on'`, `ReportOnly` must be `'off'`.

| Value | Description |
| --- | --- |
| `off` (default) | Do not save the model. |
| `on` | Save the model in the working folder. |

## Methods

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

## Copy Semantics

Handle. To learn how handle classes change copy operations, see Copying Objects (MATLAB) in the MATLAB Programming Fundamentals documentation.

## Examples

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

```
% Create a cgv.Config object and configure the model for top-model SIL.
cgvCfg = cgv.Config('rtwdemo_iec61508', 'LogMode', 'SaveOutput', ...
    'connectivity', 'sil');
cgvCfg.configModel();
% Display the results of what the cgv.Config object changed.
cgvCfg.displayReport();
% Close the rtwdemo_iec61508 model.
bdclose('rtwdemo_iec61508');
```

## See Also

`cgv.CGV`

### Topics

"Programmatic Code Generation Verification"

# coder.dataAlignment

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

## Syntax

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

## Description

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

## Examples

### Data Alignment for Imported Data

An example function that specifies data alignment for imported data.

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

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

y = x1 + x2;

end
```

### Data Alignment for Exported Data

An example function that specifies data alignment for exported data.

```
function a = exportedDataExampleFun(b)

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

a = b + z;

end
```

- "Data Alignment for Code Replacement"
- "Define Code Replacement Mappings"
- "What Is Code Replacement Customization?"
- "What Is Code Replacement?"

## Input Arguments

**'varName' — Variable name**
character array

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

**align_value — Data alignment boundary value**
integer

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

## Limitations

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

- Only use `coder.dataAlignment` to specify alignment information for function inputs, outputs, and global variables.

- `coder.dataAlignment` supports only matrix types, including matrix of complex types.
- For exported custom storage classes (CSCs), `coder.dataAlignment` supports only `ExportedGlobal`. You can specify alignment information for any imported CSCs.
- The code generator ignores `coder.dataAlignment` for non-ERT or non-ERT derived system target files.
- Global variables tagged using the `coder.dataAlignment` directive from within a MATLAB function block are ignored. Set the alignment value on the corresponding Data Store Memory.

# See Also

`codegen`

## Topics

"Data Alignment for Code Replacement"
"Define Code Replacement Mappings"
"What Is Code Replacement Customization?"
"What Is Code Replacement?"

**Introduced in R2017a**

# coder.replace

Replace current MATLAB function implementation with code replacement library function in generated code

## Syntax

`coder.replace(ifNoReplacement)`

## Description

`coder.replace(ifNoReplacement)` replaces the current function implementation with a code replacement library function.

If a match is not found in the code replacement library, code is generated without a replacement for the current function. `coder.replace` is a code generation function. It does not alter MATLAB code or MEX function generation.

During code generation, if you include `coder.replace` in a MATLAB function, `fcn`, it performs a code replacement library lookup for the following function signature:

```
[y1_type, y2_type,..., yn_type]=fcn(x1_type, x2_type,...,xn_type)
```

`y1_type, y2_type,..., yn_type` are the data types of the outputs of MATLAB function `fcn`. `x1_type, x2_type,...,xn_type` are the data types of the inputs of `fcn`. `coder.replace` derives the output types of the function based on the implementation in the MATLAB function. At code generation, the contents of `fcn` are discarded and replaced with a function call that is registered in the code replacement library as a replacement for `fcn`.

## Examples

#### Replace a MATLAB Function with Custom Code

Replace a MATLAB function with a custom implementation that is registered in the code replacement library.

Write a MATLAB function, `calculate`, that you want to replace with a custom implementation, `replacement_calculate_impl.c`, in the generated code.

```
function y = calculate(x)
% Search in the code replacement library for replacement
% and use replacement function if available
% Error if not found
  coder.replace('-errorifnoreplacement');
  y = sqrt(x);
end
```

Write a MATLAB function, `top_function`, that calls `calculate`.

```
function out = top_function(in)
  p = calculate(in);
  out = exp(p);
end
```

Create a file named `crl_table_calculate.m` that describes the function entries for a code replacement table. The replacement function `replacement_calculate_impl.c` and header file `replacement_calculate_impl.h` must be on the path.

```
hLib = RTW.TflTable;

%---------- entry: calculate -----------
hEnt = RTW.TflCFunctionEntry;
setTflCFunctionEntryParameters(hEnt, ...
  'Key', 'calculate', ...
  'Priority', 100, ...
  'ArrayLayout', 'ROW_MAJOR', ...
  'ImplementationName', ...
  'replacement_calculate_impl', ...
  'ImplementationHeaderFile', ...
  'replacement_calculate_impl.h', ...
  'ImplementationSourceFile', ...
  'replacement_calculate_impl.c')
% Conceptual Args

arg = getTflArgFromString(hEnt, 'y1','double');
```

```
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEnt, arg);

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

% Implementation Args

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

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

addEntry(hLib, hEnt);
```

Create an `rtwTargetInfo` file:

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

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

end % End of RTWTARGETINFO

function thisCrl = locCrlRegFcn

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

  % Define the CRL properties
  thisCrl.Name = 'My calculate Example';
  thisCrl.Description = 'Demonstration of function replacement';
  thisCrl.TableList = {'crl_table_calculate'};
  thisCrl.BaseTfl = 'C89/C90 (ANSI)';
  thisCrl.TargetHWDeviceType = {'*'};

end % End of LOCCRLREGFCN
```

Refresh registration information. At the MATLAB command line, enter:

```
RTW.TargetRegistry.getInstance('reset');
```

Because the data type of `x` and `y` is `double`, `coder.replace` searches for `double = calculate(double)` in the Code Replacement Library. If it finds a match, `codegen` generates the following code:

```
real_T top_function(real_T in)
{
  real_T p;
  p = replacement_calculate_impl(in);
  return exp(p);
}
```

In the generated code, the replacement function `replacement_calculate_impl` replaces the MATLAB function `calculate`.

- "Replace MATLAB Functions with Custom Code Using coder.replace"
- "Replace MATLAB Functions Specified in MATLAB Function Blocks"
- "Define Code Replacement Mappings"
- "What Is Code Replacement Customization?"
- "What Is Code Replacement?"

## Input Arguments

**`ifNoReplacement` — Selects whether the code generator produces a warning or error when no match is found**
`'-errorifnoreplacement'` | `'-warnifnoreplacement'`

The *`ifNoReplacement`* argument selects whether the code generator issues an error or warning when no match is found. If this argument is omitted, the code generator does not issue an error or warning.

`coder.replace('-errorifnoreplacement')` replaces the current function implementation with a code replacement library function. If a match is not found, code generation stops. An error message describing the code replacement library lookup failure is generated.

`coder.replace('-warnifnoreplacement')` replaces the current function implementation with a code replacement library function. If match is not found, code is

generated for the current function. A warning describing the code replacement library lookup failure is generated during code generation.

Example: `coder.replace()`

## Tips

- `coder.replace` is a code generation function. It does not alter MATLAB code or MEX function generation.
- Do not use multiple `coder.replace` statements inside a function.
- You cannot use `coder.replace` within conditional expressions and loops.
- `coder.replace` does not support replacements that require data alignment.
- `varargout` is not supported.
- You cannot use `coder.replace` to replace MATLAB functions that have variable-size inputs.
- `coder.replace` requires an Embedded Coder license.
- `coder.replace` disregards saturation and rounding modes when looking up function replacements in a code replacement library.

## See Also

`codegen`

### Topics

“Replace MATLAB Functions with Custom Code Using coder.replace”
“Replace MATLAB Functions Specified in MATLAB Function Blocks”
“Define Code Replacement Mappings”
“What Is Code Replacement Customization?”
“What Is Code Replacement?”

**Introduced in R2012b**

# coder.setupMISRAConfig

Configure code generation parameters to increase code compliance with MISRA C:2012 guidelines

## Syntax

```
coder.setupMISRAConfig(cfg)
```

## Description

`coder.setupMISRAConfig(cfg)` sets up an Embedded Coder code generation configuration object to increase the likelihood of generating code that complies with MISRA C®:2012 guidelines.

## Examples

### Configure Code Generation Parameters for Increased MISRA C Compliance

Create an Embedded Coder code generation configuration object.

```
cfg = coder.config('lib', 'ecoder', true);
```

Set properties that might impact MISRA C compliance.

```
coder.setupMISRAConfig(cfg);
```

`coder.setupMISRAConfig` sets property values according to the values in this table.

| Embedded CoderConfiguration Object Property | Value for Increased MISRA C Compliance |
|---|---|
| `CastingMode` | `'Standards'` |
| `DataTypeReplacement` | `'CoderTypedefs'` |

| Embedded CoderConfiguration Object Property | Value for Increased MISRA C Compliance |
|---|---|
| DynamicMemoryAllocation | 'Off' |
| EnableRuntimeRecursion | false |
| EnableSignedLeftShifts | false |
| EnableSignedRightShifts | false |
| GenerateDefaultInSwitch | true |
| ParenthesesLevel | 'Maximum' |
| TargetLangStandard | 'C99 (ISO)' for C, 'C++03 (ISO)' for C++ |

## Input Arguments

**cfg — Embedded Coder code generation configuration object**
coder.EmbeddedCodeConfig object

Embedded Coder configuration object for generating C/C++ code from MATLAB code. Create the object by using `coder.config`.

Example: cfg = coder.config('lib', 'ecoder',true)

## See Also

### Topics
"Increase Likelihood of Generating MISRA C Compliant Code from MATLAB Code"

### External Websites
www.misra.org.uk

**Introduced in R2017b**

# coder.storageClass

Assign storage class to global variable

## Syntax

```
coder.storageClass(global_name, storage_class)
```

## Description

`coder.storageClass(global_name, storage_class)` assigns the storage class `storage_class` to the global variable `global_name`.

Assign the storage class to a global variable in a function that declares the global variable. You do not have to assign the storage class in more than one function.

You must have an Embedded Coder license to use `coder.storageClass`. Only when you use an Embedded Coder project or configuration object for generation of C/C++ libraries or executables does the code generation software recognize `coder.storageClass` calls.

## Examples

### Export Global Variables

In the function `addglobals_ex`, assign the `'ExportedGlobal'` storage class to the global variable `myglobalone` and the `'ExportedDefine'` storage class to the global variable `myglobaltwo`.

```
function y = addglobals_ex(x)

% Define the global variables.

global myglobalone;
global myglobaltwo;
```

```
% Assign the storage classes.

coder.storageClass('myglobalone','ExportedGlobal');
coder.storageClass('myglobaltwo','ExportedDefine');
y = myglobalone + myglobaltwo + x;
end
```

Create a code configuration object for a library or executable.

```
cfg = coder.config('dll','ecoder', true);
```

Generate code. This example uses the `-globals` argument to specify the types and initial values of `myglobalone` and `myglobaltwo`. Alternatively, you can define global variables in the MATLAB global workspace. To specify the type of the input argument `x`, use the `-args` option.

```
codegen -config cfg -globals {'myglobalone', 1, 'myglobaltwo', 2} -args {1} addglobals_ex -report
```

From the initial values of `1` and `2`, `codegen` determines that `myglobalone` and `myglobaltwo` have the type `double`. `codegen` defines and declares the exported variables `myglobalone` and `myglobaltwo`. It generates code that initializes `myglobalone` to `1.0` and `myglobaltwo` to `2.0`.

To view the generated code for `myglobaltwo` and `myglobalone`, click the `View report` link.

- `myglobaltwo` is defined in the `Exported data define` section in `addglobals_ex.h`.

  ```
  /* Exported data define */

  /* Definition for custom storage class: ExportedDefine */
  #define myglobaltwo                    2.0
  ```
- `myglobalone` is defined in the `Variable Definitions` section in `addglobals_ex.c`.

  ```
  /* Variable Definitions */
  /* Definition for custom storage class: ExportedGlobal */
  double myglobalone;
  ```
- `myglobalone` is declared as `extern` in the `Variable Declarations` section in `addglobals_ex.h`.

```
/* Variable Declarations */
/* Declaration for custom storage class: ExportedGlobal */
extern double myglobalone;
```

- `myglobalone` is initialized in `addglobals_ex_initialize.c`.

```
#include "rt_nonfinite.h"
#include "addglobals_ex.h"
#include "addglobals_ex_initialize.h"

/* Named Constants */
#define b_myglobalone                     (1.0)

/* Function Definitions */

/*
 * Arguments    : void
 * Return Type  : void
 */
void addglobals_ex_initialize(void)
{
  rt_InitInfAndNaN(8U);
  myglobalone = b_myglobalone;
}
```

#### Import Global Variable

In the function `addglobal_im`, assign the `'ImportedExtern'` storage class to the global variable `myglobal`.

```
function y = addglobal_im(x)

% Define the global variable.

global myglobal;

% Assign the storage classes.

coder.storageClass('myglobal','ImportedExtern');
y = myglobal + x;
end
```

Create a file `c:\myfiles\myfile.c` that defines and initializes the imported variable `myglobal`.

```
#include <stdio.h>

/* Variable definitions for imported variables */
double myglobal = 1.0;
```

Create a code configuration object. Configure the code generation parameters to include `myfile.c`. For output type `'lib'`, or if you generate source code only, you can generate code without providing this file. Otherwise, you must provide this file.

```
cfg = coder.config('dll','ecoder', true);
cfg.CustomSource = 'myfile.c';
cfg.CustomInclude = 'c:\myfiles';
```

Generate the code. This example uses the `-globals` argument to specify the type and initial value of `myglobal`. Alternatively, you can define global variables in the MATLAB global workspace. For imported global variables, the code generation software uses the initial values to determine only the type.

```
codegen -config cfg -globals {'myglobal', 1} -args {1}  addglobal_im -report
```

From the initial value `1`, `codegen` determines that `myglobal` has type `double`. `codegen` declares the imported global variable `myglobal`. It does not define `myglobal` or generate code that initializes `myglobal`. `myfile.c` provides the code that defines and initializes `myglobal`.

To view the generated code for `myglobal`, click the `View report` link.

`myglobal` is declared as `extern` in the `Variable Declarations` section in `addglobal_im_data.h`.

```
/* Variable Declarations */
/* Declaration for custom storage class: ImportedExtern */
extern double myglobal;
```

#### Import External Pointer

In the function `addglobal_imptr`, assign the `'ImportedExternPointer'` storage class to the global variable `myglobal`.

```
function y = addglobal_imptr(x)

% Define the global variable.
```

```
global myglobal;

% Assign the storage classes.

coder.storageClass('myglobal', 'ImportedExternPointer');
y = myglobal + x;
end
```

Create a file `c:\myfiles\myfile.c` that defines and initializes the imported global variable `myglobal`.

```
#include <stdio.h>

/* Variable definitions for imported variables */
double v = 1.0;
double *myglobal = &v;
```

Create a code configuration object. Configure the code generation parameters to include `myfile.c`. For output type `'lib'`, or if you generate source code only, you can generate code without providing this file. Otherwise, you must provide this file.

```
cfg = coder.config('dll','ecoder', true);
cfg.CustomSource = 'myfile.c';
cfg.CustomInclude = 'c:\myfiles';
```

Generate the code. This example uses the `-globals` argument to specify the type and initial value of the global variable `myglobal`. Alternatively, you can define global variables in the MATLAB global workspace. For imported global variables, the code generation software uses the initial values to determine only the type.

```
codegen -config cfg -globals {'myglobal', 1} -args {1} addglobal_imptr -report
```

From the initial value `1`, `codegen` determines that `myglobal` has type `double`. `codegen` declares the imported global variable `myglobal`. It does not define `myglobal` or generate code that initializes `myglobal`. `myfile.c` provides the code that defines and initializes `myglobal`.

To view the generated code for `myglobal`, click the `View report` link.

`myglobal` is declared as `extern` in the `Variable Declarations` section in `addglobal_imptr_data.h`.

```
/* Variable Declarations */
/* Declaration for custom storage class: ImportedExternPointer */
extern double *myglobal;
```

- “Control Declarations and Definitions of Global Variables in Code Generated from MATLAB Code”

## Input Arguments

**global_name — Name of global variable**
character vector

`global_name` is the name of a global variable, specified as a character vector. `global_name` must be a compile-time constant.

Example: `'myglobal'`

Data Types: `char`

**storage_class — Name of storage class**
`'ExportedGlobal'` | `'ExportedDefine'` | `'ImportedExtern'` | `'ImportedExternPointer'`

Storage class to assign to `global_var`. `storage_class` can have one of the following values.

| Storage Class | Description |
| --- | --- |
| `'ExportedGlobal'` | • Defines the variable in the `Variable Definitions` section of the C file *`entry_point_name`*`.c`.<br>• Declares the variable as an `extern` in the `Variable Declarations` section of the header file *`entry_point_name`*`.h`<br>• Initializes the variable in the function *`entry_point_name`*`_initialize.h`. |

| Storage Class | Description |
|---|---|
| 'ExportedDefine' | Declares the variable with a #define directive in the Exported data define section of the header file *entry_point_name*.h. |
| 'ImportedExtern' | Declares the variable as an extern in the Variable Declarations section of the header file *entry_point_name*_data.h. The external code must supply the variable definition. |
| 'ImportedExternPointer' | Declares the variable as an extern pointer in the Variable Declarations section of the header file *entry_point_name*_data.h. The external code must define a valid pointer variable. |

- If you do not assign a storage class to a global variable, except for the declaration location, the variable behaves like it has an 'ExportedGlobal' storage class. For an 'ExportedGlobal' storage class, the global variable is declared in the file *entry_point_name*.h. When the global variable does not have a storage class, the variable is declared in the file *entry_point_name*_data.h.

Data Types: char

## Limitations

- After you assign a storage class to a global variable, you cannot assign a different storage class to that global variable.
- You cannot assign a storage class to a constant global variable.

## See Also

codegen

### Topics

"Control Declarations and Definitions of Global Variables in Code Generated from MATLAB Code"

"Storage Classes for Code Generation from MATLAB Code"

**Introduced in R2015b**

# compare

**Class:** cgv.CGV
**Package:** cgv

Compare signal data

## Syntax

```
[matchNames, matchFigures, mismatchNames, mismatchFigures] =
cgv.CGV.compare(data_set1, data_set2)
[matchNames, matchFigures, mismatchNames, mismatchFigures] =
cgv.CGV.compare(data_set1, data_set2, 'Plot', param_value)
[matchNames, matchFigures, mismatchNames, mismatchFigures] =
cgv.CGV.compare(data_set1, data_set2, 'Plot', 'none', 'Signals',
signal_list, 'ToleranceFile', file_name)
```

## Description

`[matchNames, matchFigures, mismatchNames, mismatchFigures] = cgv.CGV.compare(data_set1, data_set2)` compares data from two data sets which have common signal names between both executions. Possible outputs of the `cgv.CGV.compare` function are matched signal names, figure handles to the matched signal names, mismatched signal names, and figure handles to the mismatched signal names. By default, `cgv.CGV.compare` looks at the signals which have a common name between both executions.

`[matchNames, matchFigures, mismatchNames, mismatchFigures] = cgv.CGV.compare(data_set1, data_set2, 'Plot', param_value)` compares the signals and plots the signals according to `param_value`.

`[matchNames, matchFigures, mismatchNames, mismatchFigures] = cgv.CGV.compare(data_set1, data_set2, 'Plot', 'none', 'Signals', signal_list, 'ToleranceFile', file_name)` compares only the given signals and does not produce plots.

## Input Arguments

### **`data_set1, data_set2`**

Output data from a model. After running the model, use the `getOutputData` function to get the data. The `cgv.CGV.getOutputData` function returns a cell array of the output signal names.

### **`varargin`**

Variable number of parameter name and value pairs.

## varargin Parameters

You can specify the following argument properties for the `cgv.CGV.compare` function using parameter name and value argument pairs. These parameters are optional.

`Plot`(optional)

Designates which comparison data to plot. The value of this parameter must be one of the following:

- `'match'`: plot the comparison of the matched signals from the two data sets
- `'mismatch'`(default): plot the comparison of the mismatched signals from the two datasets
- `'none'`: do not produce a plot

`Signals`(optional)

A cell array of character vectors, where each vector is a signal name in the output data. Use `getSavedSignals` to view the list of available signal names in the output data. `signal_list` can contain an individual signal or multiple signals. The syntax for an individual signal name is:

```
signal_list = {'log_data.subsystem_name.Data(:,1)'}
```

The syntax for multiple signal names is:

```
signal_list = {'log_data.block_name.Data(:,1)',...
                                      'log_data.block_name.Data(:,2)',...
                       'log_data.block_name.Data(:,3)',...
                       'log_data.block_name.Data(:,4)'};
```

If a model component contains a space or newline character, MATLAB adds parentheses and a single quote to the name of the component. For example, if a section of the signal has a space, `'block name'`, MATLAB displays the signal name as:

```
log_data.('block name').Data(:,1)
```

To use the signal name as input to a CGV function, `'block name'` must have two single quotes. For example:

```
signal_list = {'log_data.(''block name'').Data(:,1)'}
```

If `Signals` is not present, the signals are compared.

`Tolerancefile`(optional)

Name for the file created by the `createToleranceFile` function. The file contains the signal names and the associated tolerance parameter name and value pair for comparing the data.

## Output Arguments

Depending on the data and the parameters, the following output arguments might be empty.

**`match_names`**

Cell array of matching signal names.

**`match_figures`**

Array of figure handles for matching signals

**`mismatch_names`**

Cell array of mismatching signal names

**`mismatch_figures`**

Array of figure handles for mismatching signals

## See Also

### Topics

"Verify Numerical Equivalence with CGV"

# configModel

**Class:** cgv.Config
**Package:** cgv

Determine and change configuration parameter values

## Syntax

`cfgObj.configModel()`

## Description

`cfgObj.configModel()` determines the recommended values for the configuration parameters in the model. *`cfgObj`* is a handle to a `cgv.Config` object. The `ReportOnly` property of the object determines whether `configModel` changes the configuration parameter values.

## See Also

### Topics

“About Model Configurations” (Simulink)
“Programmatic Code Generation Verification”

# checkEnvSetup

Configure the code generator to interact with Code Composer Studio

## Syntax

`checkEnvSetup(`*`ide`*`, `*`boardproc`*`, `*`action`*`)`

## IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3
- Texas Instruments Code Composer Studio v4
- Texas Instruments Code Composer Studio v5

## Description

The `checkEnvSetup` function is only useful for validating the toolchain when the **System target file** parameter is set to `idelink_ert.tlc` or `idelink_grt.tlc`. Do not use `checkEnvSetup` when **System target** file is set to `ert.tlc`. The System target file parameter is located on the Code Generation pane in the Configuration Parameters dialog box. For more information, see "System target file" (Simulink Coder).

Before using the code generator with Texas Instruments Code Composer Studio IDE for the first time, use the `checkEnvSetup` function to verify that you have the required third-party tools, as described in:

- "Compare Version Numbers of Installed vs. Required Tools" on page 1-117
- "Set the Environment Variables" on page 1-117.

Run `checkEnvSetup` again whenever you configure CCS IDE to interact with a new board or processor, or upgrade the related third-party tools.

The syntax for this function is: `checkEnvSetup(`*`ide`*`, `*`boardproc`*`, `*`action`*`)`:

- For the *ide* argument, enter the IDE you want to check:
  - 'ccs' checks the setup for Code Composer Studio v3
  - 'ccsv4' checks the setup for Code Composer Studio v4
  - 'ccsv5' checks the setup for Code Composer Studio v5
- For the *boardproc* argument, enter the name of a supported board or processor. You can get these names from the **Processor** parameter on the Target Hardware Resources tab (see related link at bottom of topic). For example, enter: 'F2812'.
- For the *action* argument, specify the action you want this function to perform:
  - 'list' lists the required third-party tools and version numbers.
  - 'check' lists the required third-party tools and the ones on your development system. If tools are missing, install them. If the version numbers do not match, install the required version.
  - 'setup' creates environment variables that point to the installation folders of the third-party tools. This action is required.

If your tools do not meet the requirements, the function advises you. If path information is incomplete, the function prompts you to enter path information for specific tools.

If you omit the *action* argument, the method defaults to 'setup'.

If *action* is 'list' or 'check', the checkEnvSetup function returns an output argument that contains the third-party tool information. You can assign that output argument to a variable. When *action* is 'setup', the checkEnvSetup function does not return an output argument.

## Examples

### Get Information About Required Tools

To find out which third-party tools your board requires, including version numbers, use 'list' as the third argument.

```
checkEnvSetup('ccs', 'F2808 eZdsp', 'list')
```

```
1. CCS (Code Composer Studio)
   Required version: 3.3.82.13
   Required for    : Automation and Code Generation
```

```
2. CGT (Texas Instruments C2000 Code Generation Tools)
   Required version: 5.2.1
   Required for    : Code generation

3. DSP/BIOS (Real Time Operating System)
   Required version: 5.33.05
   Required for    : Real-Time Data Exchange (RTDX)

4. Flash Tools (TMS320C2808 Flash APIs)
   Required version: 3.02
   Required for    : Flash Programming
   Required environment variables (name, value):
   (FLASH_2808_API_INSTALLDIR,  "<Flash Tools (TMS320C2808 Flash APIs) installation folder>")
```

### Compare Version Numbers of Installed vs. Required Tools

To compare “Your version” of the installed third-party tools with the “Required version”, use `'check'` as the third argument.

To resolve differences between the two version numbers, install the required software versions. Using versions of the software that are different from the required version can produce unexpected results.

```
checkEnvSetup('ccs', 'c6416', 'check')
```

```
1. CCS (Code Composer Studio)
   Your version    : 3.3.38.2
   Required version: 3.3.82.13
   Required for    : Automation and Code Generation

2. CGT (Code Generation Tools)
   Your version    : 6.0.8
   Required version: 6.1.10
   Required for    : Code generation

3. DSP/BIOS (Real Time Operating System)
   Your version    :
   Required version: 5.33.05
   Required for    : Code generation

4. Texas Instruments IMGLIB (TMS320C64x)
   Your version    : 1.04
   Required version: 1.04
   Required for    : CRL block replacement
   C64X_IMGLIB_INSTALLDIR = "E:\apps\TexasInstruments\C6400\imglib_v104b"
```

### Set the Environment Variables

After verifying that you have the required versions of the third-party tools, set the environment variables. Use `'setup'` as the *`action`* argument, or omit the *`action`* argument.

This step is required before the code generator can use Texas Instruments Code Composer Studio to build and run an executable.

```
checkEnvSetup('ccs', 'dm6437evm')
```

```
1. Checking CCS (Code Composer Studio) version
   Required version: 3.3.82.13
   Required for    : Automation and Code Generation
   Your Version    : 3.3.38.13

2. Checking CGT (Code Generation Tools) version
   Required version: 6.1.10
   Required for    : Code generation
   Your Version    : 6.1.10

3. Checking DSP/BIOS (Real Time Operating System) version
   Required version: 5.33.05
   Required for    : Code generation
   Your Version    : 5.33.05

4. Checking Texas Instruments IMGLIB (C64x+) version
   Required version: 2.0.1
   Required for    : CRL block replacement
   Your Version    : 2.0.1
   ### Setting environment variable "C64XP_IMGLIB_INSTALLDIR"
   ### to "E:\apps\TexasInstruments\C64Plus\imglib_v201"

5. Checking DM6437EVM DVSDK (Digital Video Software Developers Kit) version
   Required version: 1.01.00.15
   Required for    : Code generation
   Your Version    : 1.01.00.15
  ### Setting environment variable "DVSDK_EVMDM6437_INSTALLDIR" to "C:\[...]"
  ### Setting environment variable "CSLR_DM6437_INSTALLDIR" to "C:\dvsd[...]"
  ### Setting environment variable "PSP_EVMDM6437_INSTALLDIR" to "C:\dv[...]"
  ### Setting environment variable "NDK_INSTALL_DIR" to "C:\dvsdk_1_01_[...]"
```

## See Also

“System target file” (Simulink Coder) | “Code Generation: Target Hardware Resources Pane” on page 13-27

### Topics

“Choose Build Approach and Configure Build Process” (Simulink Coder)

**Introduced in R2011a**

# close

Close project in IDE window

## Syntax

```
close(IDE_Obj,filename,'project')
```

## IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

## Description

Use `close(IDE_Obj,filename,'project')` to close a specific project, projects, or the active open project.

For the *filename* argument:

- To close the project files, enter `'all'`.
- To close a specific project, enter the project file name, such as `'myProj'`. If the file is not an open file in the IDE, MATLAB returns a warning message.
- To close the active project, enter `[]`.

With the VisualDSP++ IDE, to close the current project group (if *filename* is `'all'` or `[]`), replace `'project'`with `'projectgroup'`.

**Note**

- The `open` method does not support the `'text'` argument.

- Save changes to your files and projects in the IDE before you use `close`. The `close` method does not save changes, nor does it prompt you to save changes, before it closes the project.

---

## Examples

To close the open project files:

```
close(IDE_Obj,'all','project')
```

To close the open project, `myProj`:

```
close(IDE_Obj,'myProj','project')
```

To close the active open project:

```
close(IDE_Obj,[],'project')
```

With the VisualDSP++ IDE, to close the open project groups:

```
close(IDE_Obj,'all','projectgroup')
```

With the VisualDSP++ IDE, to close the active project group:

```
close(IDE_Obj,[],'projectgroup')
```

## See Also

add | open | save

**Introduced in R2011a**

# coder.MATLABCodeTemplate class

**Package:** coder

Represent code generation template for MATLAB Coder

## Description

Create a coder.MATLABCodeTemplate object from a code generation template (CGT) file. You can use this file to customize the code generation output for MATLAB Coder™. If a CGT file is not provided, the coder.MATLABCodeTemplate object is created from the default template file `*matlabroot*/toolbox/coder/matlabcoder/templates/matlabcoder_default_template.cgt`.

## Construction

`newObj = coder.MATLABCodeTemplate()` creates a `coder.MATLABCodeTemplate` object from the default code generation template (CGT) file *`matlabroot`*`/toolbox/coder/matlabcoder/templates/matlabcoder_default_template.cgt`.

`newObj = coder.MATLABCodeTemplate(CGTFile)` creates a `coder.MATLABCodeTemplate` object from the code generation template file `CGTFile`. If the file is not on the MATLAB path, specify a full path to the file.

### Input Arguments

#### `CGTFile`

Name of code generation template file

## Methods

| | |
|---|---|
| emitSection | Emit comments for template section |
| getCurrentTokens | Get current tokens |
| getTokenValue | Get value of token |
| setTokenValue | Set value of token for code generation template |

## Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

## Examples

```
newObj = coder.MATLABCodeTemplate()

newObj =

  MATLABCodeTemplate with properties:

    CGTFile: 'matlabcoder_default_template.cgt'
```

```
newObj = coder.MATLABCodeTemplate('custom_matlabcoder_template.cgt')

newObj =

  MATLABCodeTemplate with properties:

    CGTFile: 'custom_matlabcoder_template.cgt'
```

## See Also

coder.MATLABCodeTemplate.emitSection | coder.MATLABCodeTemplate.getCurrentTokens | coder.MATLABCodeTemplate.getTokenValue | coder.MATLABCodeTemplate.setTokenValue

## Topics

"Generate Custom File and Function Banners for C/C++ Code"
"Code Generation Template Files for MATLAB Code"

# configure

Define size and number of RTDX channel buffers

## Syntax

`configure(rx,length,num)`

---

**Note** `configure` produces a warning on C5000™ processors and will be removed from a future version of the software.

---

## IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

## Description

`configure(rx,length,num)` sets the size of each main (host) buffer, and the number of buffers associated with `rx`. Input argument *`length`* is the size in bytes of each channel buffer and *`num`* is the number of channel buffers to create.

Main buffers must be at least 1024 bytes, with the maximum defined by the largest message. On 16-bit processors, the main buffer must be 4 bytes larger than the largest message. On 32-bit processors, set the buffer to be 8 bytes larger that the largest message. By default, `configure` creates four, 1024-byte buffers. Independent of the value of *`num`*, the IDE allocates one buffer for each processor.

Use CCS to check the number of buffers and the length of each one.

## Examples

Create a default link to CCS and configure six main buffers of 4096 bytes each for the link.

```
 IDE_Obj = ticcs            % Create the CCS link with default values.

TICCS Object:
  API version      : 1.0
  Processor type   : C67
  Processor name   : CPU
  Running?         : No
  Board number     : 0
  Processor number : 0
  Default timeout  : 10.00 secs

  RTDX channels    : 0

rx = rtdx(IDE_Obj)              % Create an alias to the rtdx portion.

RTDX channels    : 0

configure(rx,4096,6)  % Use the alias rx to configure the length
                      % and number of buffers.
```

After you configure the buffers, use the RTDX™ tools in the IDE to verify the buffers.

## See Also

`readmat` | `readmsg` | `write` | `writemsg`

**Introduced in R2011a**

# connect

Connect IDE to processor

## Syntax

```
IDE_Obj.connect()
IDE_Obj.connect(debugconnection)
IDE_Obj.connect(...,timeout)
```

## IDEs

This function supports the following IDEs:

- Green Hills® MULTI®

## Description

*IDE_Obj*.connect() connects the IDE to the processor hardware or simulator. IDE_Obj is the IDE handle.

*IDE_Obj*.connect(*debugconnection*) connects the IDE to the processor using the debug connection you specify in debugconnection. Enter debugconnection as a character vector enclosed in single quotation marks. IDE_Obj is the IDE handle. Refer to Examples to see this syntax in use.

*IDE_Obj*.connect(...,*timeout*) adds the optional parameter timeout that defines how long, in seconds, MATLAB waits for the specified connection process to complete. If the time-out period expires before the process returns a completion message, MATLAB generates an error and returns. Usually the program connection process works in spite of the error message

## Examples

The input argument character vector`debugconnection` specify the processor to connect to with the IDE. This example connects to the Freescale™ MPC5554 simulator. The `debugconnection` character vector is `simppc -fast -dec -rom_use_entry -cpu=ppc5554`.

```
IDE_Obj.connect('simppc -fast -dec -rom_use_entry -cpu=ppc5554')
```

## See Also

`load` | `run`

**Introduced in R2011a**

# copySetup

**Class:** cgv.CGV
**Package:** cgv

Create copy of `cgv.CGV` object

## Syntax

`cgvObj2 = cgvObj1.copySetup()`

## Description

`cgvObj2 = cgvObj1.copySetup()` creates a copy of a `cgv.CGV` on page 1-82 object, *`cgvObj1`*. The copied object, `cgvObj2`, has the same configuration as *`cgvObj1`*, but does not copy results of the execution.

## Examples

Make a copy of a `cgv.CGV` object, set it to run in a different mode, then run and compare the objects in a `cgv.Batch` object.

```
cgvModel = 'rtwdemo_cgv';
cgvObj1 = cgv.CGV(cgvModel, 'connectivity', 'sim');
cgvObj1.run();
cgvObj2 = cgvObj1.copySetup()
cgvObj2.setMode('sil');
cgvObj2.run();
```

## Tips

- You can use this method to make a copy of a `cgv.CGV` object and then modify the object to run in a different mode by calling `setMode`.
- If you have a `cgv.CGV` object, which reported errors or failed at execution, you can use this method to copy the object and rerun it. The copied object has the same

configuration as the original object, therefore you might want to modify the location of the output files by calling `setOutputDir`. Otherwise, during execution, the copied `cgv.CGV` object overwrites the output files.

## See Also

`cgv.CGV.run`

### Topics

"Verify Numerical Equivalence with CGV"

# copyConceptualArgsToImplementation

Copy conceptual argument specifications to implementation specfications of an entry for code replacement table entry

## Syntax

`copyConceptualArgsToImplementation(hEntry)`

## Description

`copyConceptualArgsToImplementation(hEntry)` provides a quick way to perform a shallow copy of conceptual arguments to matching implementation arguments.

The conceptual arguments and implementation arguments refer to the same argument instance. If you update an implementation argument, the corresponding conceptual argument is also updated.

Use this function when the conceptual arguments and the implementation arguments are the same for a code replacement table entry.

For arguments with an unsized type, such as `integer`, the code generator determines the size of the argument values based on hardware implementation configuration settings of the MATLAB code or model.

## Examples

### Copy Conceptual Argument to Implementation Arguments

This example shows how to use the `copyConceptualArgsToImplementation` function to copy conceptual argument specifications to matching implementation arguments for an addition operation.

```
hLib = RTW.TflTable;
```

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

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

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

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

op_entry.copyConceptualArgsToImplementation();

hLib.addEntry( op_entry );
```

## Input Arguments

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

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

Example: op_entry

## See Also

### Topics
“Define Code Replacement Mappings”

“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”

**Introduced in R2007b**

# createAndAddConceptualArg

Create conceptual argument from specified properties and add to conceptual arguments for code replacement table entry

## Syntax

```
arg = createAndAddConceptualArg(hEntry,argType, varargin)
```

## Description

`arg = createAndAddConceptualArg(hEntry,argType, varargin)` creates a conceptual argument from specified properties and adds the argument to the conceptual arguments for a code replacement table entry.

## Examples

### Specify Conceptual Output and Input Arguments

This example shows how to use the`createAndAddConceptualArg` function to specify conceptual output and input arguments for a code replacement operator entry.

For examples of fixed-point arguments that use relative scaling or relative slope/bias values, see “Net Slope Scaling Code Replacement” and “Equal Slope and Zero Net Bias Code Replacement”.

```
op_entry = RTW.TflCOperationEntry;
.
.
.
createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
    'Name',       'y1', ...
    'IOType',     'RTW_IO_OUTPUT', ...
    'IsSigned',   true, ...
    'WordLength', 32, ...
```

```
    'FractionLength', 0);

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric',...
    'Name',       'u1', ...
    'IOType',     'RTW_IO_INPUT',...
    'IsSigned',   true,...
    'WordLength', 32, ...
    'FractionLength', 0 );

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric',...
    'Name',       'u2', ...
    'IOType',     'RTW_IO_INPUT',...
    'IsSigned',   true,...
    'WordLength', 32, ...
    'FractionLength', 0 );
```

#### Specify Types for Conceptual Argument

These examples show some common type specifications using `createAndAddConceptualArg`.

```
% uint8:
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',           'u1', ...
    'IOType',         'RTW_IO_INPUT', ...
    'IsSigned',       false, ...
    'WordLength',     8, ...
    'FractionLength', 0 );

% single:
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',         'u1', ...
    'IOType',       'RTW_IO_INPUT', ...
    'DataTypeMode', 'single' );

% double:
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',         'y1', ...
    'IOType',       'RTW_IO_OUTPUT', ...
    'DataTypeMode', 'double' );

% boolean:
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
```

```
    'Name',         'u1', ...
    'IOType',       'RTW_IO_INPUT', ...
    'DataTypeMode', 'boolean' );

% Fixed-point using binary-point-only scaling:
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',           'y1', ...
    'IOType',         'RTW_IO_OUTPUT', ...
    'CheckSlope',     true, ...
    'CheckBias',      true, ...
    'DataTypeMode',   'Fixed-point: binary point scaling', ...
    'IsSigned',       true, ...
    'WordLength',     32, ...
    'FractionLength', 28);

% Fixed-point using [slope bias] scaling:
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',           'y1', ...
    'IOType',         'RTW_IO_OUTPUT', ...
    'CheckSlope',     true, ...
    'CheckBias',      true, ...
    'DataTypeMode',   'Fixed-point: slope and bias scaling', ...
    'IsSigned',       true, ...
    'WordLength',     16, ...
    'Slope',          15, ...
    'Bias',           2);
```

## Input Arguments

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

handle

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

Example: `op_entry`

**`argType` — Specifies the argument type to create**

`'RTW.TflArgNumeric'` | `'RTW.TflArgMatrix'`

The *`argType`* is a character vector that specifies the argument type to create. Use `'RTW.TflArgNumeric'` for numeric or `'RTW.TflArgMatrix'` for matrix.

Example: `'RTW.TflArgNumeric'`

**`varargin` — Name-value pair arguments that specify the conceptual argument**
name-value pair

Example: `'Name','y1'`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Name','y1'`

**`Name` — Specifies the argument name**
character vector

Example: `'Name','y1'`

**`IOType` — Specifies the I/O type of the argument**
`'RTW_IO_INPUT'` (default) | `'RTW_IO_OUTPUT'`

Use value `'RTW_IO_INPUT'` for input or value `'RTW_IO_OUTPUT'`.

Example: `'IOType','RTW_IO_INPUT'`

**`IsSigned` — Indicates whether the argument is signed**
`true` (default) | `false`

Boolean value that, when set to `true`, indicates that the argument is signed.

Example: `'IsSigned',true`

**`WordLength` — Specifies the word length, in bits, of the argument**
`16` (default) | integer

Integer specifying the word length, in bits, of the argument. The default is `16`.

Example: `'WordLength',16`

**`CheckSlope` — Selects whether to check that the slope value of the argument exactly matches the call-site slope value**
`true` (default) | `false`

Boolean flag that, when set to `true` for a fixed-point argument, causes code replacement request processing to check that the slope value of the argument exactly matches the call-site slope value.

Specify `true` if you are matching a specific [slope bias] scaling combination or a specific binary-point-only scaling combination on fixed-point operator inputs and output. Specify `false` if you are matching relative scaling or relative slope and bias values across fixed-point operator inputs and output.

Example: `'CheckSlope',true`

**`CheckBias` — Selects whether to check that the bias value of the argument exactly matches the call-site bias value**
`true` (default) | `false`

Boolean flag that, when set to `true` for a fixed-point argument, causes code replacement request processing to check that the bias value of the argument exactly matches the call-site bias value.

Specify `true` if you are matching a specific [slope bias] scaling combination or a specific binary-point-only scaling combination on fixed-point operator inputs and output. Specify `false` if you are matching relative scaling or relative slope and bias values across fixed-point operator inputs and output.

Example: `'CheckBias',true`

**DataTypeMode — Specifies the data type mode of the argument**
`'Fixed-point: binary point scaling'` (default) | `'Fixed-point: slope and bias scaling'` | `'boolean'` | `'double'` | `'single'`

You can specify either `DataType` (with `Scaling`) or `DataTypeMode`, but do not specify both.

Example: `'DataTypeMode','Fixed-point: binary point scaling'`

**DataType — Specifies the data type of the argument**
`'Fixed'` (default) | `'boolean'` | `'double'` | `'single'`

Example: `'DataType','Fixed'`

**`Scaling` — Specifies the data type scaling of the argument**
`'BinaryPoint'` (default) | `'SlopeBias'`

Specify the data type scaling of the argument as `'BinaryPoint'` for binary-point scaling or `'SlopeBias'` for slope and bias scaling.

Example: `'Scaling','BinaryPoint'`

**`Slope` — Specifies the slope of the argument**

`1` (default) | floating-point value

If you are matching a specific [slope bias] scaling combination on fixed-point operator inputs and output, specify either this parameter or a combination of the `SlopeAdjustmentFactor` and `FixedExponent` parameters.

Example: `'Slope',1.0`

**`SlopeAdjustmentFactor` — Specifies the slope adjustment factor (F) part of the slope, $F2^E$, of the argument**

`1.0` (default) | floating-point value

If you are matching a specific [slope bias] scaling combination on fixed-point operator inputs and output, specify either the `Slope` parameter or a combination of this parameter and the `FixedExponent` parameter.

Example: `'SlopeAdjustmentFactor',1.0`

**`FixedExponent` — Specifies the fixed exponent (E) part of the slope, $F2^E$, of the argument**

`-15` (default) | integer value

If you are matching a specific [slope bias] scaling combination on fixed-point operator inputs and output, specify either the `Slope` parameter or a combination of this parameter and the `SlopeAdjustmentFactor` parameter.

Example: `'FixedExponent',-15`

**`Bias` — Specifies the bias of the argument**

`0.0` (default) | floating-point value

Specify this parameter if you are matching a specific [slope bias] scaling combination on fixed-point operator inputs and output.

Example: `'Bias',2.0`

**`FractionLength` — Specifies the fraction length for the argument**

`15` (default) | integer value

Specify this parameter if you are matching a specific binary-point-only scaling combination on fixed-point operator inputs and output.

Example: `'FractionLength',15`

**`BaseType` — Specifies the base data type for which a matrix argument is valid**
character vector

Example: `'BaseType','double'`

**`DimRange` — Specifies the dimensions for which a matrix argument is valid**
matrix dimensions

You can also specify a range of dimensions specified in the format `[Dim1Min Dim2Min ... DimNMin; Dim1Max Dim2Max ... DimNMax]`. For example, `[2 2; inf inf]` means a two-dimensional matrix of size 2x2 or larger.

Example: `'DimRange',[2 2]`

# Output Arguments

**`arg` — Handle to the created conceptual argument**
handle

The *`arg`* is a handle to the created conceptual argument. Specifying the return argument in the `createAndAddConceptualArg` function call is optional.

# See Also

## Topics

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

**Introduced in R2007b**

# createAndAddImplementationArg

Create implementation argument from specified properties and add to implementation arguments for code replacement table entry

## Syntax

`arg = createAndAddImplementationArg(hEntry,argType,varargin)`

## Description

`arg = createAndAddImplementationArg(hEntry,argType,varargin)` creates an implementation argument from specified properties and adds the argument to the implementation arguments for a code replacement table entry.

Implementation arguments must describe fundamental numeric data types, such as `double`, `single`, `int32`, `int16`, `int8`, `uint32`, `uint16`, `uint8`, `boolean`, or `'logical'` (not fixed-point data types).

## Examples

### Specify Implementation Output and Input Arguments

This example shows how to use the`createAndAddImplementationArg` function with the `createAndSetCImplementationReturn` function to specify the output and input arguments for an operator implementation.

```
op_entry = RTW.TflCOperationEntry;
.
.
.
createAndSetCImplementationReturn(op_entry, 'RTW.TflArgNumeric', ...
    'Name',       'y1', ...
    'IOType',     'RTW_IO_OUTPUT', ...
    'IsSigned',   true, ...
```

```
    'WordLength', 32, ...
    'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
    'Name',       'u1', ...
    'IOType',     'RTW_IO_INPUT',...
    'IsSigned',   true,...
    'WordLength', 32, ...
    'FractionLength', 0 );

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
    'Name',       'u2', ...
    'IOType',     'RTW_IO_INPUT',...
    'IsSigned',   true,...
    'WordLength', 32, ...
    'FractionLength', 0 );
```

### Specify Types for Implementation Argument

These examples show some common type specifications using `createAndAddImplementationArg`.

```
% uint8:
createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',           'u1', ...
    'IOType',         'RTW_IO_INPUT', ...
    'IsSigned',       false, ...
    'WordLength',     8, ...
    'FractionLength', 0 );

% single:
createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',         'u1', ...
    'IOType',       'RTW_IO_INPUT', ...
    'DataTypeMode', 'single' );

% double:
createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',         'u1', ...
    'IOType',       'RTW_IO_INPUT', ...
    'DataTypeMode', 'double' );

% boolean:
```

```
createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',         'u1', ...
    'IOType',       'RTW_IO_INPUT', ...
    'DataTypeMode', 'boolean' );
```

## Input Arguments

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

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

Example: op_entry

**argType — Specifies the argument type to create**
'RTW.TflArgNumeric' | character vector

The *argType* is a character vector that specifies the argument type to create. Use 'RTW.TflArgNumeric' for numeric.

Example: 'RTW.TflArgNumeric'

**varargin — Name-value pairs that specify the implementation argument**
name-value pairs

Example: 'Name','u1'

## Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'Name','u1'

**Name — Specifies the argument name**
character vector

Example: 'Name','u1'

**`IOType` — Specifies the I/O type of the argument**
`'RTW_IO_INPUT'` | character vector

Use `'RTW_IO_INPUT'` for input.

Example: `'IOType','RTW_IO_INPUT'`

**`IsSigned` — Indicates whether the argument is signed**
`true` (default) | `false`

Boolean value that, when set to `true`, indicates that the argument is signed.

Example: `'IsSigned',true`

**`WordLength` — Specifies the word length, in bits, of the argument**
`16` (default) | integer value

Example: `'WordLength',16`

**`DataTypeMode` — Specifies the data type mode of the argument**
`'Fixed-point: binary point scaling'` (default) | `'Fixed-point: slope and bias scaling'` | `'boolean'` | `'double'` | `'single'`

You can specify either `DataType` (with `Scaling`) or `DataTypeMode`, but do not specify both.

Example: `'DataTypeMode','Fixed-point: binary point scaling'`

**`DataType` — Specifies the data type of the argument**
`'Fixed'` (default) | `'boolean'` | `'double'` | `'single'`

Example: `'DataType','Fixed'`

**`Scaling` — Specifies the data type scaling of the argument**
`'BinaryPoint'` (default) | `'SlopeBias'`

Use `'BinaryPoint'` for binary-point scaling or `'SlopeBias'` for slope and bias scaling.

Example: `'Scaling','BinaryPoint'`

**`Slope` — Specifies the slope of the argument**
`1.0` (default) | floating-point value

You can optionally specify either this parameter or a combination of the `SlopeAdjustmentFactor` and `FixedExponent` parameters, but do not specify both.

Example: 'Slope',1.0

**SlopeAdjustmentFactor — Specifies the slope adjustment factor (F) part of the slope, $F2^E$, of the argument**

1.0 (default) | floating-point value

You can optionally specify either the Slope parameter or a combination of this parameter and the FixedExponent parameter, but do not specify both.

Example: 'SlopeAdjustmentFactor',1.0

**FixedExponent — Specifies the fixed exponent (E) part of the slope, $F2^E$, of the argument**

-15 (default) | integer value

You can optionally specify either the Slope parameter or a combination of this parameter and the SlopeAdjustmentFactor parameter, but do not specify both.

Example: 'FixedExponent',0

**Bias — Specifies the bias of the argument**

0.0 (default) | floating-point value

Example: 'Bias',0.0

**FractionLength — Specifies the fraction length of the argument**

15 (default) | integer value

Example: 'FractionLength',0

**Value — Specifies the initial value of the argument**

0 (default) | constant value

Use this parameter only to set the value of injected constant input arguments, such as arguments that pass fraction-length values or flag values, in an implementation function signature. Do not use it for standard generated input arguments, such as u1u2. You can supply a constant input argument that uses this parameter anywhere in the implementation function signature, except as the return argument.

You can inject constant input arguments into the implementation signature for code replacement table entries. If the argument values or the number of arguments required depends on compile-time information, you can use custom matching. For more information, see “Customize Match and Replacement Process”.

Example: `'Value',0`

## Output Arguments

**`arg` — Handle to the created implementation argument**
handle

Specifying the return argument in the `createAndAddImplementationArg` function call is optional.

## See Also

`createAndSetCImplementationReturn`

### Topics

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

**Introduced in R2007b**

# createAndSetCImplementationReturn

Create implementation return argument from specified properties and add to implementation for code replacement table entry

## Syntax

`arg = createAndSetCImplementationReturn(hEntry,argType,varargin)`

## Description

`arg = createAndSetCImplementationReturn(hEntry,argType,varargin)` creates an implementation return argument from specified properties and adds the argument to the implementation for a code replacement table.

Implementation return arguments must describe fundamental numeric data types, such as `double`, `single`, `int32`, `int16`, `int8`, `uint32`, `uint16`, `uint8`, or `boolean` (not fixed-point data types).

## Examples

### Specify Operator Output and Input Arguments

This example shows how to use the `createAndSetCImplementationReturn` function with the `createAndAddImplementationArg` function to specify the output and input arguments for an operator implementation.

```
op_entry = RTW.TflCOperationEntry;
.
.
.
createAndSetCImplementationReturn(op_entry, 'RTW.TflArgNumeric', ...
    'Name',       'y1', ...
    'IOType',     'RTW_IO_OUTPUT', ...
    'IsSigned',   true, ...
```

```
    'WordLength', 32, ...
    'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
    'Name',       'u1', ...
    'IOType',     'RTW_IO_INPUT',...
    'IsSigned',   true,...
    'WordLength', 32, ...
    'FractionLength', 0 );

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
    'Name',       'u2', ...
    'IOType',     'RTW_IO_INPUT',...
    'IsSigned',   true,...
    'WordLength', 32, ...
    'FractionLength', 0 );
```

### Specify Types for Operator Implementation

These examples show some common type specifications using `createAndSetCImplementationReturn`.

```
% uint8:
createAndSetCImplementationReturn(hEntry, 'RTW.TflArgNumeric', ...
    'Name',           'y1', ...
    'IOType',         'RTW_IO_OUTPUT', ...
    'IsSigned',       false, ...
    'WordLength',     8, ...
    'FractionLength', 0 );

% single:
createAndSetCImplementationReturn(hEntry, 'RTW.TflArgNumeric', ...
    'Name',         'y1', ...
    'IOType',       'RTW_IO_OUTPUT', ...
    'DataTypeMode', 'single' );

% double:
createAndSetCImplementationReturn(hEntry, 'RTW.TflArgNumeric', ...
    'Name',         'y1', ...
    'IOType',       'RTW_IO_OUTPUT', ...
    'DataTypeMode', 'double' );

% boolean:
```

```
createAndSetCImplementationReturn(hEntry, 'RTW.TflArgNumeric', ...
    'Name',         'y1', ...
    'IOType',       'RTW_IO_OUTPUT', ...
    'DataTypeMode', 'boolean' );
```

## Input Arguments

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

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

Example: `op_entry`

**`argType` — Specifies the argument type to create**
`'RTW.TflArgNumeric'` | character vector

The *`argType`* is a character vector that specifies the argument type to create. Use `'RTW.TflArgNumeric'` for numeric.

Example: `'RTW.TflArgNumeric'`

**`varargin` — Name-value pairs that specify the implementation return argument**
name-value pairs

Example: `'Name','y1'`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Name','y1'`

**`Name` — Specifies the argument name**
character vector

Example: `'Name','y1'`

**`IOType` — Specifies the I/O type of the argument**
`'RTW_IO_OUTPUT'` | character vector

Use `'RTW_IO_OUTPUT'` for output.

Example: `'IOType','RTW_IO_OUTPUT'`

**`IsSigned` — Indicates whether the argument is signed**
`true` (default) | `false`

Boolean value that, when set to `true`, indicates that the argument is signed. The default is `true`.

Example: `'IsSigned',true`

**`WordLength` — Specifies the word length, in bits, of the argument**
`16` (default) | integer

Example: `'WordLength',16`

**`DataTypeMode` — Specifies the data type mode of the argument**
`'Fixed-point: binary point scaling'` (default) | `'Fixed-point: slope and bias scaling'` | `'boolean'` | `'double'` | `'single'`

You can specify either `DataType` (with `Scaling`) or `DataTypeMode`, but do not specify both.

Example: `'DataTypeMode','Fixed-point: binary point scaling'`

**`DataType` — Specifies the data type of the argument**
`'Fixed'` (default) | `'boolean'` | `'double'` | `'single'`

Example: `'DataType','Fixed'`

**`Scaling` — Specifies the data type scaling of the argument**
`'BinaryPoint'` (default) | `'SlopeBias'`

Use `'BinaryPoint'` for binary-point scaling or `'SlopeBias'` for slope and bias scaling.

Example: `'Scaling','BinaryPoint'`

**`Slope` — Specifies the slope for a fixed-point argument**
`1.0` (default) | floating-point value

You can optionally specify either this parameter or a combination of the `SlopeAdjustmentFactor` and `FixedExponent` parameters, but do not specify both.

Example: 'Slope',1.0

**SlopeAdjustmentFactor — Specifies the slope adjustment factor (F) part of the slope, $F2^E$, of the argument**
1.0 (default) | floating-point value

You can optionally specify either the Slope parameter or a combination of this parameter and the FixedExponent parameter, but do not specify both.

Example: 'SlopeAdjustmentFactor',1.0

**FixedExponent — Specifies the fixed exponent (E) part of the slope, $F2^E$, of the argument**
-15 (default) | integer value

You can optionally specify either the Slope parameter or a combination of this parameter and the SlopeAdjustmentFactor parameter, but do not specify both.

Example: 'FixedExponent',0

**Bias — Specifies the bias of the argument**
0.0 (default) | floating-point value

Example: 'Bias',0.0

**FractionLength — Specifies the fraction length of the argument**
15 (default) | integer value

Example: 'FractionLength',0

## Output Arguments

**arg — Handle to the created implementation return argument**
handle

Specifying the return argument in the createAndSetCImplementationReturn function call is optional.

## See Also

createAndAddImplementationArg

## Topics

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

**Introduced in R2007b**

# createCRLEntry

Create code replacement table entry from conceptual and implementation argument string specifications

## Syntax

```
tableEntry = createCRLEntry(crTable,conceptualSpecification,
implementationSpecification)
```

## Description

`tableEntry = createCRLEntry(crTable,conceptualSpecification, implementationSpecification)` returns a code replacement table entry. The entry maps a conceptual representation of a function or operator to an implementation representation. The `conceptualSpecification` argument is a character vector that defines the name and conceptual arguments, familiar to the code generator, for the function or operator to replace. The `implementationSpecification` argument is a character vector that defines the name and C/C++ implementation arguments for the replacement function.

This function does not support:

- C++ implementations
- Data alignment
- Operator replacement with net slope arguments
- Entry parameter specifications (for example, priority, algorithm, building information)
- Semaphore and mutex function replacements

In the syntax specifications, place a space before and after an operator symbol. For example, use `double u1 + double u2` instead of `double u1+double u2`. Also, asterisk (*), tilde (~), and semicolon (;) have the following meaning.

| Symbol | Meaning |
| --- | --- |
| * | • Following a supported data type, such as `int32*`, pass by reference (pointer). If the conceptual arguments are not scalar, in the implementation specification, pass them by reference.<br>• As part of a fixed-point data type definition, such as `fixdt(1,32,*)`, wildcard. |
| ~ | Based on the position of the symbol, slopes or bias must be the same across data types. |
| ; | Separates dimension ranges. For example, `[1 10; 1 100]` specifies a vector with length from 10 through 100. |

The following table shows syntax for the conceptual and implementation specifications based on:

- Whether you are creating an entry for a function or operator.
- The type or characterization of the code replacement.

| Type of Replacement | Conceptual Syntax | Implementation Syntax |
| --- | --- | --- |
| **Function Code Replacement Syntax** | | |
| Typical | `double y1 = sin(double u1)` | `double y1 = mySin(double u1)` |
| Derive implementation argument data types from conceptual specification | `double y1 = sin(double u1)` | `y1 = mySin(u1)` |
| Derive implementation arguments and data types from conceptual specification | `double y1 = sin(double u1)` | `mySin` |
| Change data type | `single y1 = sin(single u1)` | `double y1 = mySin(double u1)` |

| Type of Replacement | Conceptual Syntax | Implementation Syntax |
| --- | --- | --- |
| Reorder arguments | `double y1 = atan2(double u1, double u2)` | `y1 = myAtan(u2, u1)` |
| Specify column vector arguments | `double y1 = sin(double u1[10])` | `double y1 = mySin(double* u1)` |
| Specify column vector arguments and dimension range | `double y1[1 100; 1 100] = sin(double u1[1 100; 1 100])` | `mySin(double* u1, double* y1)` |
| Remap return value as output argument | `double y1 = sin(double u1)` | `mySin(double u1, double* y1)` |
| Specify fixed-point data types | `fixdt(1,16,3) y1 = sin(fixdt(1,16,3) u1)` | `int16 y1 = mySin(int16 u1)` |
| Specify fixed-point data types and set `CheckSlope` to `false`, `CheckBias` to `true`, and `Bias` to 0 | `fixdt(1,16,*) y1 = sin(fixdt(1,16,*) u1)` | `int16 y1 = mySin(int16 u1)` |
| Specify fixed-point data types and set `SlopesMustBeTheSame` to `true`, `CheckSlope` to `false`, `CheckBias` to `true`, and `Bias` to 0 | `fixdt(1,16,~) y1 = sin(fixdt(1,16,~) u1)` | `int16 y1 = mySin(int16 u1)` |

| Type of Replacement | Conceptual Syntax | Implementation Syntax |
|---|---|---|
| Specify fixed-point data types and set `SlopesMustBeTheSame` to `true`, `BiasMustBeTheSame` to `true`, `CheckSlope` to `false`, and `CheckBias` to `false` | `fixdt(1,16,~,~) y1 = sin(fixdt(1,16,~,~) u1)` | `int16 y1 = mySin(int16 u1)` |
| Specify multiple output arguments | `[double y1 double y2] = foo(double u1, double u2)` | `double y1 = myFoo(double u1, double u2, double* y2)` |
| **Operator Code Replacement Syntax** | | |
| Typical | `int16 y1 = int16 u1 + int16 u2` | `int16 y1 = myAdd(int16 u1, int16 u2)` |
| Specify fixed-point data types | `fixdt(1,16,3) y1 = fixdt(1,16,3) u1 + fixdt(1,16,3) u2` | `int16 y1 = myAdd(int16 u1, int16 u2)` |
| Specify fixed-point data types and set `CheckSlope` to `false`, `CheckBias` to `true`, and `Bias` to 0 | `fixdt(1,16,*) y1 = fixdt(1,16,*) u1 + fixdt(1,16,*) u2` | `int16 y1 = myAdd(int16 u1, int16 u2)` |
| Specify fixed-point data types, wildcard, slopes must be the same, and zero bias | `fixdt(1,16,~,0) y1 = fixdt(1,16,~,0) u1 + fixdt(1,16,~,0) u2` | `int16 y1 = myAdd(int16 u1, int16 u2)` |
| Typecast | `int16 y1 = int8 u1` | `int16 y1 = myCast(int8 u1)` |

| Type of Replacement | Conceptual Syntax | Implementation Syntax |
|---|---|---|
| Shift | `int16 y1 = int16 u1 << int16 u2`<br>`int16 y1 = int16 u1 >> int16 u2`<br>`int16 y1 = int16 u1 .>> int16 u2` | `int16 y1 = myShiftLeft(int16 u1, int16 u2)`<br>`int16 y1 = myShiftRightArithmetic(int16 u1, int16 u2)`<br>`int16 y1 = myShiftRightLogical(int16 u1, int16 u2)` |
| Specify relational operator | `bool y1 = int16 u1 < int16 u2` | `bool y1 = myLessThan(int6 u1, int16 u2)` |
| Specify multiplication and division | `int32 y1 = int32 u1 * in32 u2 / in32 u3` | `int32 y1 = myMultDiv(int32 u1, int32 u2, int32 u3)` |
| Specify matrix multiplication | `double y1[10][10] = double u1[10][10] * double u2[10][10]` | `myMult(double* u1, double* u2, double* y1)` |
| Specify element-wise matrix multiplication | `double y1[10][10] = double u1[10][10] .* double u2[10][10]` | `myMult(double* u1, double* u2, double* y1)` |
| Specify matrix multiplication with transpose of an input argument | `double y1[10][10] = double u1[10][10]' * double u2[10][10]` | `myMult(double* u1, double* u2, double* y1)` |
| Specify matrix multiplication with Hermitian of an input argument | `cdouble y1[10][10] = cdouble u1[10][10]' * cdouble u2[10][10]`<br>`cdouble y1[10][10] = cdouble u1[10][10] * cdouble u2[10][10]'` | `myMult(cdouble* u1, cdouble* u2, cdouble* y1)` |
| Specify left matrix division | `double y1[10][10] = double u1[10][10] \ double u2[10][10]` | `myLeftDiv(double* u1, double* u2, double* y1)` |

| Type of Replacement | Conceptual Syntax | Implementation Syntax |
|---|---|---|
| Specify right matrix division | `double y1[10][10] = double u1[10][10] / double u2[10][10]` | `myRightDiv(double* u1, double* u2, double* y1)` |

## Examples

### Replacement Entry for a Function

Create a table definition file that contains a function definition.

```
function crTable = crl_table_sinfcn()
```

Within the function body, create the code replacement table.

```
crTable = RTW.TflTable;
```

Create a table entry for the `sin` function.

```
tableEntry = createCRLEntry(crTable, ...
    'double y1 = sin(double u1)', ...
    'double y1 = mySin(double u1)');
```

Set entry parameters for the `sin` function. To generate the replacement code, specify that the code generator use the header and source files `mySin.h` and `mySin.c`.

```
setTflCFunctionEntryParameters(tableEntry, ...
    'ImplementationHeaderFile', 'mySin.h', ...
    'ImplementationSourceFile', 'mySin.c');
```

Add the entry to the table.

```
addEntry(crTable, tableEntry);
```

### Replacement Entry for an Operator

Create a table definition file that contains a function definition.

```
function crTable = crl_table_addfcn()
```

Within the function body, create the code replacement table.

```
crTable = RTW.TflTable;
```

Create a table entry for the addition operator.

```
tableEntry = createCRLEntry(crTable, ...
    'int16 y1 = int16 u1 + int16 u2', ...
    'int16 y1 = myAdd(int16 u1, int16 u2)');
```

Set entry parameters such that the entry specifies a cast-after-sum addition. To generate the replacement code, specify that the code generator use the header and source files `myAdd.h` and `myAdd.c`.

```
setTflCOperationEntryParameters(tableEntry, ...
    'EntryInfoAlgorithm', 'RTW_CAST_AFTER_OP', ...
    'ImplementationHeaderFile', 'myAdd.h', ...
    'ImplementationSourceFile', 'myAdd.c'));
```

Add the entry to the table.

```
addEntry(crTable, tableEntry);
```

### Replacement Entry for Fixed-Point Operator With Same Slope Across Types

Create a table definition file that contains a function definition.

```
function crTable = crl_table_intaddfcn()
```

Within the function body, create the code replacement table.

```
crTable = RTW.TflTable;
```

Create a table entry for a signed fixed-point addition operation requiring the same slope across types.

```
tableEntry = createCRLEntry(crTable, ...
    'fixdt(1,16,~,0) y1 = fixdt(1,16,~,0) u1 + fixdt(1,16,~,0) u2', ...
    'int16 y1 = myAdd(int16 u1, int16 u2)');
```

Set entry parameters. Set algorithm parameters for a cast-after-sum addition and saturation and rounding modes. To generate the replacement code, specify that the code generator use the header and source files `myIntAdd.h` and `myIntAdd.c`.

```
setTflCOperationtionEntryParameters(tableEntry, ...
    'EntryInfoAlgorithm', 'RTW_CAST_AFTER_OP', ...
    'SaturationMode', 'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingMode', 'RTW_ROUND_SIMPLEST', ...
    'ImplementationHeaderFile', 'myIntAdd.h', ...
    'ImplementationSourceFile', 'myIntAdd.c');
```

Add the entry to the table.

```
addEntry(crTable, tableEntry);
```

#### Replacement Entry That Assumes Implementation and Conceptual Argument Data Types Are the Same

Create a table definition file that contains a function definition.

```
function crTable = crl_table_sinfcn()
```

Within the function body, create the code replacement table.

```
crTable = RTW.TflTable;
```

Create a table entry for a `sin` function, where the implementation arguments are the same as the conceptual arguments.

```
tableEntry = createCRLEntry(crTable, ...
    'double y1 = sin(double u1)', ...
    'y1 = mySin(u1)';
```

Set entry parameters. To generate the replacement code, specify that the code generator use the header and source files `mySin.h` and `mySin.c`.

```
setTflCFunctionEntryParameters(tableEntry, ...
    'ImplementationHeaderFile', 'mySin.h', ...
    'ImplementationSourceFile', 'mySin.c');
```

Add the entry to the table.

```
addEntry(crTable, tableEntry);
```

- “Define Code Replacement Mappings”
- “Code You Can Replace from MATLAB Code”
- “Code You Can Replace From Simulink Models”

## Input Arguments

**`crTable` — Code replacement table**
object

Table that stores one or more code replacement entries, each representing a potential replacement for a function or operator. Each entry maps a conceptual representation of a function or operator to an implementation representation and priority.

**`conceptualSpecification` — Conceptual specification**
character vector

Representation of the name or symbol and conceptual input and output arguments for a function or operator that the software replaces, specified as a character vector. Conceptual arguments observe naming conventions ('y1', 'u1', 'u2', ...) and data types familiar to the code generator. Use the syntax table in “Description” on page 1-152 to determine the syntax to use for your conceptual argument specification.

Example: `'double y1 = sin(double u1)'`

Example: `'int16 y1 = int16 u1 + int16 u2'`

**`implementationSpecification` — Implementation specification**
character vector

Representation of the name and implementation input and output arguments for a C or C++ replacement function, specified as a character vector. Implementation arguments observe C/C++ name and data type specifications. Use the syntax table in “Description” on page 1-152 to determine the syntax for your implementation argument specification.

Example: `'double y1 = my_sin(double u1)'`

Example: `'int16 y1 = myAdd(int16 u1, int16 u2)'`

## Output Arguments

**`tableEntry` — Code replacement table entry**
object

Code replacement table entry that represents a potential code replacement for a function or operator, returned as an object. Maps the conceptual representation of a function or operator, `conceptualSpecification`, to the C/C++ implementation representation, `implementationSpecification`.

## See Also

`RTW.TflTable` | `addEntry` | `setTflCFunctionEntryParameters`

### Topics

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

**Introduced in R2015a**

# createToleranceFile

**Class:** cgv.CGV
**Package:** cgv

Create file correlating tolerance information with signal names

## Syntax

`cgvObj.createToleranceFile(file_name , signal_list, tolerance_list)`

## Description

`cgvObj.createToleranceFile(file_name , signal_list, tolerance_list)` creates a MATLAB file, named `file_name`, containing the tolerance specification for each output signal name in `signal_list`. Each signal name in the `signal_list` corresponds to the same location of a parameter name and value pair in the `tolerance_list`.

## Input Arguments

### `file_name`

Name for the file containing the tolerance specification for each signal. Use this file as input to `cgv.CGV.compare` and `cgv.Batch.addTest`.

### `signal_list`

A cell array of character vectors, where each vector is a signal name for data from the model. Use `getSavedSignals` to view the list of available signal names in the output data. `signal_list` can contain an individual signal or multiple signals. The syntax for an individual signal name is:

```
signal_list = {'log_data.subsystem_name.Data(:,1)'}
```

The syntax for multiple signal names is:

```
signal_list = {'log_data.block_name.Data(:,1)',...
'log_data.block_name.Data(:,2)',...
'log_data.block_name.Data(:,3)',...
'log_data.block_name.Data(:,4)'};
```

To specify a global tolerance for the signals, include the reserved signal name, `'global_tolerance'`, in `signal_list`. Assign a global tolerance value in the associated `tolerance_list`. If `signal_list` contains other signals, their associated tolerance value overrides the global tolerance value. In this example, the global tolerance is a relative tolerance of `0.02`.

```
signal_list = {'global_tolerance',...
'log_data.block_name.Data(:,1)',...
'log_data.block_name.Data(:,2)'};

tolerance_list = {{'relative', 0.02},...
    {'relative', 0.015},{'absolute', 0.05}};
```

---

**Note** If a model component contains a space or newline character, MATLAB adds parentheses and a single quote to the name of the component. For example, if the signal name has a space, `'block name'`, MATLAB displays the signal name as:

```
log_data.('block name').Data(:,1)
```

To use the signal name as input to a CGV function, `'block name'` must have two single quotes in the `signal_list`. For example:

```
signal_list = {'log_data.(''block name'').Data(:,1)'}
```

---

#### `tolerance_list`

Cell array of cell arrays. Each element of the outer cell array is a cell array containing a parameter name and value pair for the type of tolerance and its value. Possible parameter names are `'absolute'` | `'relative'` | `'function'`. There is a one-to-one mapping between each parameter name and value pair in the `tolerance_list` and a signal name in the `signal_list`. For example, a `tolerance_list` for a `signal_list` containing four signals might look like the following:

```
tolerance_list = {{'relative', 0.02},{'absolute', 0.06},...
    {'relative', 0.015},{'absolute', 0.05}};
```

## See Also

### Topics
"Verify Numerical Equivalence with CGV"

# crossReleaseExport

Export generated model code for cross-release reuse

## Syntax

```
artifactsFolder = crossReleaseExport(buildFolder)
artifactsFolder = crossReleaseExport(buildFolder,
'ExportedArtifactsFolder', parentArtifactsFolder)
artifactsFolder = crossReleaseExport(buildFolder, 'Prompt',
promptToStartRelease)
```

## Description

`artifactsFolder = crossReleaseExport(buildFolder)` processes generated model component code in `buildFolder` to create cross-release code artifacts. These artifacts are required if you want to reuse the generated code in a newer release. The function places the artifacts in a folder *`modelComponent_Release`* and returns the full folder path.

The function supports the export of only C code that is produced by the code generator.

`artifactsFolder = crossReleaseExport(buildFolder, 'ExportedArtifactsFolder', parentArtifactsFolder)` creates a subfolder of cross-release export artifacts in `parentArtifactsFolder`.

`artifactsFolder = crossReleaseExport(buildFolder, 'Prompt', promptToStartRelease)` specifies whether a prompt is produced before a previous release is started.

## Examples

### Export Generated Code

This example shows how to export generated model component code created in a previous release.

Register the previous release with `sharedCodeMATLABVersions`.

```
[registeredMATLABs, installationFolders] = sharedCodeMATLABVersions;
requiredVersion = 'R2015b';
typicalPath = 'C:\Program Files\MATLAB';

if isempty(registeredMATLABs) || ~any(strcmp(requiredVersion, registeredMATLABs))
  versionFolder = fullfile(typicalPath, requiredVersion);
  sharedCodeMATLABVersions('Folder', versionFolder);
end
```

Create cross-release code artifacts.

```
folderPath = 'C:\myWorkFolder';
artifactsFolder = crossReleaseExport(fullfile(folderPath,'P1_ert_rtw'));
```

The returned value is the full path to the artifacts folder, which the `crossReleaseImport` function requires.

- “Cross-Release Shared Utility Code Reuse”
- “Cross-Release Code Integration”

## Input Arguments

**`buildFolder` — Build folder**
character vector

Path to the folder with generated model component code.

**`parentArtifactsFolder` — Parent artifacts folder**
character vector

Path to a parent folder that contains cross-release export artifact subfolders.

**`promptToStartRelease` — Prompt**
true (default) | false

- `true` - Prompt produced before start of previous release.
- `false` - Previous release started without prompt.

Data Types: `logical`

## Output Arguments

**`artifactsFolder` — Artifacts folder**
character vector

File path to the folder with cross-release artifacts, which the `crossReleaseImport` function requires.

## See Also

`crossReleaseImport` | `sharedCodeMATLABVersions` | `sharedCodeUpdate`

### Topics

"Cross-Release Shared Utility Code Reuse"
"Cross-Release Code Integration"

**Introduced in R2016b**

# crossReleaseImport

Import generated model code from a previous release as SIL or PIL blocks

## Syntax

```
blockHandle = crossReleaseImport(artifactsFolder,
configSetOrModel,'SimulationMode',mode)
blockHandle = crossReleaseImport(artifactsFolder,
configSetOrModel,'SimulationMode',mode,'ConfigParams',
additionalParameterList)
blockHandle = crossReleaseImport(artifactsFolder,
configSetOrModel,'SimulationMode',mode,'CodeLocation',anchorFolder)
blockHandle = crossReleaseImport(artifactsFolder,
configSetOrModel,'SimulationMode',mode,'DataDictionary',
dictionaryFile)
blockHandle = crossReleaseImport(artifactsFolder,
configSetOrModel,'SimulationMode',mode,'OriginalPaths',
originalPaths,'ReplacementPaths',replacementPaths)
blockHandle = crossReleaseImport(artifactsFolder,
configSetOrModel,'SimulationMode',mode,'SFunctionName',
sFunctionName)
```

## Description

`blockHandle = crossReleaseImport(artifactsFolder, configSetOrModel,'SimulationMode',mode)` uses the cross-release artifacts in `artifactsFolder` to import previously generated model component code into the current release. The function imports the code as a software-in-the-loop (SIL) or processor-in-the-loop (PIL) block and returns the numeric handle of the block. The function displays the block in a new model window.

You can replace the model component in an existing model with the SIL or PIL block. Or, you can use the SIL or PIL block as a component in a new model. When you run a simulation or build the model, the model component uses generated code from the previous release.

To build a SIL or PIL block, the function by default uses the following parameters of the Simulink model specified by `configSet`:

- `SystemTargetFile`
- `Toolchain` or `TemplateMakefile`
- `ExistingSharedCode`
- `PortableWordSizes`
- `TargetLang`
- `TargetLangStandard`
- `TargetLibSuffix`
- `ModelReferenceNumInstancesAllowed`
- **Hardware Implementation** pane parameters

`blockHandle = crossReleaseImport(artifactsFolder, configSetOrModel,'SimulationMode',mode,'ConfigParams', additionalParameterList)` uses additional configuration parameters for building the SIL or PIL block.

By default, the function assumes that the generated code resides in the build folder (Simulink) that you specified for `crossReleaseExport` when creating the cross-release artifacts. If you relocate the generated code, use an anchor folder and maintain the original code folder names and structure. For example:

- For top-model code, relocate *codeGenerationFolder*/*modelName*`_ert_rtw` to *anchorFolder*/*modelName*`_ert_rtw`.
- For model reference code, relocate *codeGenerationFolder*/`slprj/ert/`*referencedModelName* to *anchorFolder*/`slprj/ert/`*referencedModelName*.
- For subsystem code, relocate *codeGenerationFolder*/*subsystemName*`_ert_rtw` to *anchorFolder*/*subsystemName*`_ert_rtw`.

`blockHandle = crossReleaseImport(artifactsFolder, configSetOrModel,'SimulationMode',mode,'CodeLocation',anchorFolder)` uses the same cross-release artifacts to import the relocated generated code.

`blockHandle = crossReleaseImport(artifactsFolder, configSetOrModel,'SimulationMode',mode,'DataDictionary', dictionaryFile)` imports generated code that uses data types specified by a data dictionary. If `configSetOrModel` is a model associated with a data dictionary, you do not

have to specify the name-value pair. By default, the function identifies and uses the data dictionary when it imports the generated code. If you specify a name-value pair, the data dictionary that you specify takes precedence over the default data dictionary.

`blockHandle = crossReleaseImport(artifactsFolder, configSetOrModel,'SimulationMode',mode,'OriginalPaths', originalPaths,'ReplacementPaths',replacementPaths)` imports generated model code with relocated custom code or modified include paths.

`blockHandle = crossReleaseImport(artifactsFolder, configSetOrModel,'SimulationMode',mode,'SFunctionName', sFunctionName)` names the generated SIL or PIL block `sFunctionName_sil` or `sFunctionName_pil`. Use the `sFunctionName` argument if the default block name produces associated MATLAB identifiers that are longer than 63 characters.

## Examples

### Import Generated Code from Previous Release

This example shows how to import generated model code from a previous release.

Specify the location of the cross-release artifacts folder.

```
artifactsFolder = fullfile(pwd,'R2015bWork', 'P1_R2015b');
```

Import code for the integration model `Controller`.

```
crossReleaseImport(artifactsFolder,'Controller','SimulationMode','SIL');
```

The function displays a SIL block in a new Simulink editor window.

- “Cross-Release Shared Utility Code Reuse”
- “Cross-Release Code Integration”

## Input Arguments

**`artifactsFolder` — Artifacts folder**
character vector

Folder with cross-release artifacts created when you run the `crossReleaseExport` command in a previous release.

**`configSetOrModel` — Configuration object or model**
`Simulink.ConfigSet`|character vector

A configuration set or Simulink model on the MATLAB path.

**`mode` — Block mode**
`'SIL'` | `'PIL'` | `{'SIL','PIL'}`

Simulation mode for block with imported code:

- `'SIL'` — Create SIL block.
- `'PIL'` — Create PIL block.
- `{'SIL','PIL'}` — Create SIL and PIL blocks.

**`additionalParameterList` — Additional parameters**
cell array of character vectors

Additional parameters for building the SIL or PIL block.

**`anchorFolder` — Anchor folder**
character vector

Path to an anchor folder. This folder contains the generated code folder that you relocated after creating cross-release artifacts.

**`dictionaryFile` — Dictionary file**
character vector

Data dictionary that specifies data types used by the generated code.

**`originalPaths` — Original custom code folders or include paths**
character vector | cell array of character vectors | string array

Folder or include paths for original custom code. Must correspond to `replacementPaths`.

**`replacementPaths` — Replacement custom code folders or include paths**
character vector | cell array of character vectors | string array

Folder or include paths for relocated custom code code. Must correspond to `originalPaths`.

**`sFunctionName` — Name for SIL or PIL block**
character vector | cell array of character vectors | string array

Specify name for SIL or PIL block that contains generated code from previous release. If the default block name produces associated MATLAB identifiers that are longer than 63 characters, use this argument to specify a shorter block name.

## Output Arguments

**`blockHandle` — Numeric handle of a block**
double|array of doubles

Numeric handle of a block. Returned as a double if `mode` is `'SIL'` or `'PIL'`. Returned as an array of doubles if `mode` is `{'SIL','PIL'}`.

## See Also

`crossReleaseExport` | `sharedCodeUpdate`

### Topics

"Cross-Release Shared Utility Code Reuse"
"Cross-Release Code Integration"

**Introduced in R2016b**

# dir

Files and folders in current IDE window

## Syntax

```
dir(IDE_Obj)
d = dir(IDE_Obj)
```

## IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

## Description

`dir(IDE_Obj)` lists the files and folders in the IDE working folder, where `IDE_Obj` is the object that references the IDE. `IDE_Obj` can be either a single object, or a vector of objects. When `IDE_Obj` is a vector, `dir` returns the files and folders referenced by each object.

`d = dir(IDE_Obj)` returns the list of files and folders as an M-by-1 structure in `d` with the fields for each file and folder shown in the following table.

| Field Name | Description |
| --- | --- |
| `name` | Name of the file or folder. |
| `date` | Date of most recent file or folder modification. |
| `bytes` | Size of the file in bytes. Folders return 0 for the number of bytes. |
| `isdirectory` | 0 if it is a file, 1 if it is a folder. |

| Field Name | Description |
|---|---|
| datenum | Code Composer Studio IDE also returns the modification date as a MATLAB serial date number. |

To view the entries in structure d, use an index in the syntax at the MATLAB prompt, as shown by the following examples.

- d(3) returns the third element in the structure.
- d(10) returns the tenth element in the structure d.
- d(4).date returns the date field value for the fourth structure element.

## See Also

cd | open

**Introduced in R2011a**

# disable

Disable RTDX interface, specified channel, or RTDX channels

---

**Note** Support for `disable` on C5000 processors will be removed in a future version.

---

## Syntax

```
disable(rx,'channel')
disable(rx,'all')
disable(rx)
```

## IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

## Description

`disable(rx,'channel')` disables the open channel specified by the character vector *`channel`*, for `rx`. Input argument `rx` represents the RTDX portion of the associated link to the IDE.

`disable(rx,'all')` disables the open channels associated with `rx`.

`disable(rx)` disables the RTDX interface for `rx`.

### Important Requirements for Using disable

On the processor side, `disable` depends on RTDX to disable channels or the interface. To use `disable`, meet the following requirements:

1. The processor must be running a program.
2. You enabled the RTDX interface.
3. Your processor program polls periodically.

## Examples

When you have opened and used channels to communicate with a processor, disable the channels and RTDX before ending your session. Use `disable` to switch off open channels and disable RTDX, as follows:

```
disable(rtdx(IDE_Obj),'all') % Disable the open RTDX channels.
disable(rtdx(IDE_Obj))       % Disable RTDX interface.
```

## See Also

`close` | `enable` | `open`

**Introduced in R2011a**

# display (IDE Object)

Properties of IDE handle

## Syntax

```
display(IDE_Obj)
```

## IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

## Description

`display(IDE_Obj)` displays the properties and property values of the IDE handle`IDE_Obj`.

For example, after you creating `IDE_Obj` with a constructor, using the `display` method with `IDE_Obj` returns a set of properties and values:

```
display(IDE_Obj)

IDE Object:
  Property1        : valuea
  Property2        : valueb
  Property3              : valuec
  Property4        : valued
```

## See Also

`get`

**Introduced in R2011b**

# display

Generate message that describes how to open code execution profiling report

## Syntax

```
myExecutionProfile
myExecutionProfile.display
```

## Description

*`myExecutionProfile`* or *`myExecutionProfile`*`.display` generates a message that describes how you can open the code execution profiling report.

*`myExecutionProfile`* is a workspace variable, specified through the configuration parameter `CodeExecutionProfileVariable` and generated by a simulation.

## See Also

`report`

### Topics

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

**Introduced in R2011a**

# displayReport

**Class:** cgv.Config
**Package:** cgv

Display results of comparing configuration parameter values

## Syntax

*cfgObj*.displayReport()

## Description

*cfgObj*.displayReport() displays the results at the MATLAB Command Window of comparing the configuration parameter values for the model with the values that the object recommends. *cfgObj* is a handle to a cgv.Config object.

## See Also

### Topics

"Verify Numerical Equivalence Between Two Modes of Execution of a Model"

# coder.MATLABCodeTemplate.emitSection

**Class:** coder.MATLABCodeTemplate
**Package:** coder

Emit comments for template section

## Syntax

```
sectionComments = emitSection(sectionName,isCPPComment)
```

## Description

`sectionComments = emitSection(sectionName,isCPPComment)` emits comments for the code template section that `sectionName` specifies. If `isCPPComment` is `true`, `emitSection` uses C++ style comments. If `emitSection` is `false`, it uses C style comments. Use `emitSection` to preview banners before you generate code. Before invoking `emitSection` to emit the banner for a template section, you must set the values for all tokens used in that section.

## Input Arguments

**`sectionName` — Name of templates section**
character vector

Name of template section specified as one of the following values:

| | |
|---|---|
| `'FileBanner'` | `'VariableDeclarationsBanner'` |
| `'FunctionBanner'` | `'VariableDefinitionsBanner'` |
| `'SharedUtilityBanner'` | `'FunctionDeclarationsBanner'` |
| `'FileTrailer'` | `'FunctionDefinitionsBanner'` |
| `'IncludeFilesBanner'` | `'CustomSourceCodeBanner'` |
| `'TypeDefinitionsBanner'` | `'CustomHeaderCodeBanner'` |

'NamedConstantsBanner'

**isCPPComment — C++ comment style flag**
true | false

Specify `true` for C++ style comments. Specify `false` for C style comments.

## Output Arguments

**sectionComments — Comments for template section**
character vector

Comments for the specified section, returned as a character vector.

## Examples

### Emit File Banner from Default Template

This example shows how to set the `FileName` token value and emit the default file banner.

Create a `coder.MATLABCodeTemplate` object from the default template.

```
newObj = coder.MATLABCodeTemplate
```

Set the `FileName` token value.

```
fileN = 'myfilename.c';
newObj.setTokenValue('FileName', fileN)
```

Emit the file banner.

```
newObj.emitSection('FileBanner', false)
```

The `emitSection` method generates the file banner replacing the `FileName` token with the file name that you specified. It replaces the `MATLABCoderVersion` token with the current MATLAB Coder version number. It replaces the `SourceGeneratedOn` token with the time stamp.

```
/*
 * File: myfilename.c
```

```
 *
 * MATLAB Coder version            : 2.7
 * C/C++ source code generated on  : 07-Apr-2014 17:43:32
 */
```

#### Emit Include Files Banner from Custom Template

This example shows how to create and modify a custom code generation template (CGT) file. It shows how to emit the include files section banner from the custom CGT file.

Create a local copy of the default CGT file for MATLAB Coder. Name it `myCGTFile.cgt`.

In your local copy of the CGT File, in the `IncludeFilesBanner` open tag, change the style to `"box"`.

```
<IncludeFilesBanner style="box">
Include Files
</IncludeFilesBanner>
```

Create a `MATLABCodeTemplate` object from your custom CGT file.

```
CGTFile = 'myCGTFile.cgt';
newObj= coder.MATLABCodeTemplate(CGTFile);
```

Emit the include files section banner using C++ style comments.

```
newObj.emitSection('IncludeFilesBanner', true)
```

The `emitSection` method generates the include files section banner using the box style with C++ style comments.

```
///////////////////////////////////////////////////////////////////////////////////
// Include Files                                                                 //
///////////////////////////////////////////////////////////////////////////////////
```

- “Generate Custom File and Function Banners for C/C++ Code”

## See Also

`coder.MATLABCodeTemplate.getCurrentTokens` | `coder.MATLABCodeTemplate.getTokenValue` | `coder.MATLABCodeTemplate.setTokenValue`

## Topics

"Generate Custom File and Function Banners for C/C++ Code"
"Code Generation Template Files for MATLAB Code"

# enable

Enable RTDX interface, specified channel, or RTDX channels

**Note** Support for `enable` on C5000 processors will be removed in a future version.

## Syntax

```
enable(rx,'channel')
enable(rx,'all')
enable(rx)
```

## IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

## Description

`enable(rx,'channel')` enables the open channel specified by the character vector *`channel`*, for RTDX link `rx`. The input argument `rx` represents the RTDX portion of the associated link to the IDE.

`enable(rx,'all')` enables the open channels associated with `rx`.

`enable(rx)` enables the RTDX interface for `rx`.

### Important Requirements for Using enable

On the processor side, `enable` depends on RTDX to enable channels. To use `enable`, meet the following requirements:

1. The processor must be running a program when you enable the RTDX interface. When the processor is not running, the state defaults to disabled.
2. Enable the RTDX interface before you enable individual channels.
3. Channels must be open.
4. Your processor program must poll periodically.
5. Using code in the program running on the processor to enable channels overrides the default disabled state of the channels.

## Examples

To use channels to RTDX, you must both open and enable the channels:

```
IDE_Obj = ticcs; % Create a new connection to the IDE.
enable(rtdx(IDE_Obj)) % Enable the RTDX interface.
open(rtdx(IDE_Obj),'inputchannel','w') % Open a channel for sending
                                 % data to the processor.
enable(rtdx(IDE_Obj),'inputchannel') % Enable the channel so you can use
                               % it.
```

## See Also

`disable` | `open`

**Introduced in R2011a**

# enableCPP

Enable C++ support for function entry in code replacement table

## Syntax

`enableCPP(hEntry)`

## Description

`enableCPP(hEntry)` enables C++ support for a function entry in a code replacement table. This support allows you to specify a C++ namespace for the implementation function defined in the entry (see the `setNameSpace` function).

When you register a code replacement library containing C++ function entries, you must specify the value `{'C++'}` for the `LanguageConstraint` property of the code replacement registry entry. For more information, see “Register Code Replacement Mappings”.

## Examples

### Enable C++ Support for Function Entry

This example shows how to use the `enableCPP` function to enable C++ support. Then, the example calls the `setNameSpace` function to set the namespace for the `sin` implementation function to `std`.

```
fcn_entry = RTW.TflCFunctionEntry;
fcn_entry.setTflCFunctionEntryParameters( ...
    'Key',                      'sin', ...
    'Priority',                 100, ...
    'ImplementationName',       'sin', ...
    'ImplementationHeaderFile', 'cmath' );
```

```
fcn_entry.enableCPP();
fcn_entry.setNameSpace('std');
```

## Input Arguments

**`hEntry` — Handle to a code replacement function entry**
handle

The *`hEntry`* is a handle to a code replacement function entry previously returned by *`hEntry`* `= RTW.TflCFunctionEntry` or *`hEntry = MyCustomFunctionEntry`*. The *`MyCustomFunctionEntry`* is a class derived from `RTW.TflCFunctionEntry`.

Example: `fcn_entry`

## See Also

`registerCPPFunctionEntry` | `setNameSpace`

### Topics

"Math Function Code Replacement"
"Define Code Replacement Mappings"
"Code You Can Replace from MATLAB Code"
"Code You Can Replace From Simulink Models"

**Introduced in R2010a**

# excludeCheck

**Class:** rtw.codegenObjectives.Objective
**Package:** rtw.codegenObjectives

Exclude checks

## Syntax

`excludeCheck(obj, checkID)`

## Description

`excludeCheck(obj, checkID)` excludes a check from the Code Generation Advisor when a user specifies the objective. When a user selects multiple objectives, if the user specifies an additional objective that includes this check as a higher priority objective, the Code Generation Advisor displays this check.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a code generation objective object previously created. |
| *checkID* | Unique identifier of the check that you exclude from the new objective. |

## Examples

Exclude the **Identify questionable code instrumentation (data I/O)** check from the objective.

```
excludeCheck(obj, 'mathworks.codegen.CodeInstrumentation');
```

## See Also

`Simulink.ModelAdvisor`

## Topics

“Create Custom Code Generation Objectives”
“About IDs” (Simulink)

# flush

Flush data or messages from specified RTDX channels

---

**Note** `flush` support for C5000 processors will be removed in a future version.

---

## Syntax

```
flush(rx,channel,num,timeout)
flush(rx,channel,num)
flush(rx,channel,[],timeout)
flush(rx,channel)
flush(rx,'all')
```

## IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

## Description

`flush(rx,`*`channel`*`,`*`num`*`,`*`timeout`*`)` removes *`num`* oldest data messages from the RTDX channel queue specified by *`channel`* in `rx`. To determine how long to wait for the function to complete, `flush` uses *`timeout`* (in seconds) rather than the global timeout period stored in `rx`. `flush` applies the timeout processing when it flushes the last message in the channel queue, because the flush function performs a read to advance the read pointer past the last message. Use this calling syntax only when you specify a channel configured for read access.

`flush(rx,`*`channel`*`,`*`num`*`)` removes the *`num`* oldest messages from the RTDX channel queue in `rx` specified by the character vector *`channel`*. `flush` uses the global timeout

period stored in `rx` to determine how long to wait for the process to complete. Compare this to the previous syntax that specifies the timeout period. Use this calling syntax only when you specify a channel configured for read access.

`flush(rx,`*`channel`*`,[],`*`timeout`*`)` removes the data messages from the RTDX channel queue specified by *`channel`* in `rx`. To determine how long to wait for the function to complete, `flush` uses *`timeout`* (in seconds) rather than the global timeout period stored in `rx`. `flush` applies the timeout processing when it flushes the last message in the channel queue, because `flush` performs a read to advance the read pointer past the last message. Use this calling syntax only when you specify a channel configured for read access.

`flush(rx,`*`channel`*`)` removes the pending data messages from the RTDX channel queue specified by *`channel`* in `rx`. Unlike the preceding syntax options, you use this statement to remove messages for both read-configured and write-configured channels.

`flush(rx,'all')` removes the data messages from the RTDX channel queues.

When you use `flush` with a write-configured RTDX channel, the code generator sends the messages in the write queue to the processor. For read-configured channels, `flush` removes one or more messages from the queue depending on the input argument *`num`* you supply and disposes of them.

## Examples

To show how to use `flush`, this example writes data to the processor over the input channel, then uses `flush` to remove a message from the read queue for the output channel:

```
IDE_Obj = ticcs;
rx = rtdx(IDE_Obj);
open(rx,'ichan','w');
enable(rx,'ichan');
open(rx,'ochan','r');
enable(rx,'ochan');
indata = 1:10;
writemsg(rx,'ichan',int16(indata));
flush(rx,'ochan',1);
```

Now flush the remaining messages from the read channel:

```
flush(rx,'ochan','all');
```

## See Also

`enable` | `open`

**Introduced in R2011a**

# getAlgorithmParameters

Examine algorithm parameter settings for lookup table function code replacement table entry

## Syntax

```
algParams = getAlgorithmParameters(tableEntry)
```

## Description

`algParams = getAlgorithmParameters(tableEntry)` returns the algorithm parameter settings for the lookup table function identified in the code replacement table entry `tableEntry`. If you call `getAlgorithmParameters` before using `setAlgorithmParameters`, `getAlgorithmParameters` lists the default parameter settings for the lookup table function.

## Examples

### Examine Default Parameter Settings for `prelookup` Table Entry

Create a code replacement table.

```
crTable = RTW.TflTable;
```

Create a table entry for a function.

```
tableEntry = RTW.TflCFunctionEntry;
```

Identify the table entry as an entry for the `prelookup` function.

```
setTflCFunctionEntryParameters(tableEntry, ...
    'Key', 'prelookup', ...
    'Priority', 100, ...
    'ImplementationName', 'myPrelookup');
```

Get the algorithm parameter settings for the `prelookup` function table entry.

```
algParams = getAlgorithmParameters(tableEntry)
```

```
algParams = 

  Prelookup with properties:

             ExtrapMethod: [1x1 coder.algorithm.parameter.ExtrapMethod]
                  RndMeth: [1x1 coder.algorithm.parameter.RndMeth]
        IndexSearchMethod: [1x1 coder.algorithm.parameter.IndexSearchMethod]
        UseLastBreakpoint: [1x1 coder.algorithm.parameter.UseLastBreakpoint]
    RemoveProtectionInput: [1x1 coder.algorithm.parameter.RemoveProtectionInput]
```

Examine the information for parameter `ExtrapMethod`.

```
algParams.ExtrapMethod
```

```
ans = 

  ExtrapMethod with properties:

       Name: 'ExtrapMethod'
    Options: {'Linear'  'Clip'}
    Primary: 1
      Value: {'Linear'}
```

Examine the information for parameter `RndMeth`.

```
algParams.RndMeth
```

```
ans = 

  RndMeth with properties:

       Name: 'RndMeth'
    Options: {1x7 cell}
    Primary: 0
      Value: {1x7 cell}
```

Examine the current `Value` setting.

```
algParams.RndMeth.Value
```

```
ans = 

  Columns 1 through 6

    'Ceiling'    'Convergent'    'Floor'    'Nearest'    'Round'    'Simplest'

  Column 7

    'Zero'
```

Examine the information for parameter `IndexSearchMethod`.

```
algParams.IndexSearchMethod
```

```
ans = 
```

```
  IndexSearchMethod with properties:

       Name: 'IndexSearchMethod'
    Options: {'Linear search'  'Binary search'  'Evenly spaced points'}
    Primary: 0
      Value: {'Binary search'  'Evenly spaced points'  'Linear search'}
```

Examine the information for parameter `UseLastBreakpoint`.

```
algParams.UseLastBreakpoint
```

```
ans =

  UseLastBreakpoint with properties:

       Name: 'UseLastBreakpoint'
    Options: {'off'  'on'}
    Primary: 0
      Value: {'off'  'on'}
```

Examine the information for parameter `RemoveProtectionInput`.

```
algParams.RemoveProtectionInput
```

```
ans =

  RemoveProtectionInput with properties:

       Name: 'RemoveProtectionInput'
    Options: {'off'  'on'}
    Primary: 0
      Value: {'off'  'on'}
```

### Examine Modified Parameter Setting for `lookup2D` Table Entry

Create a code replacement table.

```
crTable = RTW.TflTable;
```

Create a table entry for a function.

```
tableEntry = RTW.TflCFunctionEntry;
```

Identify the table entry as an entry for the `lookup2D` function.

```
setTflCFunctionEntryParameters(tableEntry, ...
    'Key', 'lookup2D', ...
    'Priority', 100, ...
    'ImplementationName', 'myLookup2D');
```

Get the algorithm parameter settings for the `lookup2D` function table entry.

```
algParams = getAlgorithmParameters(tableEntry)
```

```
algParams =
```

```
  Lookup with properties:

               InterpMethod: [1x1 coder.algorithm.parameter.InterpMethod]
               ExtrapMethod: [1x1 coder.algorithm.parameter.ExtrapMethod]
                    RndMeth: [1x1 coder.algorithm.parameter.RndMeth]
          IndexSearchMethod: [1x1 coder.algorithm.parameter.IndexSearchMethod]
          UseLastTableValue: [1x1 coder.algorithm.parameter.UseLastTableValue]
      RemoveProtectionInput: [1x1 coder.algorithm.parameter.RemoveProtectionInput]
  SaturateOnIntegerOverflow: [1x1 coder.algorithm.parameter.SaturateOnIntegerOverflow]
    SupportTunableTableSize: [1x1 coder.algorithm.parameter.SupportTunableTableSize]
            BPPower2Spacing: [1x1 coder.algorithm.parameter.BPPower2Spacing]
```

Display the possible index search method settings.

```
algParams.IndexSearchMethod.Options
```

```
ans = 

    'Linear search'    'Binary search'    'Evenly spaced points'
```

Display the current index search method setting.

```
algParams.IndexSearchMethod.Value
```

```
ans = 

    'Linear search'    'Binary search'    'Evenly spaced points'
```

By default, the parameter is set to the same value set.

Set the index search method to binary search.

```
algParams.IndexSearchMethod = 'Binary search';
```

Verify the modified parameter setting.

```
algParams.IndexSearchMethod.Value
```

```
ans = 

    'Binary search'
```

# Input Arguments

### `tableEntry` — Code replacement table entry for a lookup table function

object

Code replacement table entry that you previously created and represents a potential code replacement for a lookup table function. The entry must identify the lookup table function for which you are calling `getAlgorithmParameters`.

1. Create the entry. For example, call the function `RTW.TflCFunctionEntry`.

```
tableEntry = RTW.TflCFunctionEntry;
```

2. Specify the name of the lookup table function for which you created the entry. Use the `Key` parameter in a call to `setTflCFunctionEntryParameters`. The following function call specifies the lookup table function `prelookup`.

```
setTflCFunctionEntryParameters(tableEntry, ...
   'Key', 'prelookup', ...
   'Priority', 100, ...
   'ImplementationName', 'myPrelookup');
```

## Output Arguments

**`algParams` — Algorithm parameter settings for a lookup table function**
object

Algorithm parameter settings for the lookup table function identified with the `Key` parameter in `tableEntry`.

## See Also

`RTW.TflCFunctionEntry` | `RTW.TflTable` | `addEntry` | `setAlgorithmParameters` | `setTflCFunctionEntryParameters`

### Topics

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

**Introduced in R2015a**

# coder.dictionary.copy

**Package:** coder.dictionary

Copy code generation definitions between models and data dictionaries

## Syntax

`copy(sourceName,destinationName)`

## Description

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

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

To share code definitions between models, use a Simulink data dictionary as described in “Share Embedded Coder Dictionary Definition Between Models”. For general information about Embedded Coder Dictionaries and code generation definitions, see “Define Storage Classes, Memory Sections, and Function Templates for Software Architecture”.

## Examples

### Copy Code Definitions from Model to Model

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

Open the example model `rtwdemo_roll`.

```
rtwdemo_roll
```

Open the Embedded Coder Dictionary for the model. In the model, select **Code > C/C++ Code > Embedded Coder Dictionary**.

In the Embedded Coder Dictionary window, on the **Storage Classes** tab, create a storage class by clicking **Add**.

The new storage class is named `StorageClass1`.

Close the Embedded Coder Dictionary window.

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

Open the other model, `rtwdemo_rtwecintro`.

```
rtwdemo_rtwecintro
```

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

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

Open the Embedded Coder Dictionary for `rtwdemo_rtwecintro`.

In the Embedded Coder Dictionary window, on the **Storage Classes** tab, the storage class `StorageClass1` appears.

- “Define Storage Classes, Memory Sections, and Function Templates for Software Architecture”
- “Configure Default Code Generation for Categories of Model Data and Functions”

## Input Arguments

**`sourceName` — Source model file or data dictionary**
character vector

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

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

You do not need to specify the `.slx` file extension.

- A dictionary must be open in the Model Explorer, in the current folder, or on the MATLAB path.

  You must specify the `.sldd` file extension.

Example: `'myLoadedModel'`

Example: `'myDictionary.sldd'`

Data Types: `char`

**`destinationName` — Destination model file or data dictionary**
character vector

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

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

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

  You must specify the `.sldd` file extension.

Example: `'myLoadedModel'`

Example: `'myDictionary.sldd'`

Data Types: `char`

# See Also

Embedded Coder Dictionary

## Topics

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

**Introduced in R2018a**

# coder.dictionary.move

**Package:** coder.dictionary

Migrate code generation definitions between models and data dictionaries

## Syntax

`move(sourceName,destinationName)`

## Description

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

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

Use this function to:

- Move code generation definitions from a model to a Simulink data dictionary. For information about sharing code generation definitions between models by creating an Embedded Coder Dictionary in a data dictionary, see "Share Embedded Coder Dictionary Definition Between Models".
- In a hierarchy of referenced Simulink data dictionaries, move the Embedded Coder Dictionary from one data dictionary to another. In a hierarchy of referenced dictionaries, only one dictionary can store an Embedded Coder Dictionary.

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

## Examples

### Move Code Definitions from Model to Model

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

Open the example model `rtwdemo_roll`.

```
rtwdemo_roll
```

Open the Embedded Coder Dictionary for the model. In the model, select **Code** > **C/C++ Code** > **Embedded Coder Dictionary**.

In the Embedded Coder Dictionary window, on the **Storage Classes** tab, create a storage class by clicking **Add**.

The new storage class is named `StorageClass1`.

Close the Embedded Coder Dictionary window.

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

Open the other model, `rtwdemo_rtwecintro`.

```
rtwdemo_rtwecintro
```

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

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

Open the Embedded Coder Dictionary for `rtwdemo_rtwecintro`.

In the Embedded Coder Dictionary window, on the **Storage Classes** tab, the storage class `StorageClass1` appears. The storage class no longer exists in `rtwdemo_roll`.

- “Define Storage Classes, Memory Sections, and Function Templates for Software Architecture”

- “Configure Default Code Generation for Categories of Model Data and Functions”

## Input Arguments

**`sourceName` — Source model file or data dictionary**
character vector

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

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

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

  You must specify the `.sldd` file extension.

Example: `'myLoadedModel'`

Example: `'myDictionary.sldd'`

Data Types: `char`

**`destinationName` — Destination model file or data dictionary**
character vector

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

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

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

  You must specify the `.sldd` file extension.

Example: `'myLoadedModel'`

Example: `'myDictionary.sldd'`

Data Types: `char`

## See Also

Embedded Coder Dictionary

### Topics

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

**Introduced in R2018a**

# coder.dictionary.remove

**Package:** coder.dictionary

Remove Embedded Coder Dictionary from model or Simulink data dictionary

## Syntax

`remove(sourceName)`

## Description

`remove(sourceName)` removes the Embedded Coder Dictionary, including all code generation definitions such as storage classes and function customization templates, from the model or Simulink data dictionary identified by `sourceName`.

Use this function to:

- Remove the Embedded Coder Dictionary from a model so that the model can use the Embedded Coder Dictionary in a Simulink data dictionary instead. For information about sharing code generation definitions between models by creating an Embedded Coder Dictionary in a data dictionary, see “Share Embedded Coder Dictionary Definition Between Models”.
- In a hierarchy of referenced Simulink data dictionaries, remove the Embedded Coder Dictionary from a data dictionary. In a hierarchy of referenced dictionaries, only one dictionary can store an Embedded Coder Dictionary.

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

## Examples

### Remove Embedded Coder Dictionary from Model File

When you open the Embedded Coder Dictionary window for a model (see “Open the Embedded Coder Dictionary” on page 16-0 ), Simulink creates an Embedded Coder

Dictionary in the model file. In this example, open the Embedded Coder Dictionary window for the example model `rtwdemo_roll`, create a code generation definition (a storage class), then delete the Embedded Coder Dictionary from the model.

At the command prompt, open the model.

```
rtwdemo_roll
```

In the model, select **Code** > **C/C++ Code** > **Embedded Coder Dictionary**.

The Embedded Coder Dictionary window opens, showing the contents of the new Embedded Coder Dictionary in `rtwdemo_roll`. The dictionary contains storage classes.

Click the **Add** button to create a new storage class, whose default name is `StorageClass1`.

Close the Embedded Coder Dictionary window.

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

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

Now, the model file does not contain an Embedded Coder Dictionary or any code generation definitions (such as storage classes).

- “Define Storage Classes, Memory Sections, and Function Templates for Software Architecture”
- “Configure Default Code Generation for Categories of Model Data and Functions”

## Input Arguments

### `sourceName` — Target model file or data dictionary

character vector

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

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

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

You must specify the `.sldd` file extension.

Example: `'myLoadedModel'`

Example: `'myDictionary.sldd'`

Data Types: `char`

## Tip

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

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

## See Also

Embedded Coder Dictionary

### Topics

“Define Storage Classes, Memory Sections, and Function Templates for Software Architecture”
“Configure Default Code Generation for Categories of Model Data and Functions”

**Introduced in R2018a**

# getArgCategory

**Class:** RTW.ModelCPPArgsClass
**Package:** RTW

Get argument category for Simulink model port from model-specific C++ class interface

## Syntax

`category = getArgCategory(obj, portName)`

## Description

`category = getArgCategory(obj, portName)` gets the category — `'Value'`, `'Pointer'`, or `'Reference'` — of the argument corresponding to a specified Simulink model inport or outport from a specified model-specific C++ class interface.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a model-specific C++ class interface control object, such as a handle previously returned by `obj = RTW.getClassInterfaceSpecification (modelName)`. |
| *portName* | Character vector specifying the name of an inport or outport in your Simulink model. |

## Output Arguments

| | |
|---|---|
| *category* | Character vector specifying the argument category — `'Value'`, `'Pointer'`, or `'Reference'` — for the specified Simulink model port. |

## Alternatives

To view argument categories in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments step method` view of this dialog box, click the **Get Default Configuration** button to display step method argument categories. For more information, see “Configure Step Method for Your Model Class”.

## See Also

### Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

# getArgCategory

**Class:** RTW.ModelSpecificCPrototype
**Package:** RTW

Get argument category for Simulink model port from model-specific C function prototype

## Syntax

`category = getArgCategory(obj, portName)`

## Description

`category = getArgCategory(obj, portName)` gets the category, `'Value'` or `'Pointer'`, of the argument corresponding to a specified Simulink model inport or outport from a specified model-specific C function prototype.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a model-specific C prototype function control object previously returned by `obj = RTW.getFunctionSpecification (modelName)`. |
| *portName* | Character vector specifying the name of an inport or outport in your Simulink model. |

## Output Arguments

| | |
|---|---|
| *category* | Character vector specifying the argument category, `'Value'` or `'Pointer'`, for the specified Simulink model port. |

## Alternatives

Click the **Get Default Configuration** button in the Model Interface dialog box to get argument categories. See "Configure C Step Function Arguments".

## See Also

### Topics

"Customize Generated C Function Interfaces"

# getArgName

**Class:** RTW.ModelCPPArgsClass
**Package:** RTW

Get argument name for Simulink model port from model-specific C++ class interface

## Syntax

`argName = getArgName(obj, portName)`

## Description

`argName = getArgName(obj, portName)` gets the argument name corresponding to a specified Simulink model inport or outport from a specified model-specific C++ class interface.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a model-specific C++ class interface control object, such as a handle previously returned by `obj = RTW.getClassInterfaceSpecification (modelName)`. |
| *portName* | Character vector specifying the name of an inport or outport in your Simulink model. |

## Output Arguments

| | |
|---|---|
| *argName* | Character vector specifying the argument name for the specified Simulink model port. |

## Alternatives

To view argument names in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments step method` view of this dialog box, click the **Get Default Configuration** button to display step method argument names. For more information, see “Configure Step Method for Your Model Class”.

## See Also

### Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

# getArgName

**Class:** RTW.ModelSpecificCPrototype
**Package:** RTW

Get argument name for Simulink model port from model-specific C function prototype

## Syntax

`argName = getArgName(obj, portName)`

## Description

`argName = getArgName(obj, portName)` gets the argument name corresponding to a specified Simulink model inport or outport from a specified model-specific C function prototype.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a model-specific C prototype function control object previously returned by `obj = RTW.getFunctionSpecification(modelName)`. |
| *portName* | Character vector specifying the name of an inport or outport in your Simulink model. |

## Output Arguments

| | |
|---|---|
| *argName* | Character vector specifying the argument name for the specified Simulink model port. |

## Alternatives

Click the **Get Default Configuration** button in the Model Interface dialog box to get argument names. See “Configure C Step Function Arguments”.

## See Also

### Topics

“Customize Generated C Function Interfaces”

# getArgPosition

**Class:** RTW.ModelCPPArgsClass
**Package:** RTW

Get argument position for Simulink model port from model-specific C++ class interface

## Syntax

`position = getArgPosition(obj, portName)`

## Description

`position = getArgPosition(obj, portName)` gets the position — 1 for first, 2 for second, etc. — of the argument corresponding to a specified Simulink model inport or outport from a specified model-specific C++ class interface.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a model-specific C++ class interface control object, such as a handle previously returned by `obj = RTW.getClassInterfaceSpecification (modelName)`. |
| *portName* | Character vector specifying the name of an inport or outport in your Simulink model. |

## Output Arguments

| | |
|---|---|
| *position* | Integer specifying the argument position — 1 for first, 2 for second, etc. — for the specified Simulink model port. Without an argument for the specified port, the function returns 0. |

## Alternatives

To view argument positions in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments step method` view of this dialog box, click the **Get Default Configuration** button to display step method argument positions. For more information, see “Configure Step Method for Your Model Class”.

## See Also

### Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

# getArgPosition

**Class:** RTW.ModelSpecificCPrototype
**Package:** RTW

Get argument position for Simulink model port from model-specific C function prototype

## Syntax

`position = getArgPosition(obj, portName)`

## Description

`position = getArgPosition(obj, portName)` gets the position — 1 for first, 2 for second, etc. — of the argument corresponding to a specified Simulink model inport or outport from a specified model-specific C function prototype.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a model-specific C prototype function control object previously returned by `obj = RTW.getFunctionSpecification (modelName)`. |
| *portName* | Character vector specifying the name of an inport or outport in your Simulink model. |

## Output Arguments

| | |
|---|---|
| *position* | Integer specifying the argument position — 1 for first, 2 for second, etc. — for the specified Simulink model port. Without an argument for the specified port, the function returns 0. |

## Alternatives

Click the **Get Default Configuration** button in the Model Interface dialog box to get argument positions. See "Configure C Step Function Arguments".

## See Also

### Topics

"Customize Generated C Function Interfaces"

# getArgQualifier

**Class:** RTW.ModelCPPArgsClass
**Package:** RTW

Get argument type qualifier for Simulink model port from model-specific C++ class interface

## Syntax

`qualifier = getArgQualifier(obj, portName)`

## Description

`qualifier = getArgQualifier(obj, portName)` gets the type qualifier — `'none'`, `'const'`, `'const *'`, `'const * const'`, or `'const &'` — of the argument corresponding to a specified Simulink model inport or outport from a specified model-specific C++ class interface.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a model-specific C++ class interface control object, such as a handle previously returned by `obj = RTW.getClassInterfaceSpecification (modelName)`. |
| *portName* | Character vector specifying the name of an inport or outport in your Simulink model. |

## Output Arguments

| | |
|---|---|
| *qualifier* | Character vector specifying the argument type qualifier — `'none'`, `'const'`, `'const *'`, `'const * const'`, or `'const &'` — for the specified Simulink model port. |

## Alternatives

To view argument qualifiers in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments step method` view of this dialog box, click the **Get Default Configuration** button to display step method argument qualifiers. For more information, see “Configure Step Method for Your Model Class”.

## See Also

### Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

# getArgQualifier

**Class:** RTW.ModelSpecificCPrototype
**Package:** RTW

Get argument type qualifier for Simulink model port from model-specific C function prototype

## Syntax

`qualifier = getArgQualifier(obj, portName)`

## Description

`qualifier = getArgQualifier(obj, portName)` gets the type qualifier — `'none'`, `'const'`, `'const *'`, or `'const * const'`— of the argument corresponding to a specified Simulink model inport or outport from a specified model-specific C function prototype.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a model-specific C prototype function control object previously returned by `obj = RTW.getFunctionSpecification(modelName)`. |
| *portName* | Character vector specifying the name of an inport or outport in your Simulink model. |

## Output Arguments

| | |
|---|---|
| *qualifier* | Character vector specifying the argument type qualifier — `'none'`, `'const'`, `'const *'`, or `'const * const'`— for the specified Simulink model port. |

## Alternatives

Click the **Get Default Configuration** button in the Model Interface dialog box to get argument qualifiers. See “Configure C Step Function Arguments”.

## See Also

### Topics

“Customize Generated C Function Interfaces”

# getbuildopt

Generate structure of build tools and options

## Syntax

```
bt = getbuildopt(IDE_Obj)
cs = getbuildopt(IDE_Obj,file)
```

## IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

## Description

`bt = getbuildopt(IDE_Obj)` returns an array of structures in `bt`. Each structure includes an entry for each defined build tool. This list of build tools comes from the active project and active build configuration. Included in the structure is a character vector that describes the command-line tool options. `bt` uses the following format for elements in the structures:

- `bt(n).name` — Name of the build tool.
- `bt(n).optstring` — command-line switches for build tool in `bt(n)`.

`cs = getbuildopt(IDE_Obj,file)` returns a character vector of build options for the source file specified by *`file`*. *`file`* must exist in the active project. The resulting `cs` character vector comes from the active build configuration. The type of source file (from the file extension) defines the build tool used by the `cs` character vector.

**Introduced in R2011a**

# getClassName

**Class:** RTW.ModelCPPClass
**Package:** RTW

Get class name from model-specific C++ class interface

## Syntax

`clsName = getClassName(obj)`

## Description

`clsName = getClassName(obj)` gets the name of the class described by the specified model-specific C++ class interface.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a model-specific C++ class interface control object, such as a handle previously returned by `obj = RTW.getClassInterfaceSpecification (modelName)`. |

## Output Arguments

| | |
|---|---|
| `clsName` | A character vector specifying the name of the class described by the specified model-specific C++ class interface. |

## Alternatives

To view the model class name in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, which displays

the model class name and allows you to display and configure the step method for your model class. For more information, see “Configure Step Method for Your Model Class”.

## See Also

### Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

# getCoderExecutionProfile

Extract execution-time profile for code generated from MATLAB function

## Syntax

`myExecutionProfile=getCoderExecutionProfile('myMATLABFunction');`

## Description

`myExecutionProfile=getCoderExecutionProfile('myMATLABFunction');` creates a workspace variable that contains the execution-time profile of the code generated from your MATLAB function.

Run the command after the completion and termination of the SIL/PIL execution of your MATLAB function.

## See Also

`Sections` | `TimerTicksPerSecond` | `report`

### Topics

"Generate Execution Time Profile"
"Analyze Execution Time Data"

**Introduced in R2014b**

# coder.MATLABCodeTemplate.getCurrentTokens

**Class:** coder.MATLABCodeTemplate
**Package:** coder

Get current tokens

## Syntax

`currentTokens = getCurrentTokens()`

## Description

`currentTokens = getCurrentTokens()` returns list of current tokens in the MATLABCodeTemplate object

## Output Arguments

**`currentTokens` — Current tokens**
cell array of character vectors

A list of current tokens in the MATLABCodeTemplate object, returned as a cell array of character vectors.

## Examples

Create a MATLABCodeTemplate object with the default template, then list its tokens.

```
newObj = coder.MATLABCodeTemplate;
% Creates a MATLABCodeTemplate object from the default template
newObj.getCurrentTokens()
% Returns a list of tokens for the template
```

## See Also

`coder.MATLABCodeTemplate.emitSection` | `coder.MATLABCodeTemplate.getTokenValue` | `coder.MATLABCodeTemplate.setTokenValue`

### Topics

"Generate Custom File and Function Banners for C/C++ Code"
"Code Generation Template Files for MATLAB Code"

# getDefaultConf

**Class:** RTW.ModelCPPClass
**Package:** RTW

Get default configuration information for model-specific C++ class interface from Simulink model

## Syntax

`getDefaultConf(obj)`

## Description

`getDefaultConf(obj)` initializes the specified model-specific C++ class interface to a default configuration, based on information from the ERT-based Simulink model to which the interface is attached. On the first invocation, class and step method names and step method properties are set to default values. On subsequent invocations, only step method properties are reset to default values.

Before calling this function, you must call `attachToModel`, to attach the C++ class interface to a loaded model.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a model-specific C++ class interface control object, such as a handle previously returned by `obj = RTW.ModelCPPArgsClass` on page 1-534 or `obj = RTW.ModelCPPDefaultClass` on page 1-540. |

## Alternatives

To view C++ class interface default configuration information in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the

**Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments step method` view of this dialog box, click the **Get Default Configuration** button to display default configuration information. In the `Default step method` view, you can see the default configuration information without clicking a button. For more information, see "Configure Step Method for Your Model Class".

## See Also

### Topics

"Customize C++ Class Interfaces Programmatically"
"Configure Step Method for Model Class"
"Customize Generated C++ Class Interfaces"

# getDefaultConf

**Class:** RTW.ModelSpecificCPrototype
**Package:** RTW

Get default configuration information for model-specific C function prototype from Simulink model

## Syntax

`getDefaultConf(obj)`

## Description

`getDefaultConf(obj)` invokes the specified model-specific C function prototype to initialize the properties and the step function name of the function argument to a default configuration based on information from the ERT-based Simulink model to which it is attached. If you invoke the command again, only the properties of the function argument are reset to default values.

Before calling this function, you must call `attachToModel`, to attach the function prototype to a loaded model.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a model-specific C prototype function control object previously returned by `obj = RTW.ModelSpecificCPrototype`. |

## Alternatives

Click the **Get Default Configuration** button in the Model Interface dialog box to get the default configuration. See “Configure C Step Function Arguments”.

# See Also

## Topics

"Customize Generated C Function Interfaces"

# getFunctionName

**Class:** RTW.ModelSpecificCPrototype
**Package:** RTW

Get function name from model-specific C function prototype

## Syntax

`fcnName = getFunctionName(obj, fcnType)`

## Description

`fcnName = getFunctionName(obj, fcnType)` gets the name of the step or initialize function described by the specified model-specific C function prototype.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a model-specific C prototype function control object previously returned by `obj = RTW.getFunctionSpecification(modelName)`. |
| *fcnType* | Optional character vector specifying which function name to get. Valid options are `'step'` and `'init'`. If *fcnType* is not specified, gets the step function name. |

## Output Arguments

| | |
|---|---|
| *fcnName* | A character vector specifying the name of the function described by the specified model-specific C function prototype. |

## Alternatives

Click the **Get Default Configuration** button in the Model Interface dialog box to get function names. See “Configure C Step Function Arguments”.

## See Also

### Topics

“Customize Generated C Function Interfaces”

# Name

Get name of profiled code section

## Syntax

*SectionName* = *NthSectionProfile*.Name

## Description

*SectionName* = *NthSectionProfile*.Name returns the name that identifies the profiled code section.

The software generates an identifier based on the model entity that corresponds to the profiled section of code.

*NthSectionProfile* is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

## Output Arguments

### *SectionName*

Name that identifies profiled code section

## See Also

ExecutionTimeInTicks | MaximumExecutionTimeCallNum | MaximumExecutionTimeInTicks | MaximumSelfTimeCallNum | MaximumSelfTimeInTicks | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeInTicks | NumCalls | Number | Sections | SelfTimeInTicks | TimerTicksPerSecond | TotalExecutionTimeInTicks | TotalSelfTimeInTicks | TotalTurnaroundTimeInTicks | TurnaroundTimeInTicks | display | report

## Topics

"Code Execution Profiling with SIL and PIL"
"View and Compare Code Execution Times"
"Analyze Code Execution Data"

**Introduced in R2012b**

# Name

Get name of profiled code section

## Syntax

*SectionName* = *NthSectionProfile*.Name

## Description

*SectionName* = *NthSectionProfile*.Name returns the name that identifies the profiled code section.

*NthSectionProfile* is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

## Output Arguments

### *SectionName*

Name that identifies profiled code section

## See Also

ExecutionTimeInTicks | MaximumExecutionTimeCallNum | MaximumExecutionTimeInTicks | MaximumSelfTimeCallNum | MaximumSelfTimeInTicks | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeInTicks | NumCalls | Number | Sections | SelfTimeInTicks | TimerTicksPerSecond | TotalExecutionTimeInTicks | TotalSelfTimeInTicks | TotalTurnaroundTimeInTicks | TurnaroundTimeInTicks | getCoderExecutionProfile | report

### Topics

"Generate Execution Time Profile"

"Analyze Execution Time Data"

**Introduced in R2012b**

# getNamespace

**Class:** RTW.ModelCPPClass
**Package:** RTW

Get namespace from model-specific C++ class interface

## Syntax

*nsName* = getNamespace(*obj*)

## Description

*nsName* = getNamespace(*obj*) gets the namespace of the class described by the specified model-specific C++ class interface.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a model-specific C++ class interface control object, such as a handle previously returned by *obj* = RTW.getClassInterfaceSpecification (*modelName*). |

## Output Arguments

| | |
|---|---|
| nsName | A character vector specifying the namespace of the class described by the specified model-specific C++ class interface. |

## Alternatives

To view the model namespace in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, which displays

the model class name and namespace and allows you to display and configure the step method for your model class. For more information, see “Configure Step Method for Your Model Class”.

## See Also

### Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

# getNumArgs

**Class:** RTW.ModelCPPClass
**Package:** RTW

Get number of step method arguments from model-specific C++ class interface

## Syntax

`num = getNumArgs(obj)`

## Description

`num = getNumArgs(obj)` gets the number of arguments for the step method described by the specified model-specific C++ class interface.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a model-specific C++ class interface control object, such as a handle previously returned by `obj = RTW.getClassInterfaceSpecification (modelName)`. |

## Output Arguments

| | |
|---|---|
| `num` | An integer specifying the number of step method arguments. |

## Alternatives

To view the number of step method arguments in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. In the `I/O`

`arguments step method` view of this dialog box, click the **Get Default Configuration** button to display the step method arguments. For more information, see “Configure Step Method for Your Model Class”.

## See Also

### Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

# getNumArgs

**Class:** RTW.ModelSpecificCPrototype
**Package:** RTW

Get number of function arguments from model-specific C function prototype

## Syntax

`num = getNumArgs(obj)`

## Description

`num = getNumArgs(obj)` gets the number of function arguments for the function described by the specified model-specific C function prototype.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a model-specific C prototype function control object previously returned by `obj = RTW.getFunctionSpecification(modelName)`. |

## Output Arguments

| | |
|---|---|
| *num* | An integer specifying the number of function arguments. |

## Alternatives

Click the **Get Default Configuration** button in the Model Interface dialog box to get arguments. See “Configure C Step Function Arguments”.

## See Also

### Topics

"Customize Generated C Function Interfaces"

# NumCalls

Total number of calls to profiled code section

## Syntax

*`TotalNumCalls`* `=` *`NthSectionProfile`*`.NumCalls`

## Description

*`TotalNumCalls`* `=` *`NthSectionProfile`*`.NumCalls` returns the total number of calls to the profiled code section over the entire simulation.

*`NthSectionProfile`* is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

## Output Arguments

***`TotalNumCalls`***

Total number of calls

## See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `display` | `report`

### Topics

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

"Analyze Code Execution Data"

**Introduced in R2012b**

# NumCalls

Total number of calls to profiled code section

## Syntax

`TotalNumCalls = NthSectionProfile.NumCalls`

## Description

*TotalNumCalls* = *NthSectionProfile*.NumCalls returns the total number of calls to the profiled code section over the entire execution.

*NthSectionProfile* is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

## Output Arguments

***TotalNumCalls***

Total number of calls

## See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `getCoderExecutionProfile` | `report`

### Topics

“Generate Execution Time Profile”

"Analyze Execution Time Data"

**Introduced in R2012b**

# getOutputData

**Class:** cgv.CGV
**Package:** cgv

Get output data

## Syntax

`out = cgvObj.getOutputData(InputIndex)`

## Description

*`out`* = *`cgvObj`*`.getOutputData(`*`InputIndex`*`)` is the method that you use to retrieve the output data that the object creates during execution of the model. *`out`* is the output data that the object returns. *`cgvObj`* is a handle to a `cgv.CGV` object. *`InputIndex`* is a unique numeric identifier that specifies which output data to retrieve. The *`InputIndex`* is associated with specific input data.

## See Also

### Topics

"Verify Numerical Equivalence with CGV"

# getPreview

**Class:** RTW.ModelSpecificCPrototype
**Package:** RTW

Get model-specific C function prototype code preview

## Syntax

`preview = getPreview(obj, fcnType)`

## Description

`preview = getPreview(obj, fcnType)` gets the model-specific C function prototype code preview.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a model-specific C prototype function control object previously returned by `obj = RTW.getFunctionSpecification(modelName)`. |
| *fcnType* | Optional. Character vector specifying which function to preview. Valid options are `'step'` and `'init'`. If *fcnType* is not specified, previews the step function. |

## Output Arguments

| | |
|---|---|
| *preview* | Character vector specifying the function prototype for the step or initialization function. |

## Alternatives

Use the **Step function preview** subpane in the Model Interface dialog box to preview how your step function prototype is interpreted in generated code. See “Configure C Step Function Arguments”.

## See Also

### Topics

“Customize Generated C Function Interfaces”

# getReportData

**Class:** cgv.Config
**Package:** cgv

Return results of comparing configuration parameter values

## Syntax

`*rpt_data* = *cfgObj*.getReportData()`

## Description

*`rpt_data`* `=` *`cfgObj`*`.getReportData()` compares the original configuration parameter values with the values that the object recommends. *`cfgObj`* is a handle to a `cgv.Config` object. Returns a cell array of character vectors with the model, parameter, previous value, and recommended or new value.

## See Also

### Topics

"Verify Numerical Equivalence with CGV"

# getSavedSignals

**Class:** cgv.CGV
**Package:** cgv

Display list of signal names to command line

## Syntax

`signal_list = cgvObj.getSavedSignals(simulation_data)`

## Description

`signal_list = cgvObj.getSavedSignals(simulation_data)` returns a cell array, *signal_list*, of the output signal names of the data elements from the input data set, *simulation_data*. *simulation_data* is the output data stored in the CGV object, *cgvObj*, when you execute the model.

## Tips

- After executing your model, use the `getOutputData` function to get the output data used as the input argument to the `cgvObj.getSavedSignals` function.
- Use names from the output signal list at the command line or as input arguments to other CGV functions, for example, `createToleranceFile`, `compare`, and `plot`.

## See Also

### Topics

"Verify Numerical Equivalence with CGV"

# Number

Get number that uniquely identifies profiled code section

## Syntax

`SectionNumber = NthSectionProfile.Number`

## Description

*SectionNumber* = *NthSectionProfile*.Number returns a number that uniquely identifies the profiled code section, for example, in the code execution profiling report.

*NthSectionProfile* is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

## Output Arguments

***SectionNumber***

Number of profiled code section

## See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `display` | `report`

### Topics

"Code Execution Profiling with SIL and PIL"

"View and Compare Code Execution Times"
"Analyze Code Execution Data"

**Introduced in R2012b**

# Number

Get number that uniquely identifies profiled code section

## Syntax

`SectionNumber = NthSectionProfile.Number`

## Description

*`SectionNumber`* = *`NthSectionProfile`*`.Number` returns a number that uniquely identifies the profiled code section.

*`NthSectionProfile`* is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

## Output Arguments

***`SectionNumber`***

Number of profiled code section

## See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `getCoderExecutionProfile` | `report`

### Topics

"Generate Execution Time Profile"

"Analyze Execution Time Data"

**Introduced in R2012b**

# pil_block_replace

Replace block in model with block from another model

## Syntax

```
pil_block_replace(sourceModelBlock, destinationModelBlock)
pil_block_replace(sourceModelBlock, destinationModelBlock,
'isvisible')
```

## Description

`pil_block_replace(sourceModelBlock, destinationModelBlock)` replaces a block in the destination model with a block from the source model. To preserve the original block from the destination model, in the source model, the function replaces `sourceModelBlock` with `destinationModelBlock`.

`pil_block_replace(sourceModelBlock, destinationModelBlock, 'isvisible')` highlights the replaced block in the destination model.

## Examples

### Replace Destination Block with Source Block

This example shows how to replace a block in a model with a block from another model.

Create a destination model that contains an Outport block, `destinationBlock`.

```
new_system('destModel')
open_system('destModel');
add_block('simulink/Sinks/Out1', 'destModel/destinationBlock')
```

Create a source model that contains a Scope block, `sourceBlock`.

```
new_system('srcModel')
open_system('srcModel');
add_block('simulink/Sinks/Scope', 'srcModel/sourceBlock')
```

Replace the Outport block in the destination model with the Scope block from the source model.

```
pil_block_replace('srcModel/sourceBlock','destModel/destinationBlock','isvisible')
```

- “Cross-Release Code Integration”

## Input Arguments

**sourceModelBlock — Source block**
character vector

Full path to the replacement block in the source model.

Example: 'srcModel/sourceBlock'

**destinationModelBlock — Destination block**
character vector

Full path to the block in the destination model, which the source block replaces.

Example: 'destModel/destinationBlock'

## See Also

### Topics
“Cross-Release Code Integration”

**Introduced in R2006b**

# piltest

Verify custom target connectivity configuration for Simulink PIL simulation

## Syntax

```
piltest(config)
piltest(config,'ConfigParams',additionalParameterList)
piltest(config,'TestPoint',testName)
```

## Description

`piltest(config)` runs a suite of tests that verify your custom processor-in-the-loop (PIL) target connectivity configuration. In the tests, the function runs various normal, software-in-the-loop (SIL), and PIL simulations. The function compares results and produces errors if it detects differences between simulation modes. For the PIL simulations, the function extracts these parameters from `config`:

- `SystemTargetFile`
- `TargetHWDeviceType`
- `Toolchain`

In the current working folder, the function creates the `piltest` folder, which contains subfolders with test results.

`piltest(config,'ConfigParams',additionalParameterList)` extracts additional parameters from `config` for the PIL simulation.

`piltest(config,'TestPoint',testName)` runs a specific test from the test suite.

## Examples

### Verify Target Connectivity Configuration with `piltest`

This example uses `piltest` to verify a target connectivity configuration for PIL simulations on your development computer.

Create a target connectivity implementation in your current working folder.

```
% Make a local copy of the connectivity classes.
src_dir = ...
    fullfile(matlabroot,'toolbox','coder','simulinkcoder',...
             '+coder','+mypil');
if exist(fullfile('.','+mypil'),'dir')
    rmdir('+mypil','s')
end
mkdir +mypil
copyfile(fullfile(src_dir,'Launcher.m'), '+mypil');
copyfile(fullfile(src_dir,'TargetApplicationFramework.m'), '+mypil');
copyfile(fullfile(src_dir,'ConnectivityConfig.m'), '+mypil');

% Make the copied files writable.
fileattrib(fullfile('+mypil', '*'),'+w');

% Update the package name to reflect the new location of the files.
coder.mypil.Utils.UpdateClassName(...
    './+mypil/ConnectivityConfig.m',...
    'coder.mypil',...
    'mypil');
```

Register a target connectivity configuration using an `sl_customization.m` file. This example uses a supplied file.

```
sl_customization_path = fullfile(matlabroot,...
    'toolbox',...
    'rtw',...
    'rtwdemos',...
    'pil_demo');
addpath(sl_customization_path);
sl_refresh_customizations;
```

Specify the PIL simulation mode for the model.

```
close_system('rtwdemo_sil_topmodel')
open_system('rtwdemo_sil_topmodel')
set_param('rtwdemo_sil_topmodel','SimulationMode',...
          'processor-in-the-loop (pil)');
```

Specify the manufacturer and test hardware type. For example, PIL simulation on a 64-bit Windows® development computer requires:

```
set_param('rtwdemo_sil_topmodel','TargetHWDeviceType',...
          'Intel->x86-64 (Windows64)');
set_param('rtwdemo_sil_topmodel','TargetLongLongMode',true);
```

Run `piltest`.

```
piltest('rtwdemo_sil_topmodel', 'ConfigParam', {'ProdLongLongMode'} )
```

- “Create PIL Target Connectivity Configuration for Simulink”
- “SIL and PIL Simulations”

## Input Arguments

**`config` — Configuration set, configuration reference, or model**
`Simulink.ConfigSet`|`Simulink.ConfigSetRef`|character vector

A configuration set, configuration set reference, or Simulink model.

**`additionalParameterList` — Additional parameters**
cell array of character vectors

Extract additional parameters from `config` for PIL simulation.

**`testName` — Specific test**
`'all'` (default) | `'verifyPILBlock'` | `'verifyModelBlock'` | `'verifyTopModel'` | `'verifyExecutionOnTarget'` | `'verifyTopModelSILPILSwitching'` | `'verifyModelBlockSILPILSwitching'`

- `'verifyPILBlock'` — For normal mode results, run a simulation of a Simulink model with a subsystem. For PIL results, replace the subsystem with a PIL block and rerun the simulation. The function compares normal and PIL mode results. If the function detects differences, it produces an error.
- `'verifyModelBlock'` — For normal mode results, run simulations of a Simulink model with a Model block in normal mode.

  For PIL mode results, run simulation loops with the Model block in PIL mode. The function varies these settings:

  - Model block parameter **Code interface** — Set to `Top model` (standalone code interface) or `Model reference`.
  - **Configuration Parameters** > **Code Generation** > **Language** — Set to `C` or `C++`. For the `C++` case, the function sets **Code Generation** > **Interface** > **Code interface packaging** to `C++ class`.

  The function compares normal and PIL mode results. If the function detects differences, it produces an error.
- `'verifyTopModel'` — Run simulations of a Simulink top-model in normal and PIL modes. The function compares normal and PIL mode results. If the function detects differences, it produces an error.
- `'verifyExecutionOnTarget'` — Run simulations of a Simulink model with a Model block in normal and PIL modes. For each mode, the Model block uses standalone and model reference code interfaces. For PIL mode, the function introduces a deliberate mismatch. The function compares normal and PIL mode results. If it does not detect the deliberate mismatch, it produces an error.
- `'verifyTopModelSILPILSwitching'` — For a Simulink top model:
  - Verify that production code is not regenerated when the function switches between SIL and PIL simulation modes. The function compares timestamps of the production code in each mode.
  - Compares results from SIL and PIL mode simulations to results from a normal mode simulation.

  If the function detects differences in timestamps or simulation results, it produces an error.
- `'verifyModelBlockSILPILSwitching'` — For a Simulink Model block:
  - Verify that production code is not regenerated when the Model block simulation mode switches between SIL and PIL modes. The function compares timestamps of the production code in each mode.
  - Run simulation loops with the Model block in SIL and PIL modes. The function varies the **Code interface** Model block parameter, setting this parameter to `Top model` or `Model reference`. The function compares results from SIL and PIL mode simulations to results from a normal mode simulation.

  If the function detects differences in timestamps or simulation results, it produces an error.

- `'all'` — Run all tests from the test suite.

## See Also

`Simulink.ConfigSet` | `Simulink.ConfigSetRef`

### Topics

"Create PIL Target Connectivity Configuration for Simulink"
"SIL and PIL Simulations"

**Introduced in R2016b**

# piltest

Verify custom target connectivity configuration for MATLAB PIL execution

## Syntax

```
piltest(config)
piltest(config,'ConfigParams',additionalParameterList)
piltest(config,'TestPoint',testName)
```

## Description

`piltest(config)` runs tests that verify your custom processor-in-the-loop (PIL) target connectivity configuration. In the tests, the function runs the MATLAB function and performs PIL executions. The function compares results and produces errors if it detects differences. For PIL executions, the function extracts the `TargetHWDeviceType` and `Toolchain` settings from `config`.

In the current working folder, the function creates the `piltest` folder, which contains subfolders with test results.

`piltest(config,'ConfigParams',additionalParameterList)` extracts additional settings from `config` for the PIL execution.

`piltest(config,'TestPoint',testName)` runs the specified test.

## Examples

### Verify Target Connectivity Configuration with `piltest`

This example shows how you can use `piltest` to verify a target connectivity configuration for PIL execution.

Create a code generation configuration object for C/C++ static library generation.

```
cfg = coder.config('lib');
```

Create hardware configuration object, specify manufacturer and test hardware type, and assign handle to code generation object.

```
hwImpl = coder.HardwareImplementation;
hwImpl.TargetHWDeviceType = 'Atmel->AVR';
cfg.HardwareImplementation = hwImpl;
```

Specify the toolchain for code generation.

```
cfg.Toolchain = 'AVR tools for Arduino';
```

Run the function.

```
piltest(cfg)
```

- “Create PIL Target Connectivity Configuration for MATLAB”
- “PIL Execution of Code Generated for a Kalman Estimator”

## Input Arguments

### `config` — Configuration object

`coder.EmbeddedCodeConfig`

A configuration object that specifies code generation parameters.

### `additionalParameterList` — Additional parameters

cell array of character vectors

Extract additional parameters from `config` for PIL execution.

### `testName` — Specific test

`'all'` (default) | `'verifyPILConfig'`

- `'verifyPILConfig'` — For a given set of input values, the function:
  - Runs a MATLAB function on your development computer.
  - Performs PIL executions of generated MATLAB code on your target hardware with `config.TargetLang` set to `'C'` and `'C++'`.

The function compares MATLAB function and PIL results. If the function detects differences, it produces an error.

- `'all'` — Run all tests.

## See Also

### Topics

"Create PIL Target Connectivity Configuration for MATLAB"
"PIL Execution of Code Generated for a Kalman Estimator"

**Introduced in R2016b**

# Sections

Get array of `coder.profile.ExecutionTimeSection` objects for profiled code sections

## Syntax

```
NthSectionProfile = myExecutionProfile.Sections(N)
numberOfSections = length(myExecutionProfile.Sections)
```

## Description

`NthSectionProfile = myExecutionProfile.Sections(N)` returns an `coder.profile.ExecutionTimeSection` object for the *N*th profiled code section.

`numberOfSections = length(myExecutionProfile.Sections)` returns the number of code sections for which profile data is available.

*`myExecutionProfile`* is a workspace variable generated by a simulation.

## Input Arguments

### *N*

Index of code section for which profile data is required

## Output Arguments

### *NthSectionProfile*

Object that contains profile information about the code section. You can use the following `coder.profile.ExecutionTimeSection` methods to retrieve the information:

- `Name` — Name of the code section.

- `Number` — Number of the code section.
- `NumCalls` — Number of calls to the code section.
- `TotalExecutionTimeInTicks` — Total number of timer ticks recorded for the code section over the entire simulation.
- `TurnaroundTimeInTicks` — Time between start and finish of the code section, in timer ticks.
- `TotalTurnaroundTimeInTicks` — Total number of timer ticks between start and finish of the code section, over the entire simulation.
- `MaximumExecutionTimeInTicks` — Maximum number of timer ticks for a single invocation of the code section.
- `MaximumExecutionTimeCallNum` — Number of call associated with the maximum number of timer ticks recorded for a single invocation of the code section.
- `MaximumTurnaroundTimeInTicks` — Maximum number of ticks between start and finish for a single invocation.
- `MaximumTurnaroundTimeCallNum` — Number of call associated with the maximum time between start and finish of a single invocation.
- `MaximumSelfTimeInTicks` — Maximum self time, in timer ticks.
- `SelfTimeInTicks` — Self time for the code section, in timer ticks.
- `TotalSelfTimeInTicks` — Total self time for the code section, over the entire simulation.
- `MaximumSelfTimeCallNum` — Call associated with maximum self time.
- `ExecutionTimeInTicks` — Vector of execution times.

***numberOfSections***

Number of code sections with profile data

# See Also

`TimerTicksPerSecond` | `display` | `report`

## Topics

"Code Execution Profiling with SIL and PIL"
"View and Compare Code Execution Times"
"Analyze Code Execution Data"

**Introduced in R2012b**

# Sections

Get array of `coder.profile.ExecutionTimeSection` objects for profiled code sections

## Syntax

```
NthSectionProfile = myExecutionProfile.Sections(N)
numberOfSections = length(myExecutionProfile.Sections)
```

## Description

*`NthSectionProfile`* = *`myExecutionProfile`*`.Sections(`*`N`*`)` returns an `coder.profile.ExecutionTimeSection` object for the *N*th profiled code section.

*`numberOfSections`* = `length(`*`myExecutionProfile`*`.Sections)` returns the number of code sections for which profile data is available.

*`myExecutionProfile`* is a workspace variable that you create using `getCoderExecutionProfile`.

## Input Arguments

### *N*

Index of code section for which profile data is required

## Output Arguments

### *NthSectionProfile*

Object that contains profile information about the code section. You can use the following `coder.profile.ExecutionTimeSection` methods to retrieve the information:

- `Name` — Name of the code section.
- `Number` — Number of the code section.
- `NumCalls` — Number of calls to the code section.
- `TotalExecutionTimeInTicks` — Total number of timer ticks recorded for the code section over the entire execution.
- `TurnaroundTimeInTicks` — Time between start and finish of the code section, in timer ticks.
- `TotalTurnaroundTimeInTicks` — Total number of timer ticks between start and finish of the code section, over the entire execution.
- `MaximumExecutionTimeInTicks` — Maximum number of timer ticks for a single invocation of the code section.
- `MaximumExecutionTimeCallNum` — Number of call associated with the maximum number of timer ticks recorded for a single invocation of the code section.
- `MaximumTurnaroundTimeInTicks` — Maximum number of ticks between start and finish for a single invocation.
- `MaximumTurnaroundTimeCallNum` — Number of call associated with the maximum time between start and finish of a single invocation.
- `MaximumSelfTimeInTicks` — Maximum self time, in timer ticks.
- `SelfTimeInTicks` — Self time for the code section, in timer ticks.
- `TotalSelfTimeInTicks` — Total self time for the code section, over the entire execution.
- `MaximumSelfTimeCallNum` — Call associated with maximum self time.
- `ExecutionTimeInTicks` — Vector of execution times.

***numberOfSections***

Number of code sections with profile data

## See Also

`TimerTicksPerSecond` | `getCoderExecutionProfile` | `report`

### Topics

“Generate Execution Time Profile”
“Analyze Execution Time Data”

**Introduced in R2012b**

# sharedCodeMATLABVersions

Manage MATLAB versions for cross-release code integration

## Syntax

```
[registeredVersions, installationFolders] = sharedCodeMATLABVersions
sharedCodeMATLABVersions('Folder',versionInstallationFolder)
sharedCodeMATLABVersions('Remove', deregisterVersion)
```

## Description

`[registeredVersions, installationFolders] = sharedCodeMATLABVersions` returns the available MATLAB versions and the installation folders.

`sharedCodeMATLABVersions('Folder',versionInstallationFolder)` registers a MATLAB version. The function specifies the folder where the MATLAB version is installed. The function checks that the folder corresponds to the `matlabroot` value for a valid installation, retrieves the MATLAB version number, and stores this information as a preference.

`sharedCodeMATLABVersions('Remove', deregisterVersion)` deregisters the MATLAB version and removes installation folder and version data.

## Examples

### Register Previous MATLAB Version for Cross-Release Code Integration

This code shows how you can register a previous release for your cross-release code integration workflow.

```
[registeredMATLABs, installationFolders] = sharedCodeMATLABVersions;
requiredVersion = 'R2017a';
typicalPath = 'C:\Program Files\MATLAB';
```

```
if isempty(registeredMATLABs) || ~any(strcmp(requiredVersion, registeredMATLABs))
  versionFolder = fullfile(typicalPath, requiredVersion);
  sharedCodeMATLABVersions('Folder', versionFolder);
end
```

- “Cross-Release Shared Utility Code Reuse”
- “Cross-Release Code Integration”

## Input Arguments

**`versionInstallationFolder` — Installation folder location**
character vector

Full path to the installation folder for the MATLAB version that you want to register.

Example: `'C:\Program Files\MATLAB\R2017a'`

**`deregisterVersion` — Release version to deregister**
character vector

MATLAB version that you want to deregister.

Example: `'R2017a'`

## Output Arguments

**`registeredVersions` — Registered release versions**
cell array of character vectors

MATLAB release versions that are registered by the function.

**`installationFolders` — Installation folder paths**
cell array of character vectors

Installation folder locations for registered MATLAB versions.

## See Also

`crossReleaseImport` | `crossReleaseExport` | `sharedCodeUpdate`

### Topics

"Cross-Release Shared Utility Code Reuse"
"Cross-Release Code Integration"

**Introduced in R2017b**

# sharedCodeUpdate

Add new shared code source files to existing shared code folder

## Syntax

```
sharedCodeUpdate(sourceFolder,destinationFolder)
sharedCodeUpdate(sourceFolder,destinationFolder,
'ExistingCodeSubfolder', destinationSubfolder)
sharedCodeUpdate(artifactsLocation,destinationFolder)
sharedCodeUpdate(artifactsLocation,destinationFolder,'CodeLocation'
pathToRelocatedModelCode)
sharedCodeUpdate(artifactsLocation,configurationSetOrModel)
```

## Description

`sharedCodeUpdate(sourceFolder,destinationFolder)` copies, for example, shared utility files from `sourceFolder` to a subfolder in `destinationFolder` provided that the files do not exist within `destinationFolder`. The function:

- Identifies files in both folders that have identical names but different content. The function does not overwrite these files in `destinationFolder`. In the Command Window, you see a `compare` link for each file. To examine differences by using the Comparison tool, click the link.
- Lists `sourceFolder` files that the function intends to copy and seeks confirmation. When you provide confirmation, the function copies the files to `destinationFolder`. By default, the destination of the copied files is a subfolder that corresponds to the release in which the files were created, for example, `R2015a` or `R2015b`.

`sharedCodeUpdate(sourceFolder,destinationFolder, 'ExistingCodeSubfolder', destinationSubfolder)` copies files to the subfolder that you specify.

`sharedCodeUpdate(artifactsLocation,destinationFolder)` copies shared code source files from the shared code location specified by a cross-release artifact in `artifactsLocation`.

`sharedCodeUpdate(artifactsLocation,destinationFolder,'CodeLocation' pathToRelocatedModelCode)` copies shared code source files from a build folder hierarchy within the anchor folder specified by `pathToRelocatedModelCode`

`sharedCodeUpdate(artifactsLocation,configurationSetOrModel)` copies shared code source files to the folder specified by the `'ExistingSharedCode'` parameter of a Simulink configuration set or model.

## Examples

### Copy Shared Utility Files to Shared Code Folder

This example shows how to copy source files from a shared utilities folder to a shared code folder.

```
sourceFolder = fullfile(pwd,'R2015bWork','slprj','ert','_sharedutils');
existingSharedCodeFolder = fullfile(pwd,'SharedUtilCode');
sharedCodeUpdate(sourceFolder, existingSharedCodeFolder);
```

### Copy Shared Utility Files to Subfolder

This example shows how to copy source files from a shared utilities folder to a specified subfolder in the shared code folder.

```
sourceFolder = fullfile(pwd,'R2015bWork','slprj','ert','_sharedutils');
existingSharedCodeFolder = fullfile(pwd, 'SharedUtilCode');
destinationSubfolder = 'mySub'
sharedCodeUpdate(sourceFolder, existingSharedCodeFolder,...
'ExistingCodeSubfolder', destinationSubfolder);
```

### Copy Shared Utility Files From Relocated Code Folder

This example shows how to copy shared utility files from a relocated generated code folder to an existing shared code folder. Previously created artifacts are located in `artLocFolder`.

Specify path to shared code folder that you want to update.

```
pathToExistingSharedFolder = 'C:\mySharedCodeFolder';
```

Specify the full path to the relocated generated code folder `P1_ert_rtw`.

```
anchorFolder = 'C:\myWorkFolder';
relocatedCodeFolder = fullfile(anchorFolder, 'P1_ert_rtw');
```

Update the existing shared code folder.

```
sharedCodeUpdate(artLocFolder, pathToExistingSharedFolder,...
    'CodeLocation', fileparts(relocatedCodeFolder));
```

- “Cross-Release Shared Utility Code Reuse”
- “Cross-Release Code Integration”

## Input Arguments

**`sourceFolder` — Source folder**
character vector

File path to folder with shared code files that you want to add to existing shared code folder.

**`destinationFolder` — Existing shared code folder**
character vector

File path to existing shared code folder.

**`destinationSubfolder` — Destination subfolder**
character vector

Destination subfolder in existing shared code folder.

**`artifactsLocation` — Artifacts folder**
character vector

Path to an export artifacts folder previously created by `crossReleaseExport`.

**`pathToRelocatedModelCode` — Anchor folder for relocated code**
character vector

Path to the anchor folder of a build folder hierarchy. The build folders in the hierarchy contain generated code, which you relocated after the creation of cross-release artifacts.

**`configurationSetOrModel` — Configuration set or model**
character vector

Simulink configuration set or model that uses an existing shared code folder specified by the `'ExistingSharedCode'` parameter.

# See Also

`crossReleaseImport` | `crossReleaseExport`

## Topics

"Cross-Release Shared Utility Code Reuse"
"Cross-Release Code Integration"

**Introduced in R2016b**

# getStatus

**Class:** cgv.CGV
**Package:** cgv

Return execution status

## Syntax

```
status = cgvObj.getStatus()
status = cgvObj.getStatus(inputName)
```

## Description

`status = cgvObj.getStatus()` returns the execution status of *`cgvObj`*. *`cgvObj`* is a handle to a `cgv.CGV` object.

`status = cgvObj.getStatus(inputName)` returns the status of a single execution for `inputName`.

## Input Arguments

**`inputName`**

`inputName` is a unique numeric or character identifier associated with input data, which is added to the `cgv.CGV` object using `addInputData`.

## Output Arguments

**`status`**

If `inputName` is provided, `status` is the result of the execution of input data associated with `inputName`.

| Value | Description |
|---|---|
| none | Execution has not run. |
| pending | Execution is currently running. |
| completed | Execution ran to completion without errors and output data is available. |
| passed | Baseline data was provided. Execution ran to completion and comparison to the baseline data returned no differences. |
| error | Execution produced an error. |
| failed | Baseline data was provided. Execution ran to completion and comparison to the baseline data returned a difference. |

If `inputName` is not provided, the following pseudocode describes the return status:

```
if (all executions return 'passed')
    status = 'passed'
else if (all executions return 'passed' or 'completed')
  status = 'completed'
else if (an execution returns 'error')
    status = 'error'
else if (an execution returns 'failed')
    status = 'failed'
else if (an execution returns 'none' or 'pending'
    status = 'none'
```

## See Also

`cgv.CGV.addBaseline` | `cgv.CGV.addInputData` | `cgv.CGV.run`

### Topics

"Verify Numerical Equivalence with CGV"

# getStepMethodName

**Class:** RTW.ModelCPPClass
**Package:** RTW

Get step method name from model-specific C++ class interface

## Syntax

`fcnName = getStepMethodName(obj)`

## Description

`fcnName = getStepMethodName(obj)` gets the name of the step method described by the specified model-specific C++ class interface.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a model-specific C++ class interface control object, such as a handle previously returned by `obj = RTW.getClassInterfaceSpecification (modelName)`. |

## Output Arguments

| | |
|---|---|
| *fcnName* | A character vector specifying the name of the step method described by the specified model-specific C++ class interface. |

## Alternatives

To view the step method name in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, which displays

the step method name and allows you to display and configure the step method for your model class. For more information, see “Configure Step Method for Your Model Class”.

# See Also

## Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

# getTflArgFromString

Create code replacement argument based on specified name and built-in data type

## Syntax

```
arg = getTflArgFromString(hTable,name,datatype)
```

## Description

`arg = getTflArgFromString(hTable,name,datatype)` creates a code replacement argument that is based on a specified name and built-in or fixed-point data type.

The `IOType` property of the created argument defaults to `'RTW_IO_INPUT'`, indicating an input argument. For an output argument, change the `IOType` value to `'RTW_IO_OUTPUT'` by directly assigning the argument property.

This function does not support matrices. To create a matrix argument, use the argument class `RTW.TflArgMatrix` as shown in “Small Matrix Operation to Processor Code Replacement”, “Matrix Multiplication Operation to MathWorks BLAS Code Replacement”, and “Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement”.

## Examples

### Create and Add an Output Argument

This example shows how to use `getTflArgFromString` to create an `int16` output argument named `y1`. Then, the example adds the argument as a conceptual argument for a code replacement table entry.

```
hLib = RTW.TflTable;
op_entry = RTW.TflCOperationEntry;
.
.
.
```

```
arg = hLib.getTflArgFromString('y1', 'int16');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.addConceptualArg(arg);
```

## Input Arguments

**hTable — Handle to a code replacement table**
handle

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

Example: hLib

**name — Specifies the name to use for a code replacement argument**
character vector

Example: 'y1'

**datatype — Specifies a built-in data type or a fixed-point data type to use for the code replacement argument**
'integer' | 'int8' | 'int16' | 'int32' | 'long' | 'long_long' | 'uinteger' | 'uint8' | 'uint16' | 'uint32' | 'ulong' | 'ulong_long' | 'single' | 'double' | 'boolean' | 'logical'

You can specify fixed-point data types using the fixdt function from Fixed-Point Designer™ software; for example, 'fixdt(1,16,2)'.

Example: 'integer'

## Output Arguments

**arg — Handle to the created code replacement argument**
handle

The *arg* is a handle to the created code replacement argument, which can be specified to the addConceptualArg function.

## See Also

`addConceptualArg`

### Topics

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

**Introduced in R2008a**

# getTflDWorkFromString

Create code replacement DWork argument for semaphore entry based on specified name and data type

## Syntax

`arg = getTflDWorkFromString(hTable,name,datatype)`

## Description

`arg = getTflDWorkFromString(hTable,name,datatype)` creates a code replacement DWork argument, based on a specified name and data type, for a semaphore entry in a code replacement table.

## Examples

### Create and Add a DWork Argument

This example shows how to use the `getTflDworkFromString` to create a `void*` argument named `d1`. Then, the example adds the argument as a DWork argument for a semaphore entry in a code replacement table.

```
hLib = RTW.TflTable;

% specify semaphore init function.
hEnt = RTW.TflCSemaphoreEntry;
hEnt.setTflCFunctionEntryParameters( ...
  'Key',                      'sem_init', ...
  'Priority',                 30, ...
  'ImplementationName',       'mySemCreate', ...
  'ImplementationHeaderFile', 'mySem.h', ...
  'ImplementationSourceFile', 'mySem.c', ...
  'ImplementationHeaderPath', LibPath, ...
  'ImplementationSourcePath', LibPath, ...
```

```
  'GenCallback',              'RTW.copyFileToBuildDir', ...
  'SideEffects',              true);

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

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

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

hLib.addEntry( hEnt );
```

## Input Arguments

**hTable — Handle to a code replacement table**
handle

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

Example: `hLib`

**name — Specifies the name to use for the code replacement DWork argument**
character vector

Example: `'d1'`

**datatype — Specifies a data type to use for the code replacement DWork argument**
character vector

You must specify `'void*'`.

Example: `'void*'`

## Output Arguments

**`arg` — Handle to the created code replacement argument**
arg

The *`arg`* is a handle to the created code replacement argument, which can be specified to the `addDWorkArg` function.

## See Also

`addDWorkArg`

### Topics

“Semaphore and Mutex Function Replacement”
“Define Code Replacement Mappings”

**Introduced in R2013a**

# coder.hardware

Create hardware configuration object for PIL execution

## Syntax

```
hw = coder.hardware(name)
coder.hardware()
```

## Description

`hw = coder.hardware(name)` returns the configuration object of the hardware `name` that you select. You can use this object as the hardware configuration object for a Processor-in-the-Loop (PIL) execution with MATLAB Coder.

`coder.hardware()` returns a cell array of names of hardware that the installed hardware support packages support. If hardware support is not installed, it returns an empty cell array.

The support packages that support PIL execution by using the `coder.hardware` class are:

- Embedded Coder Support Package for BeagleBone® Black Hardware
- Embedded Coder Support Package for ARM® Cortex®-A Processors
- Embedded Coder Support Package for Intel® SoC Devices
- Embedded Coder Support Package for Xilinx® Zynq® Platform

You must install one of these support packages. For an example of the full PIL execution with each target hardware, see the "Processor-in-the-Loop Verification of MATLAB Functions" example in the support packages' documentation.

## Examples

### Create a Hardware Configuration Object for BeagleBone Hardware

```
hw = coder.hardware('BeagleBone Black');
disp(hw)

  Hardware with properties:

             Name: 'BeagleBone Black'
     CPUClockRate: 1000
         Password: 'root'
         Username: 'admin'
    DeviceAddress: '192.168.1.10'
```

### View Supported Hardware List

```
coder.hardware()


ans =

    'ARM Cortex-A9 (QEMU)'    'BeagleBone Black'
```

### Set Hardware Configuration to Coder Configuration Object

Create hardware configuration object and set properties.

```
hw = coder.hardware('BeagleBone Black');
hw.DeviceAddress = '192.168.1.100'
hw.UserName = 'admin';
hw.Password = 'password';
disp(hw)

  Hardware with properties:

             Name: 'BeagleBone Black'
     CPUClockRate: 1000
    DeviceAddress: '192.168.1.100'
         Username: 'admin'
         Password: 'password'
```

Set hardware configuration object to coder configuration object.

```
cfg = coder.config('lib','ecoder',true);
cfg.VerificationMode = 'PIL';
cfg.Hardware = hw;
```

- “PIL Execution with ARM Cortex-A at the Command Line”
- “PIL Execution with ARM Cortex-A by Using the MATLAB Coder App”

## Input Arguments

**name — Name of target hardware**
character vector | `'BeagleBone Black'` | `'ARM Cortex-A9 (QEMU)'`

Name of target hardware, specified as a character vector. You can get the names of the installed hardware support packages by calling `coder.hardware()` with no input arguments.

Example: `'BeagleBone Black'`

Data Types: `char`

## Output Arguments

**hw — Hardware configuration object**
object

Hardware configuration object used for the code generation configuration class.

## See Also

**Functions**
`codegen` | `coder.config`

**Classes**
`coder.hardware`

### Topics
“PIL Execution with ARM Cortex-A at the Command Line”

"PIL Execution with ARM Cortex-A by Using the MATLAB Coder App"

**Introduced in R2015b**

# coder.Hardware class

**Package:** coder

codegen configuration object that specifies hardware parameters for PIL execution

## Description

`coder.Hardware` is a class that defines the properties of the target hardware for a Processor-in-the-Loop (PIL) execution with MATLAB Coder. Each target hardware has a `CPUClockRate`, `Name`, and other dynamic properties specific to their hardware.

The support packages that support Processor-in-the-Loop (PIL) execution by using the `coder.Hardware` class are:

- Embedded Coder Support Package for BeagleBone Black Hardware
- Embedded Coder Support Package for ARM Cortex-A Processors
- Embedded Coder Support Package for Intel SoC Devices
- Embedded Coder Support Package for Xilinx Zynq Platform

You must install one of these support packages. For an example of the full PIL execution with each target hardware, see the "Processor-in-the-Loop Verification of MATLAB Functions" example in the support packages' documentation.

## Construction

`hw = coder.hardware(name)` returns the configuration object for the target hardware `name`. You can use this object as the hardware configuration object for PIL execution with MATLAB Coder.

## Input Arguments

**name — Name of target hardware**
character vector | `'BeagleBone Black'` | `'ARM Cortex-A9 (QEMU)'`

Name of target hardware, specified as a character vector. You can get the names of the installed hardware support packages by calling `coder.hardware()` with no input arguments.

Example: `'BeagleBone Black'`

Data Types: `char`

## Properties

**`CPUClockRate` — Clock rate of target hardware**
100 (default) | scalar

Clock rate of target hardware, stored as a scalar.

Data Types: `double`

**`Name` — Name of target hardware**
character vector

Name of target hardware, stored as a character vector. This name matches the input `name`. You can get the current installed hardware support packages by calling `coder.hardware()` with no input arguments.

Example: `'BeagleBone Black'`

Data Types: `char`

## Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

## Examples

### Create a Hardware Configuration Object for ARM Cortex-A9 Hardware

```
hw = coder.hardware('ARM Cortex-A9 (QEMU)');
disp(hw)
```

```
  Hardware with properties:

           Name: 'ARM Cortex-A9 (QEMU)'
    CPUClockRate: 1000
```

#### Set Hardware Configuration to Coder Configuration Object

Dynamic properties for device address, user name, and password need to be specified for the BeagleBone Black.

Create hardware configuration object and set properties.

```
hw = coder.hardware('BeagleBone Black');
hw.DeviceAddress = '192.168.1.100'
hw.UserName = 'admin';
hw.Password = 'password';
disp(hw)
```

```
  Hardware with properties:

            Name: 'BeagleBone Black'
     CPUClockRate: 1000
    DeviceAddress: '192.168.1.100'
         Username: 'admin'
         Password: 'password'
```

Set hardware configuration object to coder configuration object.

```
cfg = coder.config('lib','ecoder',true);
cfg.VerificationMode = 'PIL';
cfg.Hardware = hw;
```

- “PIL Execution with ARM Cortex-A at the Command Line”
- “PIL Execution with ARM Cortex-A by Using the MATLAB Coder App”

## See Also

`codegen` | `coder.config` | `coder.hardware`

### Topics

“PIL Execution with ARM Cortex-A at the Command Line”

"PIL Execution with ARM Cortex-A by Using the MATLAB Coder App"

**Introduced in R2015b**

# RTW.TflBlasEntryGenerator

**Package:** RTW

Create code replacement table entry for a BLAS operation

## Syntax

`obj = RTW.TflBlasEntryGenerator`

## Description

`obj = RTW.TflBlasEntryGenerator` creates a handle, *`obj`*, to a code replacement table entry for a BLAS operator. The entry maps a conceptual representation of an operator to an implementation (replacement) representation.

## Examples

### Create Table Entry for BLAS Operator

This example shows how to create a code replacement table entry for a BLAS operator, `op_entry`.

```
hTable = RTW.TflTable;

arch = computer('arch');
compilerName = 'microsoft';
LibPath = fullfile('$(MATLAB_ROOT)', 'extern', ...
    'lib', arch, compilerName);

op_entry = RTW.TflBlasEntryGenerator;

libExt = 'lib';

setTflCOperationEntryParameters(op_entry, ...
    'Key',                      'RTW_OP_MUL', ...
```

```
    'Priority',                   100, ...
    'ImplementationName',         'dgemm32', ...
    'ImplementationHeaderFile',   'blascompat32_crl.h', ...
    'ImplementationHeaderPath',   fullfile('$(MATLAB_ROOT)','extern','include'), ...
    'AdditionalLinkObjs',         {['libmwblascompat32.' libExt]}, ...
    'AdditionalLinkObjsPaths',    {LibPath}, ...
    'SideEffects',                true);
```

## Output Arguments

**obj — Handle to code replacement table entry for a BLAS operator**
handle

The *obj* is a handle to the created code replacement table entry for a BLAS operator.

## See Also

RTW.TflCBlasEntryGenerator | RTW.TflCOperationEntry | RTW.TflTable

### Topics

"Define Code Replacement Mappings"
"Matrix Multiplication Operation to MathWorks BLAS Code Replacement"
"Code You Can Replace from MATLAB Code"
"Code You Can Replace From Simulink Models"

**Introduced in R2010a**

# RTW.TflCBlasEntryGenerator

**Package:** RTW

Create code replacement table entry for a CBLAS operation

## Syntax

`obj = RTW.TflCBlasEntryGenerator`

## Description

`obj = RTW.TflCBlasEntryGenerator` creates a handle, *`obj`*, to a code replacement table entry for a CBLAS operator. The entry maps a conceptual representation of an operator to an implementation (replacement) representation.

## Examples

### Create Table Entry for CBLAS Operator

This example shows how to create a code replacement table entry for a CBLAS operator, `hEnt`.

```
hTable = RTW.TflTable;

arch = computer('arch');
compilerName = 'my_compiler';
LibPath = fullfile('$(MATLAB_ROOT)', 'extern', ...
    'lib', arch, compilerName);

op_entry = RTW.TflCBlasEntryGenerator;

libExt = 'lib';

setTflCOperationEntryParameters(op_entry, ...
    'Key',                      'RTW_OP_MUL', ...
```

```
    'Priority',                  100, ...
    'ImplementationName',        'dgemm32', ...
    'ImplementationHeaderFile', 'my_cblas_compatible_crl.h', ...
    'ImplementationHeaderPath', fullfile('$(MATLAB_ROOT)','extern','include'), ...
    'AdditionalLinkObjs',        {['my_lib_cblas_compatible.' libExt]}, ...
    'AdditionalLinkObjsPaths',   {LibPath}, ...
    'SideEffects',               true);
```

## Output Arguments

**obj — Handle to code replacement table entry for a CBLAS operator**
handle

The *obj* is a handle to the created code replacement table entry for a CBLAS operator.

## See Also

RTW.TflBlasEntryGenerator | RTW.TflCOperationEntry | RTW.TflTable

### Topics

"Define Code Replacement Mappings"
"Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement"
"Code You Can Replace from MATLAB Code"
"Code You Can Replace From Simulink Models"

**Introduced in R2010a**

# RTW.TflCFunctionEntry

**Package:** RTW

Create code replacement table entry for a function

## Syntax

`obj = RTW.TflCFunctionEntry`

## Description

`obj = RTW.TflCFunctionEntry` creates a handle, *`obj`*, to a code replacement table entry for a function. The entry maps a conceptual representation of a function to an implementation (replacement) representation.

## Examples

### Create Table Entry for Function

This example shows how to create a code replacement table entry for a function, `hEnt`.

```
hEnt = RTW.TflCFunctionEntry;
```

## Output Arguments

**`obj` — Handle to code replacement table entry for a function**
handle

The *`obj`* is a handle to the created code replacement table entry for a function.

## See Also

`RTW.TflCFunctionEntryML` | `RTW.TflTable`

### Topics

"Define Code Replacement Mappings"
"Math Function Code Replacement"
"Memory Function Code Replacement"
"Nonfinite Function Code Replacement"
"Lookup Table Function Code Replacement"
"Code You Can Replace from MATLAB Code"
"Code You Can Replace From Simulink Models"

**Introduced in R2007b**

# RTW.TflCFunctionEntryML

Base class for custom code replacement table function entry

## Syntax

`RTW.TflCFunctionEntryML`

## Description

Derive a class from `RTW.TflCFunctionEntryML` to represent your custom function entry.

## Examples

"Customize Match and Replacement Process"

## See Also

`RTW.TflCFunctionEntry` | `RTW.TflTable`

### Topics

"Define Code Replacement Mappings"
"Customize Match and Replacement Process"
"Code You Can Replace from MATLAB Code"
"Code You Can Replace From Simulink Models"

# RTW.TflCOperationEntry

**Package:** RTW

Create code replacement table entry for an operator

## Syntax

```
obj = RTW.TflCOperationEntry
```

## Description

`obj = RTW.TflCOperationEntry` creates a handle, *obj*, to a code replacement table entry for an operator. The entry maps a conceptual representation of an operator to an implementation (replacement) representation.

## Examples

### Create Table Entry for Operator

This example shows how to create a code replacement table entry for an operator, `hEnt`.

```
hEnt = RTW.TflCOperationEntry;
```

## Output Arguments

**obj — Handle to code replacement table entry for an operator**
handle

The *obj* is a handle to the created code replacement table entry for an operator.

## See Also

`RTW.TflCOperationEntryGenerator` | `RTW.TflCOperationEntryGenerator_NetSlope` | `RTW.TflCOperationEntryML` | `RTW.TflTable`

### Topics

"Define Code Replacement Mappings"
"Scalar Operator Code Replacement"
"Addition and Subtraction Operator Code Replacement"
"Small Matrix Operation to Processor Code Replacement"
"Code You Can Replace from MATLAB Code"
"Code You Can Replace From Simulink Models"

**Introduced in R2007b**

# RTW.TflCOperationEntryGenerator

**Package:** RTW

Create code replacement table entry for a fixed-point addition or subtraction operation

## Syntax

`obj = RTW.TflCOperationEntryGenerator`

## Description

`obj = RTW.TflCOperationEntryGenerator` creates a handle, *`obj`*, to a code replacement table entry for a fixed-point addition or subtraction operation. The entry maps a conceptual representation of an operator to an implementation (replacement) representation.

## Examples

### Create Table Entry for Fixed-Point Add or Subtract Operation

This example shows how to create a code replacement table entry for a fixed-point addition or subtraction operation, `hEnt`.

```
hEnt = RTW.TflCOperationEntryGenerator;
```

## Output Arguments

**`obj` — Handle to code replacement table entry for a fixed-point addition or subtraction operation**
handle

The *`obj`* is a handle to the created code replacement table entry for a fixed-point addition or subtraction operation.

## See Also

`RTW.TflCOperationEntry` | `RTW.TflCOperationEntryGenerator_NetSlope` | `RTW.TflCOperationEntryML` | `RTW.TflTable`

### Topics

"Define Code Replacement Mappings"
"Fixed-Point Operator Code Replacement"
"Binary-Point-Only Scaling Code Replacement"
"Slope Bias Scaling Code Replacement"
"Code You Can Replace from MATLAB Code"
"Code You Can Replace From Simulink Models"

**Introduced in R2008a**

# RTW.TflCOperationEntryGenerator_NetSlope

**Package:** RTW

Create code replacement table entry for a net slope fixed-point operation

## Syntax

```
obj = RTW.TflCOperationEntryGenerator_NetSlope
```

## Description

`obj = RTW.TflCOperationEntryGenerator_NetSlope` creates a handle, *`obj`*, to a code replacement table entry for a net slope fixed-point operation. The entry maps a conceptual representation of an operator to an implementation (replacement) representation.

## Examples

### Create Table Entry for Net Slope Fixed-Point Operation

This example shows how to create a code replacement table entry for a net slope fixed-point operation, `hEnt`.

```
hEnt = RTW.TflCOperationEntryGenerator_NetSlope;
```

## Output Arguments

**`obj` — Handle to code replacement table entry for a net slope fixed-point operation**
handle

The *`obj`* is a handle to the created code replacement table entry for a net slope fixed-point operation.

## See Also

`RTW.TflCOperationEntry` | `RTW.TflCOperationEntryGenerator` | `RTW.TflCOperationEntryML`

### Topics

“Define Code Replacement Mappings”
“Fixed-Point Operator Code Replacement”
“Net Slope Scaling Code Replacement”
“Equal Slope and Zero Net Bias Code Replacement”
“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”

**Introduced in R2008b**

# RTW.TflCOperationEntryML

Base class for custom code replacement table operator entry

## Syntax

`RTW.TflCOperationEntryML`

## Description

Derive a class from `RTW.TflCOperationEntryML` to represent your custom operator entry.

## Examples

"Customize Code Match and Replacement for Scalar Operations"

## See Also

`RTW.TflCOperationEntry` | `RTW.TflTable`

### Topics

"Define Code Replacement Mappings"
"Customize Match and Replacement Process"
"Code You Can Replace from MATLAB Code"
"Code You Can Replace From Simulink Models"

# RTW.TflCSemaphoreEntry

**Package:** RTW

Create code replacement table entry for a semaphore or mutex

## Syntax

`obj = RTW.TflCSemaphoreEntry`

## Description

`obj = RTW.TflCSemaphoreEntry` creates a handle, *`obj`*, to a code replacement table entry for a semaphore or mutex. The entry maps a conceptual representation of a semaphore or mutex to an implementation (replacement) representation.

## Examples

### Create Table Entry for Semaphore or Mutex

This example shows how to create a code replacement table entry for a semaphore or mutex, `hEnt`.

```
hEnt = RTW.TflCSemaphoreEntry;
```

## Output Arguments

**`obj` — Handle to code replacement table entry for a semaphore or mutex**
handle

The *`obj`* is a handle to the created code replacement table entry for a semaphore or mutex.

## See Also

### Topics
"Define Code Replacement Mappings"
"Semaphore and Mutex Function Replacement"
"Code You Can Replace from MATLAB Code"
"Code You Can Replace From Simulink Models"

**Introduced in R2010a**

# RTW.TflTable

**Package:** RTW

Create code replacement table

## Syntax

```
obj = RTW.TflTable
```

## Description

`obj = RTW.TflTable` creates a handle, *`obj`*, to a code replacement table.

## Examples

### Create a Code Replacement Table

This example shows how to create a code replacement table object, `hTable`.

```
hTable = RTW.TflTable;
```

## Output Arguments

**`obj` — Handle to code replacement table**
handle

The *`obj`* is a handle to the created code replacement table.

## See Also

### Topics
“Define Code Replacement Mappings”
“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”

**Introduced in R2007b**

# Time

Get simulation time for code section

## Syntax

`SimTime = NthSectionProfile.Time`

## Description

`SimTime = NthSectionProfile.Time` returns a simulation time vector that corresponds to the execution time measurements for the code section.

## Examples

### Get Simulation Time for Code Section

Run a simulation with a model that is configured to generate a workspace variable with execution time measurements.

```
rtwdemo_sil_topmodel;
set_param('rtwdemo_sil_topmodel',...
          'CodeExecutionProfiling', 'on');
set_param('rtwdemo_sil_topmodel',...
          'SimulationMode', 'software-in-the-loop (SIL)');
set_param('rtwdemo_sil_topmodel',...
          'CodeProfilingInstrumentation', 'on');
set_param('rtwdemo_sil_topmodel',...
          'CodeProfilingSaveOptions', 'AllData');
sim('rtwdemo_sil_topmodel');
```

The simulation generates the workspace variable `executionProfile` (default).

At the end of the simulation, get profile for the seventh code section.

```
seventhSectionProfile = executionProfile.Sections(7);
```

Get vector representing simulation time for code section.

```
simulationTimeVector = seventhSectionProfile.Time;
```

## Input Arguments

### `NthSectionProfile` — `coder.profile.ExecutionTimeSection`
object

Object generated by the `coder.profile.ExecutionTime` property `Sections`.

## Output Arguments

### `SimTime` — Simulation time
double

Simulation time, in seconds, for section of code. Returned as a vector.

## See Also
`ExecutionTimeInSeconds` | `ExecutionTimeInTicks` | `Sections`

### Topics
“Code Execution Profiling with SIL and PIL”
“Analyze Code Execution Data”

**Introduced in R2013a**

# Time

Time over which code section execution time measurements are made

## Syntax

`Time = NthSectionProfile.Time`

## Description

`Time = NthSectionProfile.Time` returns a time vector corresponding to the period over which execution times are measured for the code section.

## Examples

### Get Time Vector for Code Section

Copy MATLAB code to your working folder.

```
src_dir = ...
    fullfile(docroot,'toolbox','coder','examples','kalman');

copyfile(fullfile(src_dir,'kalman01.m'), '.')
copyfile(fullfile(src_dir,'test01_ui.m'), '.')
copyfile(fullfile(src_dir,'plot_trajectory.m'), '.')
copyfile(fullfile(src_dir,'position.mat'), '.')
```

Set up and run a SIL execution.

```
config = coder.config('lib');
config.GenerateReport = true;

config.VerificationMode = 'SIL';
config.CodeExecutionProfiling = true;

codegen('-config', config, '-args', {zeros(2,1)}, 'kalman01');
```

```
coder.runTest('test01_ui', ['kalman01_sil.' mexext]);
```

At end of the execution, you see the following message.

```
To terminate execution: clear kalman01_sil
Execution profiling report available after termination.
```

Click the link `clear kalman01_sil`.

```
### Stopping SIL execution for 'kalman01'
    Execution profiling report: report(getCoderExecutionProfile('kalman01'))
```

Create a workspace variable that holds execution time data.

```
executionProfile=getCoderExecutionProfile('kalman01');
```

Get the profile for the second code section.

```
secondSectionProfile = executionProfile.Sections(2);
```

Get time vector for code section.

```
time = secondSectionProfile.Time;
```

## Input Arguments

**NthSectionProfile — coder.profile.ExecutionTimeSection**
object

Object generated by the `coder.profile.ExecutionTime` property `Sections`.

## Output Arguments

**Time — Time**
double

Time, in seconds, over which measurements are made for code section. Returned as a vector.

## See Also

`ExecutionTimeInSeconds` | `ExecutionTimeInTicks` | `Sections` | `getCoderExecutionProfile`

### Topics

"Generate Execution Time Profile"
"Analyze Execution Time Data"

**Introduced in R2013a**

# timeline

Display invocations of code sections over execution timeline

## Syntax

```
timeline(executionProfile)
timeline(executionProfile, 'MaxResizeIncrement', numberOfPoints)
```

## Description

`timeline(executionProfile)` displays invocations of each profiled code section over the execution timeline.

`timeline(executionProfile, 'MaxResizeIncrement', numberOfPoints)` specifies the maximum increment by which you:

- Increase the number of displayed points when you click the zoom-out tool.
- Move along the `timeline` plot when you sweep right or left with the pan tool.

Use this command when you want to review large `timeline` plots quickly.

## Examples

### Display Code Section Invocations

Run a simulation with a model that is configured to generate a workspace variable with execution-time measurements.

```
rtwdemo_sil_topmodel;
set_param('rtwdemo_sil_topmodel',...
          'CodeExecutionProfiling', 'on');
set_param('rtwdemo_sil_topmodel',...
          'SimulationMode', 'software-in-the-loop (SIL)');
set_param('rtwdemo_sil_topmodel',...
```

```
            'CodeProfilingInstrumentation', 'on');
set_param('rtwdemo_sil_topmodel',...
            'CodeProfilingSaveOptions', 'AllData');
sim('rtwdemo_sil_topmodel');
```

The simulation generates the workspace variable `executionProfile` (default).

At the end of the simulation, open a code execution report.

```
report(executionProfile)
```

Under **Profiled Sections of Code**, in the **Model** column, expand all nodes. You see profile information for eight code sections. For example, the task `rtwdemo_sil_topmodel_step` and functions `CounterTypeA` and `CounterTypeB`.

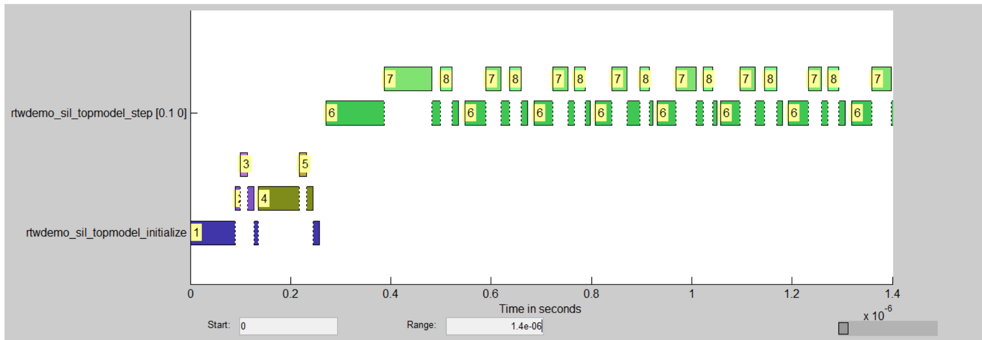
**2. Profiled Sections of Code**

| Model | Maximum Execution Time | Average Execution Time | Maximum Self Time | Average Self Time | Calls | |
|---|---|---|---|---|---|---|
| [-] rtwdemo_sil_topmodel_initialize | 257 | 257 | 111 | 111 | 1 | |
| [-] CounterTypeA | 38 | 38 | 23 | 23 | 1 | |
| CounterTypeA | 15 | 15 | 15 | 15 | 1 | |
| [-] CounterTypeB | 109 | 109 | 94 | 94 | 1 | |
| CounterTypeB | 15 | 15 | 15 | 15 | 1 | |
| [-] rtwdemo_sil_topmodel_step [0.1 0] | 265 | 121 | 147 | 61 | 101 | |
| CounterTypeA | 94 | 36 | 94 | 36 | 101 | |
| CounterTypeB | 37 | 24 | 37 | 24 | 101 | |

Display code section invocations.

```
timeline(executionProfile)
```

In the Execution Profile window, you see numbered horizontal bars that represent invocations of the code sections.



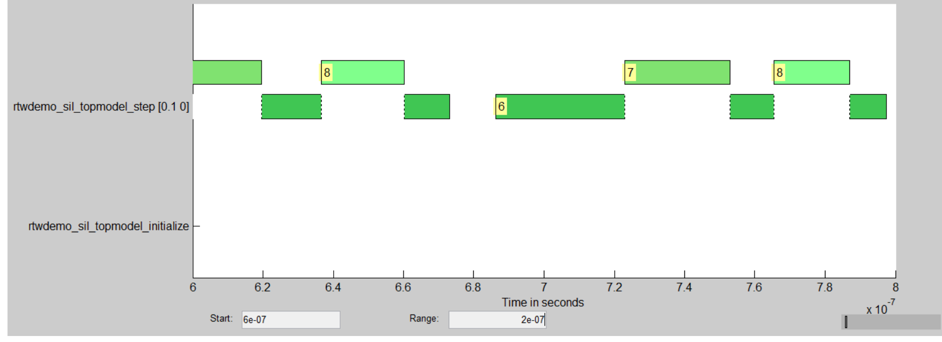

For example, the blue bars show when the first section, `rtwdemo_sil_topmodel_initialize`, is invoked.

To see the first code section, in the first row of the Code Execution Profiling Report, click the icon .

The Code Generation Report displays the function call.

```
64          PROFILE_START_TASK_SECTION(1U);
65          rtwdemo_sil_topmodel_initialize();
66          PROFILE_END_TASK_SECTION(1U);
```

To see what code sections are invoked over a specific time period, use the **Start** and **Range** fields of the Execution Profile window. For example, in the **Start** and **Range** fields, enter `6e-07` and `2e-07` respectively. Then press **Enter**.




Between 0.6 μs and 0.8 μs, you see that the task `rtwdemo_sil_topmodel_step` (code section 6) and the functions `CounterTypeA` (code section 7) and `CounterTypeB` (code section 8) are invoked.

On the bottom right of the Execution Profile window, the indicator shows what portion of the execution timeline is being displayed.

## Input Arguments

**`executionProfile` — `coder.profile.ExecutionTime`**
object

When you run a simulation with code execution profiling, the software generates `executionProfile` as a workspace variable.

**`numberOfPoints` — Number of points**
20 (default) | integer

Maximum increment for zoom-out and pan tools.

## See Also

`report`

### Topics

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

**Introduced in R2013b**

# TimerTicksPerSecond

Get and set number of timer ticks per second

## Syntax

```
timerTicksPerSecVal = myExecutionProfile.TimerTicksPerSecond
myExecutionProfile.TimerTicksPerSecond = timerTicksPerSecVal
```

## Description

`timerTicksPerSecVal = myExecutionProfile.TimerTicksPerSecond` returns the number of timer ticks per second. For example, if the timer runs at 1 MHz, then the number of ticks per second is $10^6$.

`myExecutionProfile.TimerTicksPerSecond = timerTicksPerSecVal` sets the number of timer ticks per second. Use this method if the "Create PIL Target Connectivity Configuration for Simulink" does not specify this value.

*`myExecutionProfile`* is a workspace variable generated by a simulation.

**Tip** You can calculate the execution time in seconds using the formula

$ExecutionTimeInSecs = ExecutionTimeInTicks / TimerTicksPerSecond$ .

## Input Arguments

***`timerTicksPerSecVal`***

Number of timer ticks per second

## Output Arguments

***timerTicksPerSecVal***

Number of timer ticks per second

## See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `display` | `report`

### Topics

"Code Execution Profiling with SIL and PIL"
"View and Compare Code Execution Times"
"Analyze Code Execution Data"

**Introduced in R2012b**

# TimerTicksPerSecond

Get and set number of timer ticks per second

## Syntax

```
timerTicksPerSecVal = myExecutionProfile.TimerTicksPerSecond
myExecutionProfile.TimerTicksPerSecond = timerTicksPerSecVal
```

## Description

`timerTicksPerSecVal = myExecutionProfile.TimerTicksPerSecond` returns the number of timer ticks per second. For example, if the timer runs at 1 MHz, then the number of ticks per second is $10^6$.

`myExecutionProfile.TimerTicksPerSecond = timerTicksPerSecVal` sets the number of timer ticks per second. Use this method if the target connectivity configuration does not specify this value.

*`myExecutionProfile`* is a workspace variable that you create using `getCoderExecutionProfile`.

---

**Tip** You can calculate the execution time in seconds using the formula

$ExecutionTimeInSecs = ExecutionTimeInTicks / TimerTicksPerSecond$ .

---

## Input Arguments

***`timerTicksPerSecVal`***

Number of timer ticks per second

## Output Arguments

***timerTicksPerSecVal***

Number of timer ticks per second

## See Also

ExecutionTimeInTicks | MaximumExecutionTimeCallNum | MaximumExecutionTimeInTicks | MaximumSelfTimeCallNum | MaximumSelfTimeInTicks | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeInTicks | Name | NumCalls | Number | Sections | SelfTimeInTicks | TotalExecutionTimeInTicks | TotalSelfTimeInTicks | TotalTurnaroundTimeInTicks | TurnaroundTimeInTicks | getCoderExecutionProfile | report

### Topics

"Generate Execution Time Profile"
"Analyze Execution Time Data"
"Create PIL Target Connectivity Configuration for MATLAB"

**Introduced in R2012b**

# coder.MATLABCodeTemplate.getTokenValue

**Class:** coder.MATLABCodeTemplate
**Package:** coder

Get value of token

## Syntax

```
tokenValue = getTokenValue(tokenName)
```

## Description

`tokenValue = getTokenValue(tokenName)` returns the value of the specified token.

## Input Arguments

**`tokenName`**

Name of token

**Default:** empty

## Output Arguments

**`tokenValue` — Token value**
character vector

The current value of `tokenName`, returned as a character vector.

## Examples

Create a MATLABCodeTemplate object with the default template, then get the value for a token.

```
newObj = coder.MATLABCodeTemplate;
% Creates a MATLABCodeTemplate object from the default template
newObj.getCurrentTokens()
% Get list of current tokens
newObj.getTokenValue('MATLABCoderVersion')
% Check value of a token
```

## See Also

coder.MATLABCodeTemplate.emitSection | coder.MATLABCodeTemplate.getCurrentTokens | coder.MATLABCodeTemplate.setTokenValue

### Topics

"Generate Custom File and Function Banners for C/C++ Code"
"Code Generation Template Files for MATLAB Code"

# halt

Halt program execution by processor

## Syntax

```
halt(IDE_Obj)
halt(IDE_Obj,timeout)
```

## IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

## Description

`halt(IDE_Obj)` stops the program running on the processor. After you issue this command, MATLAB waits for a response from the processor that the processor has stopped. By default, the wait time is 10 seconds. If 10 seconds elapses before the response arrives, MATLAB returns an error. In this syntax, the timeout period defaults to the global timeout period specified in `IDE_Obj`. Use `get(IDE_Obj)` to determine the global timeout period. However, the processor usually stops in spite of the error message.

To resume processing after you halt the processor, use `run`. Also, the `read(IDE_Obj,'pc')` function can determine the memory address where the processor stopped after you use `halt`.

`halt(IDE_Obj,timeout)` immediately stops program execution by the processor. After the processor stops, `halt` returns to the host. `timeout` defines, in seconds, how long the host waits for the processor to stop running. If the processor does not stop within the specified timeout period, the routine returns with a timeout error.

## Examples

Use one of the provided example programs to show how halt works. Load and run one of the example projects. At the MATLAB prompt, check whether the program is running on the processor.

```
isrunning(IDE_Obj)

ans  =

     1

halt(IDE_Obj) % Stop the running application on the processor.
isrunning(IDE_Obj)

ans  =

     0
```

Issuing the halt stops the process on the processor. Checking in the IDE confirms that the process has stopped.

# See Also

isrunning | reset | run

**Introduced in R2011a**

# info

Information about processor

## Syntax

```
adf = info(IDE_Obj)
adf = info(IDE_Obj)
adf = info(rx)
adf = info(IDE_Obj)
adf = info(rx)
```

## IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

## Description

`adf = info(IDE_Obj)` returns debugger or processor properties associated with the IDE handle object, `IDE_Obj`.

### Using info with Multiprocessor Boards

For multiprocessor targets, the `info` method returns properties for each processor with the array.

Using info with `IDE_Obj`, which is associated with 1 processor:

```
oinfo = info(IDE_Obj);
```

Using info with `IDE_Obj`, which is associated with 2 processors:

```
oinfo = info(IDE_Obj); % Returns a 1x2 array of infor struct
```

## Using info with CCS IDE

`adf = info(IDE_Obj)` returns the property names and property values associated with the processor accessed by *`IDE_Obj`*. `adf` is a structure containing the following information elements and values.

| Structure Element | Data Type | Description |
|---|---|---|
| `adf.procname` | Character vector | Processor name as defined in the CCS setup utility. In multiprocessor systems, this name reflects the specific processor associated with *`IDE_Obj`*. |
| `adf.isbigendian` | Boolean | Value describing the byte ordering used by the processor. When the processor is big-endian, this value is `1`. Little-endian processors return `0`. |
| `adf.family` | Integer | Three-digit integer that identifies the processor family, ranging from 000 to 999. For example, 320 for Texas Instruments digital signal processors. |
| `adf.subfamily` | Decimal | Decimal representation of the hexadecimal identification value that TI assigns to the processor to identify the processor subfamily. IDs range from 0x000 to 0x3822. Use `dec2hex` to convert the value in `adf.subfamily` to standard notation. For example<br>`dec2hex(adf.subfamily)`<br>produces `'67'` when the processor is a member of the 67xx processor family. |
| `adf.timeout` | Integer | Default timeout value MATLAB software uses when transferring data to and from CCS. Functions that use a timeout value have an optional *`timeout`* input argument. When you omit the optional argument, MATLAB software uses 10s as the default value. |

`adf = info(rx)` returns info as a cell arraying containing the names of your open RTDX channels.

On a PC with a simulator configured in CCS IDE, `info` returns the configuration for the processor being simulated:

```
info(IDE_Obj)

ans =

       procname: 'CPU'
    isbigendian: 0
         family: 320
      subfamily: 103
        timeout: 10
```

This example simulates the TMS320C62xx processor running in little-endian mode. When you use CCS Setup Utility to change the processor from little-endian to big-endian, `info` shows the change.

```
info(IDE_Obj)

ans =

       procname: 'CPU'
    isbigendian: 1
         family: 320
      subfamily: 103
        timeout: 10
```

If you have two open channels, `chan1` and `chan2`,

```
adf = info(rx)
```

returns

```
adf =
'chan1'
'chan2'
```

where `adf` is a cell array. You can dereference the entries in `adf` to manipulate the channels. For example, you can close a channel by dereferencing the channel in `adf` in the close function syntax.

```
close(rx.adf{1,1})
```

## Using info with VisualDSP++ IDE

`adf = info(IDE_Obj)` returns the property names and property values associated with the processor accessed by *`IDE_Obj`*. The `adf` variable is a structure containing the following information elements and values.

| Structure Element | Data Type | Description |
| --- | --- | --- |
| `adf.procname` | Character vector | Processor name as defined in the CCS setup utility. In multiprocessor systems, this name reflects the specific processor associated with *`IDE_Obj`*. |
| `adf.proctype` | Character vector | Character vector with the type of the DSP processor. The type property is the processor type like "ADSP-21065L" or "ADSP-2181". |
| `adf.revision` | Character vector | Character vector with the silicon revision string of the processor. |

`adf = info(rx)` returns info as a cell arraying containing the names of your open RTDX channels.

When you have an adivdsp object IDE_Obj, info provides information about the object:

```
IDE_Obj = adivdsp('sessionname','Testsession')

ADIVDSP Object:
  Session name     : Testsession
  Processor name   : ADSP-BF533
  Processor type   : ADSP-BF533
  Processor number : 0
  Default timeout  : 10.00 secs

objinfo = info(IDE_Obj)

objinfo =

    procname: 'ADSP-BF533'
    proctype: 'ADSP-BF533'
    revision: ''

objinfo.procname

ans =
```

`ADSP-BF533`

## See Also

`dec2hex` | `get` | `set`

**Introduced in R2011a**

# insert

Insert debug point in file

## Syntax

```
insert(IDE_Obj,addr,type,timeout)
insert(IDE_Obj,addr)
```

## IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

## Description

`insert(IDE_Obj,addr,type,timeout)` places a debug point at the provided address of the processor. The IDE_Obj handle defines the processor that will receive the new debug point. The debug point location is defined by *addr*, the desired memory address. The IDEs support several types of debug points. Refer to your IDE help documentation for information on their respective behavior. The following table shows which debug types each IDE supports.

| | CCS IDE | VisualDSP++ |
|---|---|---|
| `'break'` (default) | Yes | Yes |
| `'watch'` | | |
| `'probe'` | Yes | |

The *timeout* parameter defines how long to wait (in seconds) for the insert to complete. If this period is exceeded, the routine returns immediately with a timeout error. In general the action (insert) still occurs, but the timeout value gave insufficient time to verify the completion of the action.

`insert(IDE_Obj,addr)` same as the preceding example, except the *timeout* value defaults to the timeout property specified by the IDE_Obj object. Use `get(IDE_Obj,'timeout')` to examine this default timeout value.

With `insert(IDE_Obj,file,line)` the timeout value defaults to the timeout property specified by the IDE_Obj object. Use `get(IDE_Obj,'timeout')` to examine this default timeout value.

## See Also

`address` | `run`

**Introduced in R2011a**

# isenabled

Determine whether RTDX link is enabled for communications

---

**Note** Support for `isenabled` on C5000 processors will be removed in a future version.

---

## Syntax

```
isenabled(rx,'channel')
isenabled(rx)
```

## IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

## Description

`isenabled(rx,'channel')` returns `ans = 1` when the RTDX channel specified by character vector `'channel'` is enabled for read or write communications. When `'channel'` has not been enabled, `isenabled` returns `ans = 0`.

`isenabled(rx)` returns `ans = 1` when RTDX has been enabled, independent of a channel. When you have not enabled RTDX you get `ans = 0` back.

### Important Requirements for Using isenabled

On the processor side, `isenabled` depends on RTDX to determine and report the RTDX status. Therefore the you must meet the following requirements to use `isenabled`.

1 The processor must be running a program when you query the RTDX interface.

2. You must enable the RTDX interface before you check the status of individual channels or the interface.
3. Your processor program must be polling periodically for `isenabled` to work.

---

**Note** For `isenabled` to return valid results, your processor must be running a loaded program. When the processor is not running, `isenabled` returns a status that may not represent the true state of the channels or RTDX.

---

## Examples

With a program loaded on your processor, you can determine whether RTDX channels are ready for use. Restart your program to be sure it is running. The processor must be running for `isenabled` and `enabled` to function. This example creates a `ticcs` object `IDE_Obj` to begin.

```
restart(IDE_Obj)
run(IDE_Obj,'run');
rtdx.enable(IDE_Obj,'ichan');
rtdx.isenabled(IDE_Obj,'ichan')
```

MATLAB software returns 1 indicating that your channel `'ichan'` is enabled for RTDX communications. To determine the mode for the channel, use `rtdx(IDE_Obj)` to display the object properties.

## See Also

`disable` | `enable` | `clear`

**Introduced in R2011a**

# isreadable

Determine whether specified memory block can read MATLAB software

---

**Note** Support for `isreadable(rx,'channel')` on C5000 processors will be removed in a future version.

---

## Syntax

```
isreadable(IDE_Obj,address,'datatype',count)
isreadable(IDE_Obj,address,'datatype')
isreadable(rx,'channel')
```

## IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

## Description

`isreadable(IDE_Obj,address,'datatype',count)` returns 1 if the processor referred to by *IDE_Obj* can read the memory block defined by the *address*, *count*, and *datatype* input arguments. When the processor cannot read a portion of the specified memory block, `isreadable` returns 0. You use the same memory block specification for this function as you use for the read function.

The data block being tested begins at the memory location defined by *address*. *count* determines the number of values to be read. *datatype* defines the format of data stored in the memory block. `isreadable` uses the *datatype* character vector to determine the number of bytes to read per stored value. For details about each input parameter, read the following descriptions.

*address* — `isreadable` uses *address* to define the beginning of the memory block to read. You provide values for *address* as either decimal or hexadecimal representations of a memory location in the processor. The full address at a memory location consists of two parts: the offset and the memory page, entered as a vector `[location, page]`, a character vector, or a decimal value.

When the processor has only one memory page, as is true for many digital signal processors, the page portion of the memory address is 0. By default, `ticcs` sets the page to `0` at creation if you omit the page property as an input argument. For processors that have one memory page, setting the page value to 0 lets you specify memory locations in the processor using the memory location without the page value.

**Examples of Address Property Values**

| Property Value | Address Type | Interpretation |
| --- | --- | --- |
| `'1F'` | Character vector | Location is 31 decimal on the page referred to by `page(IDE_Obj)` |
| 10 | Decimal | Address is 10 decimal on the page referred to by `page(IDE_Obj)` |
| [18,1] | Vector | Address location 10 decimal on memory page 1 (`page(IDE_Obj) = 1`) |

To specify the address in hexadecimal format, enter the *address* property value as a character vector. `isreadable` interprets the character vector as the hexadecimal representation of the desired memory location. To convert the hex value to a decimal value, the function uses `hex2dec`. When you use the character vector option to enter the address as a hex value, you cannot specify the memory page. For character vector input, the memory page defaults to the page specified by `page(IDE_Obj)`.

*count* — A numeric scalar or vector that defines the number of *datatype* values to test for being readable. To produce parallel structure with `read`, *count* can be a vector to define multidimensional data blocks. This function tests a block of data whose size is the product of the dimensions of the input vector.

*datatype* — A character vector that represents a MATLAB software data type. The total memory block size is derived from the value of *count* and the *datatype* you specify. *datatype* determines how many bytes to check for each memory value. `isreadable` supports the following data types.

| datatype Value | Number of Bytes/Value | Description |
|---|---|---|
| 'double' | 8 | Double-precision floating point values |
| 'int8' | 1 | Signed 8-bit integers |
| 'int16' | 2 | Signed 16-bit integers |
| 'int32' | 4 | Signed 32-bit integers |
| 'single' | 4 | Single-precision floating point data |
| 'uint8' | 1 | Unsigned 8-bit integers |
| 'uint16' | 2 | Unsigned 16-bit integers |
| 'uint32' | 4 | Unsigned 32-bit integers |

Like the `iswritable`, `write`, and `read` functions, `isreadable` checks for valid address values. Illegal address values would be an address space larger than the available space for the processor:

- $2^{32}$ for the C6xxx series
- $2^{16}$ for the C5xxx series

When the function identifies an illegal address, it returns an error message stating that the address values are out of range.

`isreadable(`*`IDE_Obj`*`,`*`address`*`,'`*`datatype`*`')` returns 1 if the processor referred to by *`IDE_Obj`* can read the memory block defined by the *`address`*, and *`datatype`* input arguments. When the processor cannot read a portion of the specified memory block, `isreadable` returns 0. Notice that you use the same memory block specification for this function as you use for the read function. The data block being tested begins at the memory location defined by *`address`*. When you omit the *`count`* option, *`count`* defaults to one.

`isreadable(rx,'`*`channel`*`')` returns a 1 when the RTDX channel specified by the character vector *`channel`*, associated with link `rx`, is configured for `read` operation. When *`channel`* is not configured for reading, `isreadable` returns 0.

Like the `iswritable`, `read`, and `write` functions, `isreadable` checks for valid address values. Illegal address values are address spaces larger than the available space for the processor:

- $2^{32}$ for the C6xxx series
- $2^{16}$ for the C5xxx series

When the function identifies an illegal address, it returns an error message stating that the address values are out of range.

---

**Note** `isreadable` relies on the memory map option in the IDE. If you did not define the memory map for the processor in the IDE, `isreadable` does not produce useful results. Refer to your Texas Instruments Code Composer Studio documentation for information on configuring memory maps.

---

## Examples

When you write scripts to run models in the MATLAB environment and the IDE, the `isreadable` function is very useful. Use `isreadable` to check that the channel from which you are reading is configured.

```
IDE_Obj = ticcs;
rx = rtdx(IDE_Obj);

% Define read and write channels to the processor linked by IDE_Obj.
open(rx,'ichannel','r');s
open(rx,'ochannel','w');
enable(rx,'ochannel');
enable(rx,'ichannel');

isreadable(rx,'ochannel')
ans =
    0
isreadable(rx,'ichannel')
ans =
    1
```

Now that your script knows that it can read from `ichannel`, it proceeds to read messages as required.

## See Also

`iswritable` | `read` | `hex2dec`

**Introduced in R2011a**

# isrtdxcapable

Determine whether processor supports RTDX

---

**Note** Support for `isrtdxcapable` on C5000 processors will be removed in a future version.

---

## Syntax

```
b = isrtdxcapable(IDE_Obj)
```

## IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

## Description

`b = isrtdxcapable(IDE_Obj)` returns `b = 1` when the processor referenced by object *`IDE_Obj`* supports RTDX. When the processor does not support RTDX, `isrtdxcapable` returns `b = 0`.

### Using isrtdxcapable with Multiprocessor Boards

When your board contains more than one processor, `isrtdxcapable` checks each processor on the processor, as defined by the *`IDE_Obj`* object, and returns the RTDX capability for each processor on the board. In the returned variable `b`, you find a vector that contains the information for each accessed processor.

# Examples

Create a link to your C6711 DSK. Test to see if the processor on the board supports RTDX.

```
IDE_Obj = ticcs; %Assumes you have one board and it is the C6711 DSK.
b = isrtdxcapable(IDE_Obj)
b  =
   1
```

**Introduced in R2011a**

# isrunning

Determine whether processor is executing process

## Syntax

```
isrunning(IDE_Obj)
```

## IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

## Description

`isrunning(IDE_Obj)` returns `1` when the processor is executing a program. When the processor is halted, `isrunning` returns 0.

## Examples

`isrunning` lets you determine whether the processor is running. After you load a program to the processor, use `isrunning` to verify that the program is running.

```
load(IDE_Obj,'program.exe','program')
run(IDE_Obj)
isrunning(IDE_Obj)

ans  =

     1
halt(IDE_Obj)
isrunning(IDE_Obj)
```

```
ans  =

     0
```

## See Also

`halt` | `load` | `run`

**Introduced in R2011a**

# isvisible

Determine whether IDE appears on desktop

## Syntax

```
isvisible(IDE_Obj)
```

## IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

## Description

`isvisible(IDE_Obj)` returns `1` if the IDE is running on the desktop and the window is open. If the IDE is not running or is running in the background, this method returns `0`.

## Examples

First use a constructor to create an IDE handle object and start the IDE. To determine if the IDE is visible:

```
isvisible(IDE_Obj)    #determine if the ide is visible

ans  =

     1
visible(IDE_Obj,0)    #make the ide invisible
isvisible(IDE_Obj)    #determine if the ide is visible

ans  =
```

```
    0
```

Notice that the IDE is not visible on your desktop. Recall that MATLAB software did not open the IDE. When you close MATLAB software with the IDE in this invisible state, the IDE remains running in the background. To close it, perform either of the following tasks:

- Open MATLAB software. Create a link to the IDE. Use the new link to make the IDE visible. Close the IDE.
- Open Microsoft® Windows Task Manager. Click **Processes**. Find and highlight `IDE_Obj_app.exe`. Click **End Task**.

## See Also

`info` | `visible`

**Introduced in R2011a**

# iswritable

Determine whether MATLAB can write to specified memory block

---

**Note** Support for `iswritable(rx,'channel')` on C5000 processors will be removed in a future version.

---

## Syntax

```
iswritable(IDE_Obj,address,'datatype',count)
iswritable(IDE_Obj,address,'datatype')
iswritable(rx,'channel')
```

## IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

## Description

`iswritable(IDE_Obj,address,'datatype',count)` returns 1 if MATLAB software can write to the memory block defined by the `address`, `count`, and `datatype` input arguments on the processor referred to by `IDE_Obj`. When the processor cannot write to a portion of the specified memory block, `iswritable` returns 0. You use the same memory block specification for this function as you use for the write function.

The data block being tested begins at the memory location defined by `address`. `count` determines the number of values to write. `datatype` defines the format of data stored in the memory block. `iswritable` uses the `datatype` parameter to determine the number of bytes to write per stored value. For details about each input parameter, read the following descriptions.

`address` — `iswritable` uses `address` to define the beginning of the memory block to write to. You provide values for `address` as either decimal or hexadecimal representations of a memory location in the processor. The full address at a memory location consists of two parts: the offset and the memory page, entered as a vector [*location*, *page*], a character vector, or a decimal value. When the processor has only one memory page, as is true for many digital signal processors, the page portion of the memory address is 0. By default, `ticcs` sets the page to `0` at creation if you omit the page property as an input argument.

For processors that have one memory page, setting the page value to 0 lets you specify memory locations in the processor using the memory location without the page value.

**Examples of Address Property Values**

| Property Value | Address Type | Interpretation |
| --- | --- | --- |
| 1F | Character vector | Location is 31 decimal on the page referred to by `page(IDE_Obj)` |
| 10 | Decimal | Address is 10 decimal on the page referred to by `page(IDE_Obj)` |
| [18,1] | Vector | Address location 10 decimal on memory page 1 (`page(IDE_Obj) = 1`) |

To specify the address in hexadecimal format, enter the address property value as a character vector. `iswritable` interprets the character vector as the hexadecimal representation of the desired memory location. To convert the hex value to a decimal value, the function uses `hex2dec`. When you use the character vector option to enter the address as a hex value, you cannot specify the memory page. For character vector input, the memory page defaults to the page specified by `page(IDE_Obj)`.

`count` — A numeric scalar or vector that defines the number of `datatype` values to test for being writable. To produce parallel structure with `write`, `count` can be a vector to define multidimensional data blocks. This function tests a block of data whose size is the total number of elements in matrix specified by the input vector. If `count` is the vector [10 10 10], then:

```
iswritable(IDE_Obj,31,[10 10 10])
```

`iswritable` writes 1000 values (10*10*10) to the processor. For a two-dimensional matrix defined with `count` as

`iswritable(IDE_Obj,31,[5 6])`

`iswritable` writes 30 values to the processor.

`datatype` — a character vector that represents a MATLAB data type. The total memory block size is derived from the value of `count` and the specified `datatype`. `datatype` determines how many bytes to check for each memory value. `iswritable` supports the following data types.

| datatype Value | Description |
|---|---|
| 'double' | Double-precision floating point values |
| 'int8' | Signed 8-bit integers |
| 'int16' | Signed 16-bit integers |
| 'int32' | Signed 32-bit integers |
| 'single' | Single-precision floating point data |
| 'uint8' | Unsigned 8-bit integers |
| 'uint16' | Unsigned 16-bit integers |
| 'uint32' | Unsigned 32-bit integers |

`iswritable(IDE_Obj,address,'datatype')` returns `1` if the processor referred to by `IDE_Obj` can write to the memory block defined by the `address`, and `count` input arguments. When the processor cannot write a portion of the specified memory block, `iswritable` returns `0`. Notice that you use the same memory block specification for this function as you use for the `write` function. The data block tested begins at the memory location defined by `address`. When you omit the `count` option, `count` defaults to one.

---

**Note** `iswritable` relies on the memory map option in the IDE. If you did not define the memory map for the processor in the IDE, this function does not produce useful results. Refer to your Texas Instruments Code Composer Studio documentation for information on configuring memory maps.

---

Like the `isreadable`, `read`, and `write` functions, `iswritable` checks for valid address values. Illegal address values would be an address space larger than the available space for the processor:

- $2^{32}$ for the C6xxx series

- $2^{16}$ for the C5xxx series

When the function identifies an illegal address, it returns an error message stating that the address values are out of range.

`iswritable(rx,'channel')` returns a Boolean value signifying whether the RTDX channel specified by `channel` and `rx`, is configured for write operations.

## Examples

When you write scripts to run models in MATLAB software and the IDE, the `iswritable` function is very useful. Use `iswritable` to check that the channel to which you are writing to is indeed configured.

```
IDE_Obj = ticcs;
rx = rtdx(IDE_Obj);

% Define read and write channels to the processor linked by IDE_Obj.
open(rx,'ichannel','r');
open(rx,'ochannel','w');
enable(rx,'ochannel');
enable(rx,'ichannel');

iswritable(rx,'ochannel')
ans =
    1
iswritable(rx,'ichannel')
ans =
    0
```

Now that your script knows that it can write to `'ichannel'`, it proceeds to write messages as required.

## See Also

`isreadable` | `read` | `hex2dec`

**Introduced in R2011a**

# list

Information listings from IDE

## Syntax

## IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

## Description

### Using list with CCS IDE

`infolist = list(`*`IDE_Obj`*`,type)`reads information about your CCS session and returns it in `infolist`. Different types of information and return formats apply depending on the input arguments you supply to the `list` function call. The `type` argument specifies which information listing to return. To determine the information that `list` returns, use one of the following as the `type` parameter character vector:

- **`project`** — Tell `list` to return information about the current project in CCS.
- **`variable`** — Tell `list` to return information about one or more embedded variables.
- **`globalvar`** — Tell `list` to return information about one or more global embedded variables.
- **`function`** — Tell `list` to return details about one or more functions in your project.

The `list` function returns dynamic CCS information that can be altered by the user. Returned listings represent snapshots of the current CCS configuration only. Be aware that earlier copies of `infolist` might contain stale information.

Also, `list` may report incorrect information when you make changes to variables from MATLAB software. To report variable information, `list` uses the CCS API, which only

knows about variables in CCS. Your changes from MATLAB software do not appear through the API and `list`. For example, the following operations return incorrect or old data information from `list`.

Suppose your original prototype is

```
unsigned short tgtFunction7(signed short signedShortArray1[]);
```

After creating the `function` object `fcnObj`, perform a `declare` operation with this character vector to change the declaration:

```
unsigned short tgtFunction7(unsigned short signedShortArray1[]);
```

Now try using `list` to return information about `signedShortArray1`.

```
list(fcnObj,'signedShortArray1')

 address: [3442 1]
location: [1x66 char]
    size: 1
 bitsize: 16
 reftype: 'short'
  referent: [1x1 struct]
member_pts_to_same_struct: 0
      name: 'signedShortArray1'
```

You get this outcome because `list` uses the CCS API to query information about a particular variable. As far as the API is concerned, the first input variable is a `short*`. Changing the declaration does not change anything.

When you specify option `type` as **`project`**, for example `infolist = list('project')`, the method returns a vector of structures that contain project information in the following format.

| infolist Structure Element | Description |
|---|---|
| `infolist(1).name` | Project file name (with path). |
| `infolist(1).type` | Project type — `project`, `projlib`, or `projext`, refer to `new`. |
| `infolist(1).processortype` | Character vector description of processor CPU. |

| infolist Structure Element | Description |
|---|---|
| `infolist(1).srcfiles` | Vector of structures that describes project source files. Each structure contains the name and path for each source file — `infolist(1).srcfiles.name`. |
| `infolist(1).buildcfg` | Vector of structures that describe build configurations, each with the following entries:<br>• `infolist(1).buildcfg.name` — the build configuration name.<br>• `infolist(1).buildcfg.outpath` — the default folder for storing the build output. |
| `infolist(2)....` | ... |
| `infolist(n)....` | ... |

`infolist = list(`*`IDE_Obj`*`,'variable')` returns a structure of structures that contains information on the local variables within scope. The list also includes information on the global variables. However, that if a local variable has the same symbol name as a global variable, list returns the information about the local variable.

`infolist = list(`*`IDE_Obj`*`,'variable',varname)` returns information about the specified variable `varname`.

`infolist = list(`*`IDE_Obj`*`,'variable',varnamelist)` returns information about variables in a list specified by `varnamelist`. The information returned in each structure follows the following format when you specify option `type` as **`variable`**.

| infolist Structure Element | Description |
|---|---|
| `infolist.varname(1).name` | Symbol name. |
| `infolist.varname(1).isglobal` | Indicates whether symbol is global or local. |
| `infolist.varname(1).location` | Information about the location of the symbol. |
| `infolist.varname(1).size` | Size per dimension. |
| `infolist.varname(1).uclass` | `ticcs` object class that matches the type of this symbol. |

| infolist Structure Element | Description |
| --- | --- |
| `infolist.varname(1).bitsize` | Size in bits. More information is added to the structure depending on the symbol type. |
| `infolist.varname(2)....` | ... |
| `infolist.varname(n)....` | ... |

`list` uses the variable name as the field name to refer to the structure information for the variable.

`infolist = list(`*`IDE_Obj`*`,'globalvar')` returns a structure that contains information on the global variables.

`infolist = list(`*`IDE_Obj`*`,'globalvar',varname)` returns a structure that contains information on the specified global variable.

`infolist = list(`*`IDE_Obj`*`,'globalvar',varnamelist)` returns a structure that contains information on global variables in the list. The returned information follows the same format as the syntax `infolist = list(`*`IDE_Obj`*`,'variable',...)`.

`infolist = list(`*`IDE_Obj`*`,'function')` returns a structure that contains information on the functions in the embedded program.

`infolist = list(`*`IDE_Obj`*`,'function',functionname)` returns a structure that contains information on the specified function `functionname`.

`infolist = list(`*`IDE_Obj`*`,'function',functionnamelist)` returns a structure that contains information on the specified functions in `functionnamelist`. The returned information follows the following format when you specify option `type` as **`function`**.

| infolist Structure Element | Description |
| --- | --- |
| `infolist.functionname(1).name` | Function name |
| `infolist.functionname(1).filename` | Name of file where function is defined |
| `infolist.functionname(1).address` | Relevant address information such as start address and end address |
| `infolist.functionname(1).funcvar` | Variables local to the function |

| infolist Structure Element | Description |
| --- | --- |
| infolist.functionname(1).uclass | ticcs object class that matches the type of this symbol — **function** |
| infolist.functionname(1).funcdecl | Function declaration — where information such as the function return type is contained |
| infolist.functionname(1).islibfunc | Determine if the library is a function |
| infolist.functionname(1).linepos | Start and end line positions of function |
| infolist.functionname(1).funcinfo | Miscellaneous information about the function |
| infolist.functionname(2)... | ... |
| infolist.functionname(n)... | ... |

To refer to the function structure information, list uses the function name as the field name.

The following list provides important information about variable and field names:

- When a variable name, type name, or function name is not a valid MATLAB software structure field name, list replaces or modifies the name so it becomes valid.
- In field names that contain the invalid dollar character $, list replaces the $ with DOLLAR.
- Changing the MATLAB software field name does not change the name of the embedded symbol or type.

To show how to use list with a defined C type, variable typename1 includes the type argument. Because valid field names cannot contain the $ character, list changes the $ to DOLLAR.

```
typename1 = '$fake3'; % name of defined C type with no tag
list(IDE_Obj,'type',typename1);
ans  =
```

```
       DOLLARfake0 : [typeinfo]

ans.DOLLARfake0 =

        type: 'struct $fake0'
        size: 1
      uclass: 'structure'
      sizeof: 1
     members: [1x1 struct]
```

When you request information about a project in CCS, you see a listing like the following that includes structures containing details about your project.

```
projectinfo = list(IDE_Obj,'project')

projectinfo =

          name: 'D:\Work\c6711dskafxr_c6000_rtw\c6711dskafxr.pjt'
          type: 'project'
    processortype: 'TMS320C67XX'
      srcfiles: [69x1 struct]
      buildcfg: [3x1 struct]
```

# See Also

`info`

**Introduced in R2011a**

# listsessions

List existing sessions

## Syntax

```
list = listsessions
list = listsessions('verbose')
```

## IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++

## Description

`list = listsessions` returns `list` that contains a listing of the sessions by name currently in the development environment.

`list = listsessions('verbose')` adds the optional input argument `verbose`. When you include the `verbose` argument, `listsessions` returns a cell array that contains one row for each existing session. Each row has three columns — processor type, platform name, and processor name.

## See Also

`adivdsp`

**Introduced in R2011a**

# load

Load program file onto processor

## Syntax

`load(IDE_Obj,filename,timeout)`

## IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

## Description

`load(IDE_Obj,filename,timeout)` loads the file specified by the *`filename`* argument to the processor.

The *`filename`* argument can include a full path to the file, or the name of a file in the IDE working folder.

With the VisualDSP++ and Code Composer Studio IDEs, you can use the cd method to check or modify the IDE working folder.

Only use `load` with program files created by the IDE build process.

The *`timeout`* argument defines the number of seconds MATLAB waits for the load process to complete. If the time-out period expires before the load process returns a completion message, MATLAB generates an error and returns. Usually the program load process works in spite of the error message.

If you omit the *`timeout`* argument, `load` uses the timeout property of the IDE handle object, which you can get by entering `get(IDE_Obj,'timeout')`.

## Examples

```
load(IDE_Obj,programfile)
run(id)
```

## See Also

cd | dir | open

**Introduced in R2011a**

# ExecutionTimeInSeconds

Get execution time in seconds for profiled section of code

## Syntax

`ExecutionTimes = NthSectionProfile.ExecutionTimeInSeconds`

## Description

`ExecutionTimes = NthSectionProfile.ExecutionTimeInSeconds` returns a vector of execution times, measured in seconds, for the profiled section of code. Each element of `ExecutionTimes` contains the difference between the timer reading at the start and the end of the section.

If you set the `CodeProfilingSaveOptions` parameter to `'SummaryOnly'`, `NthSectionProfile.ExecutionTimeInSeconds` returns an empty array. To change that parameter, open the Configuration Parameters dialog box by pressing **Ctrl+E**, open the **Verification** pane under **Code Generation**, and change the **Save options** parameter to `All data`.

## Examples

### Get Execution Times for Code Section

Run a simulation with a model that is configured to generate a workspace variable with execution time measurements.

```
rtwdemo_sil_topmodel;
set_param('rtwdemo_sil_topmodel', 'CodeExecutionProfiling', 'on');
set_param('rtwdemo_sil_topmodel', 'SimulationMode', 'software-in-the-loop (SIL)');
set_param('rtwdemo_sil_topmodel', 'CodeProfilingInstrumentation', 'on');
set_param('rtwdemo_sil_topmodel', 'CodeProfilingSaveOptions', 'AllData');
sim('rtwdemo_sil_topmodel');
```

The simulation generates the workspace variable `executionProfile` (default).

At the end of the simulation, get the profile for the seventh code section.

```
SeventhSectionProfile = executionProfile.Sections(7);
```

Get vector of execution times for the code section.

```
time_vector = SeventhSectionProfile.ExecutionTimeInSeconds;
```

## Input Arguments

**NthSectionProfile — coder.profile.ExecutionTimeSection**
object

Object generated by the `coder.profile.ExecutionTime` property `Sections`.

## Output Arguments

**ExecutionTimes — Execution time measurements**
double

Execution times, in seconds, for section of code. Returned as a vector.

## See Also

`ExecutionTimeInTicks` | `Sections`

### Topics

"Code Execution Profiling with SIL and PIL"
"Analyze Code Execution Data"

**Introduced in R2013a**

# ExecutionTimeInSeconds

Get execution time in seconds for profiled section of code

## Syntax

`ExecutionTimes = NthSectionProfile.ExecutionTimeInSeconds`

## Description

`ExecutionTimes = NthSectionProfile.ExecutionTimeInSeconds` returns a vector of execution times, measured in seconds, for the profiled section of code. Each element of `ExecutionTimes` contains the difference between the timer reading at the start and the end of the section.

## Examples

### Get Execution Times for Code Section

Copy MATLAB code to your working folder.

```
src_dir = ...
    fullfile(docroot,'toolbox','coder','examples','kalman');

copyfile(fullfile(src_dir,'kalman01.m'), '.')
copyfile(fullfile(src_dir,'test01_ui.m'), '.')
copyfile(fullfile(src_dir,'plot_trajectory.m'), '.')
copyfile(fullfile(src_dir,'position.mat'), '.')
```

Set up and run a SIL execution.

```
config = coder.config('lib');
config.GenerateReport = true;

config.VerificationMode = 'SIL';
config.CodeExecutionProfiling = true;
```

```
codegen('-config', config, '-args', {zeros(2,1)}, 'kalman01');
```

```
coder.runTest('test01_ui', ['kalman01_sil.' mexext]);
```

At end of the execution, you see the following message.

```
To terminate execution: clear kalman01_sil
Execution profiling report available after termination.
```

Click the link `clear kalman01_sil`.

```
### Stopping SIL execution for 'kalman01'
    Execution profiling report: report(getCoderExecutionProfile('kalman01'))
```

Create a workspace variable that holds execution time data.

```
executionProfile=getCoderExecutionProfile('kalman01');
```

Get the profile for the second code section.

```
SecondSectionProfile = executionProfile.Sections(2);
```

Get vector of execution times for the code section.

```
time_vector = SecondSectionProfile.ExecutionTimeInSeconds;
```

## Input Arguments

**`NthSectionProfile` — `coder.profile.ExecutionTimeSection`**
object

Object generated by the `coder.profile.ExecutionTime` property `Sections`.

## Output Arguments

**`ExecutionTimes` — Execution time measurements**
double

Execution times, in seconds, for section of code. Returned as a vector.

## See Also

`ExecutionTimeInTicks` | `Sections` | `getCoderExecutionProfile`

### Topics

"Generate Execution Time Profile"
"Analyze Execution Time Data"

**Introduced in R2013a**

# ExecutionTimeInTicks

Get execution times in timer ticks for profiled section of code

## Syntax

*`ExecutionTimes`* `=` *`NthSectionProfile`*`.ExecutionTimeInTicks`

## Description

*`ExecutionTimes`* `=` *`NthSectionProfile`*`.ExecutionTimeInTicks` returns a vector of execution times, measured in timer ticks, for the profiled section of code. Each element of *`ExecutionTimes`* contains the difference between the timer reading at the start and the end of the section. The data type of the arrays is the same as the data type of the timer used on the target, which allows you to infer the maximum range of the timer measurements.

*`NthSectionProfile`* is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

If you set the `CodeProfilingSaveOptions` parameter to `'SummaryOnly'`, *`NthSectionProfile`*`.ExecutionTimeInTicks` returns an empty array. To change that parameter, open the Configuration Parameters dialog box by pressing **Ctrl+E**, open the **Verification** pane under **Code Generation**, and change the **Save options** parameter to `All data`.

**Tip** You can calculate the execution time in seconds using the formula

$$ExecutionTimeInSecs = ExecutionTimeInTicks\,/\,TimerTicksPerSecond$$

## Output Arguments

### *`ExecutionTimes`*

Vector of execution times, in timer ticks, for profiled section of code

***`SelfExecutionTimes`***

Vector of execution times, in timer ticks, for profiled section of code but excluding time spent in child functions

## See Also

`MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `display` | `report`

### Topics

"Code Execution Profiling with SIL and PIL"
"View and Compare Code Execution Times"
"Analyze Code Execution Data"

**Introduced in R2012b**

# ExecutionTimeInTicks

Get execution times in timer ticks for profiled section of code

## Syntax

*ExecutionTimes* = *NthSectionProfile*.ExecutionTimeInTicks

## Description

*ExecutionTimes* = *NthSectionProfile*.ExecutionTimeInTicks returns a vector of execution times, measured in timer ticks, for the profiled section of code. Each element of *ExecutionTimes* contains the difference between the timer reading at the start and the end of the section. The data type of the arrays is the same as the data type of the timer used on the target, which allows you to infer the maximum range of the timer measurements.

*NthSectionProfile* is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

**Tip** You can calculate the execution time in seconds using the formula

$$ExecutionTimeInSecs = ExecutionTimeInTicks / TimerTicksPerSecond$$

Alternatively, setTimerTicksPerSecond and use ExecutionTimeInSeconds.

## Output Arguments

***ExecutionTimes***

Vector of execution times, in timer ticks, for profiled section of code

## See Also

ExecutionTimeInSeconds | MaximumExecutionTimeCallNum | MaximumExecutionTimeInTicks | MaximumSelfTimeCallNum |

`MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `getCoderExecutionProfile` | `report`

## Topics

"Generate Execution Time Profile"
"Analyze Execution Time Data"

**Introduced in R2012b**

# MaximumExecutionTimeCallNum

Get the call number at which maximum number of timer ticks occurred

## Syntax

`MaxTicksCallNum = NthSectionProfile.MaximumExecutionTimeCallNum`

## Description

`MaxTicksCallNum = NthSectionProfile.MaximumExecutionTimeCallNum` returns the call number at which the maximum number of timer ticks was recorded in a single invocation of the profiled code section during a simulation.

*`NthSectionProfile`* is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

## Output Arguments

### *MaxTicksCallNum*

Call number at which the maximum number of timer ticks occurred for a single invocation of the profiled code section

## See Also

`ExecutionTimeInTicks` | `ExecutionTimeInTicks` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `display` | `report`

## Topics

"Code Execution Profiling with SIL and PIL"
"View and Compare Code Execution Times"
"Analyze Code Execution Data"

**Introduced in R2012b**

# MaximumExecutionTimeCallNum

Get the call number at which maximum number of timer ticks occurred

## Syntax

`MaxTicksCallNum = NthSectionProfile.MaximumExecutionTimeCallNum`

## Description

`MaxTicksCallNum = NthSectionProfile.MaximumExecutionTimeCallNum` returns the call number at which the maximum number of timer ticks was recorded in a single invocation of the profiled code section during an execution.

*`NthSectionProfile`* is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

## Output Arguments

***`MaxTicksCallNum`***

Call number at which the maximum number of timer ticks occurred for a single invocation of the profiled code section

## See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `getCoderExecutionProfile` | `report`

## Topics

"Generate Execution Time Profile"
"Analyze Execution Time Data"

**Introduced in R2012b**

# MaximumExecutionTimeInTicks

Get maximum number of timer ticks for single invocation of profiled code section

## Syntax

*`MaxTicks`* `=` *`NthSectionProfile`*`.MaximumExecutionTimeInTicks`

## Description

*`MaxTicks`* `=` *`NthSectionProfile`*`.MaximumExecutionTimeInTicks` returns the maximum number of timer ticks recorded in a single invocation of the profiled code section during a simulation.

*`NthSectionProfile`* is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

## Output Arguments

***`MaxTicks`***

Maximum number of timer ticks for single invocation of profiled code section

## See Also

`ExecutionTimeInTicks` | `ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `display` | `report`

## Topics

“Code Execution Profiling with SIL and PIL”
“View and Compare Code Execution Times”
“Analyze Code Execution Data”

**Introduced in R2012b**

# MaximumExecutionTimeInTicks

Get maximum number of timer ticks for single invocation of profiled code section

## Syntax

*MaxTicks* = *NthSectionProfile*.MaximumExecutionTimeInTicks

## Description

*MaxTicks* = *NthSectionProfile*.MaximumExecutionTimeInTicks returns the maximum number of timer ticks recorded in a single invocation of the profiled code section during an execution.

*NthSectionProfile* is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

## Output Arguments

### ***MaxTicks***

Maximum number of timer ticks for single invocation of profiled code section

## See Also

ExecutionTimeInTicks | MaximumExecutionTimeCallNum | MaximumSelfTimeCallNum | MaximumSelfTimeInTicks | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeInTicks | MaximumTurnaroundTimeInTicks | Name | NumCalls | Number | Sections | SelfTimeInTicks | TimerTicksPerSecond | TotalExecutionTimeInTicks | TotalSelfTimeInTicks | TotalTurnaroundTimeInTicks | TotalTurnaroundTimeInTicks | TurnaroundTimeInTicks | TurnaroundTimeInTicks | getCoderExecutionProfile | report

## Topics

"Generate Execution Time Profile"
"Analyze Execution Time Data"

**Introduced in R2012b**

# TotalExecutionTimeInTicks

Get total number of timer ticks recorded for profiled code section

## Syntax

`*TotalTicks* = *NthSectionProfile*.TotalExecutionTimeInTicks`

## Description

*TotalTicks* = *NthSectionProfile*.TotalExecutionTimeInTicks returns the total number of timer ticks recorded for the profiled code section over the entire simulation.

*NthSectionProfile* is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

## Output Arguments

### *TotalTicks*

Total number of timer ticks for profiled code section

## See Also

ExecutionTimeInTicks | MaximumExecutionTimeCallNum | MaximumExecutionTimeInTicks | MaximumSelfTimeCallNum | MaximumSelfTimeInTicks | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeInTicks | Name | NumCalls | Number | Sections | SelfTimeInTicks | TimerTicksPerSecond | TotalSelfTimeInTicks | TotalTurnaroundTimeInTicks | TurnaroundTimeInTicks | display | report

### Topics

"Code Execution Profiling with SIL and PIL"

“View and Compare Code Execution Times”
“Analyze Code Execution Data”

**Introduced in R2012b**

# TotalExecutionTimeInTicks

Get total number of timer ticks recorded for profiled code section

## Syntax

`TotalTicks = NthSectionProfile.TotalExecutionTimeInTicks`

## Description

*TotalTicks* = *NthSectionProfile*.TotalExecutionTimeInTicks returns the total number of timer ticks recorded for the profiled code section over the entire execution.

*NthSectionProfile* is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

## Output Arguments

### *TotalTicks*

Total number of timer ticks for profiled code section

## See Also

ExecutionTimeInTicks | MaximumExecutionTimeCallNum | MaximumExecutionTimeInTicks | MaximumSelfTimeCallNum | MaximumSelfTimeInTicks | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeInTicks | Name | NumCalls | Number | Sections | SelfTimeInTicks | TimerTicksPerSecond | TotalSelfTimeInTicks | TotalTurnaroundTimeInTicks | TurnaroundTimeInTicks | getCoderExecutionProfile | report

## Topics

"Generate Execution Time Profile"
"Analyze Execution Time Data"

**Introduced in R2012b**

# SelfTimeInTicks

Get number of timer ticks recorded for profiled code section, excluding time spent in child functions

## Syntax

*SelfTicks* = *NthSectionProfile*.SelfTimeInTicks

## Description

*SelfTicks* = *NthSectionProfile*.SelfTimeInTicks returns the number of timer ticks recorded for the profiled code section. However, this number excludes the time spent in calls to child functions.

*NthSectionProfile* is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

## Output Arguments

### *SelfTicks*

Number of timer ticks for profiled code section, excluding periods in child functions

## See Also

ExecutionTimeInTicks | ExecutionTimeInTicks | MaximumExecutionTimeCallNum | MaximumExecutionTimeInTicks | MaximumSelfTimeCallNum | MaximumSelfTimeInTicks | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeInTicks | Name | NumCalls | Number | Sections | TimerTicksPerSecond | TotalExecutionTimeInTicks | TotalSelfTimeInTicks | TotalTurnaroundTimeInTicks | TurnaroundTimeInTicks | display | report

## Topics

"Code Execution Profiling with SIL and PIL"
"View and Compare Code Execution Times"
"Analyze Code Execution Data"

**Introduced in R2012b**

# SelfTimeInTicks

Get number of timer ticks recorded for profiled code section, excluding time spent in child functions

## Syntax

`SelfTicks = NthSectionProfile.SelfTimeInTicks`

## Description

*`SelfTicks`* = *`NthSectionProfile`*`.SelfTimeInTicks` returns the number of timer ticks recorded for the profiled code section. However, this number excludes the time spent in calls to child functions.

*`NthSectionProfile`* is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

## Output Arguments

### *SelfTicks*

Number of timer ticks for profiled code section, excluding periods in child functions

## See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `getCoderExecutionProfile` | `report`

## Topics

"Generate Execution Time Profile"
"Analyze Execution Time Data"

**Introduced in R2012b**

# MaximumSelfTimeCallNum

Get the call number at which the maximum number of timer ticks occurred, excluding time spent in child functions

## Syntax

*MaxSelfTicksCallNum* = *NthSectionProfile*.MaxSelfTimeCallNum

## Description

*MaxSelfTicksCallNum* = *NthSectionProfile*.MaxSelfTimeCallNum returns the call number at which the maximum number of self-time ticks occurred for the profiled code section.

*NthSectionProfile* is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

## Output Arguments

### *MaxSelfTicksCallNum*

Call number at which the maximum number of self-time ticks occurred for profiled code section

## See Also

ExecutionTimeInTicks | ExecutionTimeInTicks | MaximumExecutionTimeCallNum | MaximumExecutionTimeInTicks | MaximumSelfTimeInTicks | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeInTicks | Name | NumCalls | Number | Sections | SelfTimeInTicks | TimerTicksPerSecond | TotalExecutionTimeInTicks | TotalSelfTimeInTicks | TotalTurnaroundTimeInTicks | TurnaroundTimeInTicks | display | report

## Topics

“Code Execution Profiling with SIL and PIL”
“Code Execution Profiling with SIL and PIL”
“View and Compare Code Execution Times”

**Introduced in R2012b**

# MaximumSelfTimeCallNum

Get the call number at which the maximum number of timer ticks occurred, excluding time spent in child functions

## Syntax

*MaxSelfTicksCallNum* = *NthSectionProfile*.MaxSelfTimeCallNum

## Description

*MaxSelfTicksCallNum* = *NthSectionProfile*.MaxSelfTimeCallNum returns the call number at which the maximum number of self-time ticks occurred for the profiled code section.

*NthSectionProfile* is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

## Output Arguments

***MaxSelfTicksCallNum***

Call number at which the maximum number of self-time ticks occurred for profiled code section

## See Also

ExecutionTimeInTicks | MaximumExecutionTimeCallNum | MaximumExecutionTimeInTicks | MaximumSelfTimeInTicks | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeInTicks | Name | NumCalls | Number | Sections | SelfTimeInTicks | TimerTicksPerSecond | TotalExecutionTimeInTicks | TotalSelfTimeInTicks | TotalTurnaroundTimeInTicks | TurnaroundTimeInTicks | getCoderExecutionProfile | report

## Topics

“Generate Execution Time Profile”
“Analyze Execution Time Data”

**Introduced in R2012b**

# MaximumSelfTimeInTicks

Get the maximum number of timer ticks recorded for profiled code section, excluding time spent in child functions

## Syntax

`MaxSelfTicks = NthSectionProfile.MaximumSelfTimeInTicks`

## Description

*MaxSelfTicks* = *NthSectionProfile*.MaximumSelfTimeInTicks returns the maximum number of timer ticks recorded for the profiled code section. This number excludes the time spent in calls to child functions.

*NthSectionProfile* is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

## Output Arguments

### ***MaxSelfTicks***

Maximum number of timer ticks for profiled code section, excluding periods in child functions

## See Also

`ExecutionTimeInTicks` | `ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `display` | `report`

## Topics

"Code Execution Profiling with SIL and PIL"
"View and Compare Code Execution Times"
"Analyze Code Execution Data"

**Introduced in R2012b**

# MaximumSelfTimeInTicks

Get the maximum number of timer ticks recorded for profiled code section, excluding time spent in child functions

## Syntax

*MaxSelfTicks* = *NthSectionProfile*.MaximumSelfTimeInTicks

## Description

*MaxSelfTicks* = *NthSectionProfile*.MaximumSelfTimeInTicks returns the maximum number of timer ticks recorded for the profiled code section. This number excludes the time spent in calls to child functions.

*NthSectionProfile* is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

## Output Arguments

### *MaxSelfTicks*

Maximum number of timer ticks for profiled code section, excluding periods in child functions

## See Also

ExecutionTimeInTicks | MaximumExecutionTimeCallNum | MaximumExecutionTimeInTicks | MaximumSelfTimeCallNum | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeInTicks | Name | NumCalls | Number | Sections | SelfTimeInTicks | TimerTicksPerSecond | TotalExecutionTimeInTicks | TotalSelfTimeInTicks | TotalTurnaroundTimeInTicks | TurnaroundTimeInTicks | getCoderExecutionProfile | report

## Topics

"Generate Execution Time Profile"
"Analyze Execution Time Data"

**Introduced in R2012b**

# TotalSelfTimeInTicks

Get total number of timer ticks recorded for profiled code section, excluding time spent in child functions

## Syntax

*`TotalSelfTicks`* `=` *`NthSectionProfile`*`.TotalSelfTimeInTicks`

## Description

*`TotalSelfTicks`* `=` *`NthSectionProfile`*`.TotalSelfTimeInTicks` returns the total number of timer ticks recorded for the profiled code section over the entire simulation. However, this number excludes the time spent in calls to child functions.

*`NthSectionProfile`* is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

## Output Arguments

### *`TotalSelfTicks`*

Total number of timer ticks for profiled code section, excluding periods in child functions

## See Also

`ExecutionTimeInTicks` | `ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `display` | `report`

## Topics

"Code Execution Profiling with SIL and PIL"
"View and Compare Code Execution Times"
"Analyze Code Execution Data"

**Introduced in R2012b**

# TotalSelfTimeInTicks

Get total number of timer ticks recorded for profiled code section, excluding time spent in child functions

## Syntax

`TotalSelfTicks = NthSectionProfile.TotalSelfTimeInTicks`

## Description

*TotalSelfTicks* = *NthSectionProfile*.TotalSelfTimeInTicks returns the total number of timer ticks recorded for the profiled code section over the entire execution. However, this number excludes the time spent in calls to child functions.

*NthSectionProfile* is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

## Output Arguments

### ***TotalSelfTicks***

Total number of timer ticks for profiled code section, excluding periods in child functions

## See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `getCoderExecutionProfile` | `report`

## Topics

"Generate Execution Time Profile"
"Analyze Execution Time Data"

**Introduced in R2012b**

# MaximumTurnaroundTimeInTicks

Get maximum number of timer ticks between start and finish of a single invocation of profiled code section

## Syntax

*MaxTicks* = *NthSectionProfile*.MaximumTurnaroundTimeInTicks

## Description

*MaxTicks* = *NthSectionProfile*.MaximumTurnaroundTimeInTicks returns the maximum number of timer ticks recorded between the start and finish of a single invocation of the profiled code section during a simulation. Unless the code is pre-empted, this is the same as the maximum execution time.

*NthSectionProfile* is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

## Output Arguments

### *MaxTurnaroundTicks*

Maximum number of timer ticks between start and finish of a single invocation of profiled code section

## See Also

ExecutionTimeInTicks | MaximumExecutionTimeCallNum | MaximumExecutionTimeInTicks | MaximumSelfTimeCallNum | MaximumSelfTimeInTicks | MaximumTurnaroundTimeCallNum | Name | NumCalls | Number | Sections | SelfTimeInTicks | TimerTicksPerSecond | TotalExecutionTimeInTicks | TotalSelfTimeInTicks | TotalTurnaroundTimeInTicks | TurnaroundTimeInTicks | display | report

## Topics

"Code Execution Profiling with SIL and PIL"
"View and Compare Code Execution Times"
"Analyze Code Execution Data"

# MaximumTurnaroundTimeInTicks

Get maximum number of timer ticks between start and finish of a single invocation of profiled code section

## Syntax

*MaxTicks* = *NthSectionProfile*.MaximumTurnaroundTimeInTicks

## Description

*MaxTicks* = *NthSectionProfile*.MaximumTurnaroundTimeInTicks returns the maximum number of timer ticks recorded between the start and finish of a single invocation of the profiled code section during a execution. Unless the code is pre-empted, this is the same as the maximum execution time.

*NthSectionProfile* is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

## Output Arguments

### ***MaxTurnaroundTicks***

Maximum number of timer ticks between start and finish of a single invocation of profiled code section

## See Also

ExecutionTimeInTicks | MaximumExecutionTimeCallNum | MaximumExecutionTimeInTicks | MaximumSelfTimeCallNum | MaximumSelfTimeInTicks | MaximumTurnaroundTimeCallNum | Name | NumCalls | Number | Sections | SelfTimeInTicks | TimerTicksPerSecond | TotalExecutionTimeInTicks | TotalSelfTimeInTicks | TotalTurnaroundTimeInTicks | TurnaroundTimeInTicks | getCoderExecutionProfile | report

## Topics

“Generate Execution Time Profile”
“Analyze Execution Time Data”

**Introduced in R2012b**

# MaximumTurnaroundTimeCallNum

Get call number of the maximum number of timer ticks between start and finish of a single invocation of profiled code section

## Syntax

*MaxTurnaroundTicksCallNum* = *NthSectionProfile*.MaximumTurnaroundTimeCallNum

## Description

*MaxTurnaroundTicksCallNum* = *NthSectionProfile*.MaximumTurnaroundTimeCallNum returns the call number in which the maximum number of timer ticks was recorded between start and finish of a single invocation of the profiled code section during a simulation. Unless the code is pre-empted, this is the same as the maximum execution time.

*NthSectionProfile* is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

## Output Arguments

### *MaxTurnaroundTicksCallNum*

Call number of the maximum number of timer ticks between start and finish of a single invocation of profiled code section

## See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` |

`TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `display` | `report`

## Topics

“Code Execution Profiling with SIL and PIL”
“View and Compare Code Execution Times”
“Analyze Code Execution Data”

# MaximumTurnaroundTimeCallNum

Get call number for the code section invocation with the maximum number of timer ticks between the start and the finish

## Syntax

`MaxTurnaroundTicksCallNum = NthSectionProfile.MaximumTurnaroundTimeCallNum`

## Description

*`MaxTurnaroundTicksCallNum`* = *`NthSectionProfile`*`.MaximumTurnaroundTimeCallNum` returns the call number in which the maximum number of timer ticks is recorded between the start and the finish of an invocation of the profiled code section. Unless the code is pre-empted, this is the same as the maximum execution time.

*`NthSectionProfile`* is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

## Output Arguments

### *MaxTurnaroundTicksCallNum*

Call number for the profiled code section invocation with the maximum number of timer ticks between start and finish

## See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` |

`TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `getCoderExecutionProfile` | `report`

## Topics

"Generate Execution Time Profile"
"Analyze Execution Time Data"

**Introduced in R2012b**

# TotalTurnaroundTimeInTicks

Get total number of timer ticks between start and finish of the profiled code section over the entire simulation.

## Syntax

*`TotalTurnaroundTicks`* `=` *`NthSectionProfile`*`.TotalTurnaroundTimeInTicks`

## Description

*`TotalTurnaroundTicks`* `=` *`NthSectionProfile`*`.TotalTurnaroundTimeInTicks` returns the total number of timer ticks recorded between the start and finish of the profiled code section over the entire simulation. Unless the code is pre-empted, this is the same as the total execution time.

*`NthSectionProfile`* is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

## Output Arguments

### *`TotalTurnaroundTicks`*

Total number of timer ticks between start and finish of the profiled code section over the entire simulation

## See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TurnaroundTimeInTicks` | `display` | `report`

## Topics

"Code Execution Profiling with SIL and PIL"
"View and Compare Code Execution Times"
"Analyze Code Execution Data"

# TotalTurnaroundTimeInTicks

Get total number of timer ticks between start and finish of the profiled code section over the entire execution.

## Syntax

*totalTurnaroundTicks* = *NthSectionProfile*.TotalTurnaroundTimeInTicks

## Description

*totalTurnaroundTicks* = *NthSectionProfile*.TotalTurnaroundTimeInTicks returns the total number of timer ticks recorded between the start and finish of the profiled code section over the entire execution. Unless the code is pre-empted, this is the same as the total execution time.

*NthSectionProfile* is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

## Output Arguments

### *totalTurnaroundTicks*

Total number of timer ticks between start and finish of the profiled code section over the entire execution

## See Also

ExecutionTimeInTicks | MaximumExecutionTimeCallNum | MaximumExecutionTimeInTicks | MaximumSelfTimeCallNum | MaximumSelfTimeInTicks | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeInTicks | Name | NumCalls | Number | Sections | SelfTimeInTicks | TimerTicksPerSecond | TotalExecutionTimeInTicks | TotalSelfTimeInTicks | TurnaroundTimeInTicks | getCoderExecutionProfile | report

## Topics

“Generate Execution Time Profile”
“Analyze Execution Time Data”

**Introduced in R2012b**

# TurnaroundTimeInTicks

Get number of timer ticks between start and finish of the profiled code section

## Syntax

*TurnaroundTicks* = *NthSectionProfile*.TurnaroundTimeInTicks

## Description

*TurnaroundTicks* = *NthSectionProfile*.TurnaroundTimeInTicks returns the number of timer ticks recorded between the start and finish of the profiled code section. Unless the code is pre-empted, this is the same as the execution time.

*NthSectionProfile* is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

## Output Arguments

### ***TurnaroundTicks***

Number of timer ticks between start and finish of the profiled code section

## See Also

ExecutionTimeInTicks | MaximumExecutionTimeCallNum | MaximumExecutionTimeInTicks | MaximumSelfTimeCallNum | MaximumSelfTimeInTicks | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeInTicks | Name | NumCalls | Number | Sections | SelfTimeInTicks | TimerTicksPerSecond | TotalExecutionTimeInTicks | TotalSelfTimeInTicks | TotalTurnaroundTimeInTicks | display | report

### Topics

"Code Execution Profiling with SIL and PIL"

"View and Compare Code Execution Times"
"Analyze Code Execution Data"

# TurnaroundTimeInTicks

Get number of timer ticks between start and finish of the profiled code section

## Syntax

`*TurnaroundTicks* = *NthSectionProfile*.TurnaroundTimeInTicks`

## Description

*`TurnaroundTicks`* `=` *`NthSectionProfile`*`.TurnaroundTimeInTicks` returns the number of timer ticks recorded between the start and finish of the profiled code section. Unless the code is pre-empted, this is the same as the execution time.

*`NthSectionProfile`* is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

## Output Arguments

***`TurnaroundTicks`***

Number of timer ticks between start and finish of the profiled code section

## See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `getCoderExecutionProfile` | `report`

## Topics

"Generate Execution Time Profile"
"Analyze Execution Time Data"

**Introduced in R2012b**

# modifyInheritedParam

**Class:** rtw.codegenObjectives.Objective
**Package:** rtw.codegenObjectives

Modify inherited parameter values

## Syntax

`modifyInheritedParam(obj, paramName, value)`

## Description

`modifyInheritedParam(obj, paramName, value)` changes the value of an inherited parameter that the Code Generation Advisor verifies in **Check model configuration settings against code generation objectives**. Use this method when you create a new objective from an existing objective.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a code generation objective object previously created. |
| *paramName* | Parameter that you modify in the objective. |
| *value* | Value of the parameter. |

## Examples

Change the value of `DefaultParameterBehavior` to `Tunable` in the objective.

```
modifyInheritedParam(obj, 'DefaultParameterBehavior', 'Tunable');
```

## See Also

`get_param`

## Topics

"Create Custom Code Generation Objectives"

# msgcount

Number of messages in read-enabled channel queue

---

**Note** Support for `msgcount` on C5000 processors will be removed in a future version.

---

## Syntax

```
msgcount(rx,'channel')
```

## IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

## Description

`msgcount(rx,'channel')` returns the number of unread messages in the read-enabled queue specified by `channel` for the RTDX interface `rx`. You cannot use `msgcount` on channels configured for write access.

## Examples

If you have created and loaded a program to the processor, you can write data to the processor, then use `msgcount` to determine the number of messages in the read queue.

1. Create and load a program to the processor.
2. Write data to the processor from MATLAB software.

   ```
   indata = 1:100;
   writemsg(rtdx(IDE_Obj),'ichannel', int32(indata));
   ```

3. Use `msgcount` to determine the number of messages available in the queue.

```
num_of_msgs = msgcount(rtdx(IDE_Obj),'ichannel')
```

## See Also

`read` | `readmat` | `readmsg`

**Introduced in R2011a**

# new

Create project, library, or build configuration in IDE

## Syntax

```
new(IDE_Obj,'name','type')
```

## IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

## Description

`new(IDE_Obj,'name','type')` creates a project, library, or build configuration in the IDE.

The `name` argument specifies the name of the new project, library, or build configuration

The type argument specifies whether to create a project, library, or build configuration. The options are:

- `'project'` — Executable project. Sometimes this file is called a "DSP executable file".
- `'projlib'` — Library project.
- `'projext'` — External make project. Only the CCS IDE supports this option.
- `'buildcfg'` — Build configuration in the active project. Only the VisualDSP++ and CCS IDEs support this option.

When *type* is `'project'` or `'projlib'` , *name* can include the full path to the new file. You can use the path to differentiate two files with the same name. If you omit the path, the `new` method creates the file or project in the current IDE working folder.

If you omit the *type* argument, and the *name* argument does not include a file extension, *type* defaults to 'project'.

When *type* is 'buildcfg', use a unique name to differentiate the build configuration from other build configurations in the active project.

The new method does not support 'text' as a *type* argument.

## Examples

```
new(IDE_Obj,'my_project','project') #Create an IDE project, 'my_project.gpj'
new(IDE_Obj,'my_build_config','buildcfg') #Create a build configuration.
```

## See Also

activate | close

**Introduced in R2011a**

# open

Open project in IDE

## Syntax

```
open(IDE_Obj,filename,filetype,timeout)
open(IDE_Obj,myproject)
```

## IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

## Description

`open(IDE_Obj,filename,filetype,timeout)` opens a project in the IDE.

Use the *filename* argument to specify the file name, including the file name extension. If the *filename* does not include a file name extension, you can specify the file type using the *filetype* argument. If the file does not exist in the current project or folder path, MATLAB returns a warning and returns control to MATLAB.

For the optional *filetype* argument, you can specify the following types.

| | CCS IDE | VisualDSP++ IDE |
|---|---|---|
| `'project'` — Project files | Yes | Yes |
| `'ProjectGroup'` — Project group files | No | Yes |
| `'program'` — Target program file (executable) | No. Use `load` instead. | No |

If you omit the *filetype* argument, *filetype* defaults to 'project'.

The optional *timeout* argument determines the number of seconds MATLAB waits for the IDE to finish opening the file before returning an error. If you omit the *timeout* argument, the open method uses the timeout property of the IDE handle object (IDE_Obj) instead. The timeout error does not terminate the loading process on the IDE.

---

**Note** The open method does not support the 'text', 'program', or 'workspace' arguments.

---

## Examples

open(*IDE_Obj*,myproject) opens the myproject project in the IDE.

## See Also

cd | dir | load | new

**Introduced in R2011a**

# plot

**Class:** cgv.CGV
**Package:** cgv

Create plot for signal or multiple signals

## Syntax

```
[signal_names, signal_figures] = cgv.CGV.plot(data_set)
[signal_names, signal_figures] = cgv.CGV.plot(data_set,'Signals',
signal_list)
```

## Description

`[signal_names, signal_figures] = cgv.CGV.plot(data_set)` create a plot for each signal in the `data_set`.

`[signal_names, signal_figures] = cgv.CGV.plot(data_set,'Signals', signal_list)` create a plot for each signal in the value of `'Signals'` and return the names and figure handles for the given signal names.

## Input Arguments

**`data_set`**

Output data from a model. After running the model, use the `getOutputData` function to get the data. The `cgv.CGV.getOutputData` function returns a cell array of the output signal names.

**`'Signals', signal_list`**

Parameter/value argument pair specifying the signal or signals to plot. The value for this parameter can be an individual signal name, or a cell array of character vectors, where each character vector is a signal name in the `data_set`. Use `getSavedSignals` to view

the list of available signal names in the `data_set`. The syntax for an individual signal name is:

```
signal_list = {'log_data.subsystem_name.Data(:,1)'}
```

The syntax for a list of signal names is:

```
signal_list = {'log_data.block_name.Data(:,1)',...
               'log_data.block_name.Data(:,2)',...
               'log_data.block_name.Data(:,3)',...
               'log_data.block_name.Data(:,4)'};
```

If a component of your model contains a space or newline character, MATLAB adds parentheses and a single quote to the name of the component. For example, if a section of the signal has a space, `'block name'`, MATLAB displays the signal name as:

```
log_data.('block name').Data(:,1)
```

To use the signal name as input to a CGV function, `'block name'` must have two single quotes. For example:

```
signal_list = {'log_data.(''block name'').Data(:,1)'}
```

## Output Arguments

Depending on the data, one or more of the following parameters might be empty:

**`signal_names`**

Cell array of signal names

**`signal_figures`**

Array of figure handles for signals

## See Also

### Topics

“Verify Numerical Equivalence with CGV”

# profile

Generate real-time execution or stack profiling report

## Syntax

```
profile(IDE_Obj,type,action,timeout)
```

## IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

## Description

Use `profile(IDE_Obj,type,action,timeout)` to generate real-time execution or stack profiling report.

Create the *`IDE_Obj`* IDE handle object using a constructor function before you use the `profile` method.

The *`type`* argument determines the type of profile to generate. The following types are available for the IDEs specified.

| | CCS IDE | VisualDSP++ IDE |
|---|---|---|
| `'execution'` — Execution profiling | Yes | Yes |
| `'stack'`— Stack profiling | Yes | Yes |

To get a real-time task execution profile report in HTML and graphical plot forms, set the *`type`* argument to `'execution'` and omit the *`action`* argument, which defaults to `'report'`. For more information, see “Perform Execution-Time Profiling for IDE and Toolchain Targets”.

To prepare the stack memory on the processor for profiling, set the *type* argument to `'stack'`, and set the *action* argument to `'setup'`. This action writes a repetitive series of known values to the stack memory. For more information, see “Perform Stack Profiling with IDE and Toolchain Targets”.

After preparing the stack memory, to measure and report the percentage of stack usage, set the *type* argument to `'stack'`, and set the *action* argument to `'report'`.

If you omit the *action* argument, *action* defaults to `'report'`.

The optional *timeout* argument determines the number of seconds MATLAB waits for the IDE to finish profiling before returning an error. If you omit the *timeout* argument, the open method uses the timeout property of the IDE handle object (IDE_Obj) instead.

**Note** You can use real-time task execution profiling with hardware only. Simulators do not support the profiling feature.

## Examples

To use `profile` to assess how your program executes in real-time, complete the following tasks with a Simulink model:

1. Open the model configuration parameters (**Ctrl+ E**).
2. Select the Coder Target pane.
3. Under the Tool Chain Automation tab, enable **Profile real-time execution**.
4. Build your model.

   ```
   build(IDE_Obj)
   ```
5. Load your program to the processor.

   ```
   load(IDE_Obj,'c:\work\sumdiff.out')
   ```
6. For stack profiling, initialize the stack to a known state. (For execution profiling, skip this step.)

   ```
   profile(IDE_Obj,'stack','setup')
   ```

   With the **`setup`** input argument, `profile` writes a known pattern into the addresses that compose the stack. For C6000 processors, the pattern is A5. For C2000™ and C5000 processors, the pattern is A5A5 to account for the address size. As long as

your application does not write the same pattern to the system stack, `profile` can report the stack usage.

7. Run the program on the processor.

   ```
   run(IDE_Obj)
   ```

8. Stop the running program.

   ```
   halt(IDE_Obj)
   ```

9. To get the profiling reports enter one of the following commands:

   ```
   profile(IDE_Obj,'stack','report') #Get stack profiling report
   profile(IDE_Obj,'execution') #Get execution profiling report
   ```

The HTML report contains the sections described in the following table.

| Section Heading | Description |
|---|---|
| Worst case task turnaround times | Maximum task turnaround time for each task since model execution started. |
| Maximum number of concurrent overruns for each task | Maximum number of concurrent task overruns since model execution started. |
| Analysis of profiling data recorded over *nnn* seconds. | Profiling data was recorded over *nnn* seconds. The recorded data for task turnaround times and task execution times is presented in the table following this heading. |

Task turnaround time is the elapsed time between starting and finishing the task. If the task is not preempted, task turnaround time equals the task execution time.

Task execution time is the time between task start and finish when the task is actually running. It does not include time during which the task may have been preempted by another task.

**Note** Task execution time cannot be measured directly. Task profiling infers the execution time from the task start and finish times, and the intervening periods during which the task was preempted by another task.

The execution time calculations do not account for processor time consumed by the scheduler while switching tasks. In cases where preemption occurs, the reported task execution times overestimate the true task execution time.

Task overruns occur when a timer task does not complete before the same task is scheduled to run again. Depending on how you configure the real-time scheduler, a task overrun may be handled as a real-time failure. Alternatively, you might allow a small number of task overruns to accommodate cases where a task occasionally takes longer than normal to complete. If a task overrun occurs, and the same task is scheduled to run again before the first overrun has been cleared, concurrent task overruns are said to have occurred.

## See Also

`load` | `run`

**Introduced in R2011a**

# read

Read data from processor memory

## Syntax

```
mem = read(IDE_Obj,address)
mem = read(IDE_Obj,…,datatype)
mem = read(IDE_Obj,…,count)
mem = read(IDE_Obj,…,memorytype)
```

## IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

## Description

`mem = read(IDE_Obj,address)` returns a block of data values from the memory space of the processor referenced by `IDE_Obj`. The block to read begins from the DSP memory location given by the *address* argument. The data is read starting from *address* without regard to type-alignment boundaries in the processor. Conversely, the byte ordering defined by the data type is automatically applied.

The *address* argument is a decimal or hexadecimal representation of a memory address in the processor. The full memory address consist of two parts:

- The start address
- The memory type

You can define the memory type value can be explicitly using a numeric vector representation of the address.

Alternatively, the `IDE_Obj` object has a default memory type value that is applied if the memory type value is not explicitly incorporated in the passed address parameter. In DSP processors with only a single memory type, it is possible to specify addresses using the abbreviated (implied memory type) format by setting the `IDE_Obj` object memory type value to zero.

**Note** You cannot read data from processor memory while the processor is running.

Provide the *`address`* argument either as a numerical value that is a decimal representation of the DSP memory address, or as a character vector that `read` converts to the decimal representation of the start address. (Refer to function `hex2dec` in the *MATLAB Function Reference*. `read` uses `hex2dec` to convert the hexadecimal character vector to a decimal value).

The examples in the following table show how `read` uses the `address` parameter.

| address Parameter Value | Description |
| --- | --- |
| `131082` | Decimal address specification. The memory start address is 131082 and memory type is 0. This action is the same as specifying [131082 0]. |
| `[131082 1]` | Decimal address specification. The memory start address is 131082 and memory type is 1. |
| `'2000A'` | Hexadecimal address specification provided as a character vector entry. The memory start address is 131082 (converted to the decimal equivalent) and memory type is 0. |

It is possible to specify `address` as a cell array. You can use a combination of numbers and character vectors for the start address and memory type values. For example, the following are valid addresses from cell array `myaddress`:

```
myaddress1 myaddress1{1} = 131072; myadddress1{2} = 'Program(PM)
Memory';

myaddress2 myaddress2{1} = '20000'; myadddress2{2} = 'Program(PM)
Memory';

myaddress3 myaddress3{1} = 131072; myaddress3{2} = 0;
```

`mem = read(IDE_Obj,…,datatype)` where the input argument `datatype` defines the interpretation of the raw values read from DSP memory. Parameter `datatype` specifies the data format of the raw memory image. The data is read starting from `address` without regard to data type alignment boundaries in the processor. The byte ordering defined by the data type is automatically applied. This syntax supports the following MATLAB data types.

| MATLAB Data Type | Description |
|---|---|
| `double` | IEEE double-precision floating point value |
| `single` | IEEE single-precision floating point value |
| `uint8` | 8-bit unsigned binary integer value |
| `uint16` | 16-bit unsigned binary integer value |
| `uint32` | 32-bit unsigned binary integer value |
| `int8` | 8-bit signed two's complement integer value |
| `int16` | 16-bit signed two's complement integer value |
| `int32` | 32-bit signed two's complement integer value |

The `read` method does not coerce data type alignment. Some combinations of `address` and `datatype` will be difficult for the processor to use.

`mem = read(IDE_Obj,…,count)` adds the `count` input parameter that defines the dimensions of the returned data block `mem`. To read a block of multiple data values. Specify `count` to determine how many values to read from `address`. `count` can be a scalar value that causes `read` to return a column vector that has `count` values. You can perform multidimensional reads by passing a vector for `count`. The elements in the input vector of `count` define the dimensions of the returned data matrix. The memory is read in column-major order. `count` defines the dimensions of the returned data array `mem` as shown in the following table.

- `n` — Read `n` values into a column vector.
- `[m,n]` — Read `m`-by-`n` values into `m` by `n` matrix in column-major order.
- `[m,n,...]` — Read a multidimensional matrix `m`-by-`n`-by…of values into an `m`-by-`n`-by…array.

To read a block of multiple data values, specify the input argument `count` that determines how many values to read from `address`.

`mem = read(IDE_Obj,…,memorytype)` adds an optional input argument `memorytype`. Object `IDE_Obj` has a default memory type value 0 that `read` applies if the memory type value is not explicitly incorporated into the passed address parameter.

In processors with only a single memory type, it is possible to specify addresses using the implied memory type format by setting the `IDE_Objmemorytype` property value to zero.

## Examples

This example reads one 16-bit integer from memory on the processor.

```
mlvar = read(IDE_Obj,131072,'int16')
```

`131072` is the decimal address of the data to read.

You can read more than one value at a time. This `read` command returns 100 32-bit integers from the address 0x20000 and plots the result in MATLAB.

```
data = read(IDE_Obj,'20000','int32',100)
plot(double(data))
```

## See Also

`write`

**Introduced in R2011a**

# readmat

Matrix of data from RTDX channel

---

**Note** Support for `readmat` on C5000 processors will be removed in a future version.

---

## Syntax

```
data = readmat(rx,channelname,'datatype',siz,timeout)
data = readmat(rx,channelname,'datatype',siz)
```

## IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

## Description

`data = readmat(rx,channelname,'datatype',siz,timeout)` reads a matrix of data from an RTDX channel configured for read access. `datatype` defines the type of data to read, and `channelname` specifies the queue to read. `readmat` reads the desired data from the RTDX link specified by `rx`.

Before you read from a channel, open and enable the channel for read access.

Replace `channelname` with the character vector you specified when you opened the desired channel. `channelname` must identify a channel that you defined in the program loaded on the processor.

You cannot read data from a channel you have not opened and configured for read access. To determine which channels exist for the loaded program, use the RTDX tools provided in the IDE.

`data` contains a matrix whose dimensions are given by the input argument vector `siz`, where `siz` can be a vector of two or more elements. To operate, the number of elements in the output matrix `data` must be an integral number of channel messages.

When you omit the `timeout` input argument, `readmat` reads messages from the specified channel until the output matrix is full or the global `timeout` period specified in `rx` elapses.

**Caution** If the timeout period expires before the output data matrix is fully populated, you lose the messages read from the channel to that point.

MATLAB software supports reading five data types with `readmat`.

| datatype Value | Data Format |
| --- | --- |
| 'double' | Double-precision floating point values. 64 bits. |
| 'int16' | 16-bit signed integers |
| 'int32' | 32-bit signed integers |
| 'single' | Single-precision floating point values. 32 bits. |
| 'uint8' | Unsigned 8-bit integers |

`data = readmat(rx,channelname,'datatype',siz)` reads a matrix of data from an RTDX channel configured for read access. `datatype` defines the type of data to read, and `channelname` specifies the queue to read. `readmat` reads the desired data from the RTDX link specified by `rx`.

## Examples

In this data read and write example, you write data to the processor through the IDE. You can then read the data back in two ways — either through `read` or through `readmsg`.

To duplicate this example you need to have a program loaded on the processor. The channels listed in this example, `ichannel` and `ochannel`, must be defined in the loaded program. If the current program on the processor defines different channels, replace the listed channels with your current ones.

```
IDE_Obj = ticcs;
rx = rtdx(IDE_Obj);
```

```
open(rx,'ichannel','w');
enable(rx,'ichannel');
open(rx,'ochannel','r');
enable(rx,'ochannel');
indata = 1:25; % Set up some data.
write(IDE_Obj,0,indata,30);
outdata = read(IDE_Obj,0,'double',25,10)

outdata  =
  Columns 1 through 13
  1   2   3   4   5   6   7   8   9  10  11  12  13
  Columns 14 through 25
  14  15  16  17  18  19  20  21  22  23  24  25
```

Now use RTDX to read the data into a 5-by-5 array called `out_array`.

```
out_array = readmat('ochannel','double',[5 5])
```

# See Also

`readmsg` | `writemsg`

**Introduced in R2011a**

# readmsg

Read messages from specified RTDX channel

**Note** Support for `readmsg` on C5000 processors will be removed in a future version.

## Syntax

```
data = readmsg(rx,channelname,'datatype',siz,nummsgs,timeout)
data = readmsg(rx,channelname,'datatype',siz,nummsgs)
data = readmsg(rx,channelname,datatype,siz)
data = readmsg(rx,channelname,datatype,nummsgs)
data = readmsg(rx,channelname,datatype)
```

## IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

## Description

`data = readmsg(rx,channelname,'datatype',siz,nummsgs,timeout)` reads `nummsgs` from a channel associated with `rx`. `channelname` identifies the channel queue, which must be configured for read access. Each message is the same type, defined by `datatype`. `nummsgs` can be an integer that defines the number of messages to read from the specified queue, or `all` to read the messages present in the queue when you call the `readmsg` function.

Each read message becomes an output matrix in `data`, with dimensions specified by the elements in vector `siz`. For example, when `siz` is `[m n]`, reading 10 messages (`nummsgs` equal 10) creates 10 `m`-by-`n` matrices in `data`. Each output matrix in `data` must have the same number of elements (`m` x `n`) as the number of elements in each message.

You must specify the type of messages you are reading by including the `datatype` argument. `datatype` supports character vectors that define the type of data you are expecting, as shown in the following table.

| datatype Value | Specified Data Type |
|---|---|
| 'double' | Floating point data, 64-bits (double-precision). |
| 'int16' | Signed 16-bit integer data. |
| 'int32' | Signed 32-bit integers. |
| 'single' | Floating-point data, 32-bits (single-precision). |
| 'uint8' | Unsigned 8-bit integers. |

When you include the `timeout` input argument in the function, `readmsg` reads messages from the specified queue until it receives `nummsgs`, or until the period defined by `timeout` expires while `readmsg` waits for more messages to be available.

When the desired number of messages is not available in the queue, `readmsg` enters a wait loop and stays there until more messages become available or `timeout` seconds elapse. The `timeout` argument overrides the global timeout specified when you create `rx`.

`data = readmsg(rx,channelname,'datatype',siz,nummsgs)` reads `nummsgs` from a channel associated with `rx`. `channelname` identifies the channel queue, which must be configured for read access. Each message is the same type, defined by `datatype`. `nummsgs` can be an integer that defines the number of messages to read from the specified queue, or `all` to read the messages present in the queue when you call the `readmsg` function.

Each read message becomes an output matrix in `data`, with dimensions specified by the elements in vector `siz`. When `siz` is `[m n]`, reading 10 messages (`nummsgs` equal 10) creates 10 `n`-by-`m` matrices in `data`.

Each output matrix in `data` must have the same number of elements (`m` x `n`) as the number of elements in each message.

You must specify the type of messages you are reading by including the `datatype` argument. `datatype` supports six character vectors that define the type of data you are expecting.

`data = readmsg(rx,channelname,datatype,siz)` reads one data message because `nummsgs` defaults to one when you omit the input argument. `readmsgs` returns the message as a row vector in `data`.

`data = readmsg(rx,channelname,datatype,nummsgs)` reads the number of messages defined by `nummsgs`. `data` becomes a cell array of row matrices, `data = {msg1,msg2,...,msg(nummsgs)}`, because `siz` defaults to `[1,nummsgs]`; each returned message becomes one row matrix in the cell array.

Each row matrix contains one element for each data value in the current message `msg# = [element(1), element(2),...,element(l)]` where `l` is the number of data elements in message. In this syntax, the read messages can have different lengths, unlike the previous syntax options.

`data = readmsg(rx,channelname,datatype)` reads one data message, returning a row vector in `data`. The optional input arguments—`nummsgs`, `siz`, and `timeout`—use their default values.

In the calling syntaxes for `readmsg`, you can set `siz` and `nummsgs` to empty matrices, causing them to use their default values—`nummsgs` = 1 and `siz` = [1,l], where l is the number of data elements in the read message.

**Caution** If the timeout period expires before the output data matrix is fully populated, you lose the messages read from the channel to that point.

# Examples

```
IDE_Obj = ticcs;
rx = rtdx(IDE_Obj);
open(rx,'ichannel','w');
enable(rx,'ichannel');
open(rx,'ochannel','r');
enable(rx,'ochannel');
indata = 1:25; % Set up some data.
write(IDE_Obj,0,indata,30);
outdata = read(IDE_Obj,0,'double',25,10)

outdata  =
  Columns 1 through 13
   1   2   3   4   5   6   7   8   9  10  11  12  13
  Columns 14 through 25
  14  15   16  17  18  19  20  21  22  23  24  25
```

Now use RTDX to read the messages into a 4-by-5 array called `out_array`.

```
number_msgs = msgcount(rx,'ochannel') % Check number of msgs
                                      % in read queue.
out_array = rtdx(IDE_Obj).readmsg('ochannel','double',[4 5])
```

# See Also

read | readmat | writemsg

**Introduced in R2011a**

# register

**Class:** rtw.codegenObjectives.Objective
**Package:** rtw.codegenObjectives

Register objective

## Syntax

`register(obj)`

## Description

`register(obj)` registers *obj* Register and add *obj* to the end of the list of available objectives that you can use with the Code Generation Advisor.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a code generation objective object previously created. |

## Examples

Register the objective:

```
register(obj);
```

## See Also

### Topics

"Create Custom Code Generation Objectives"
"Registering Customizations" (Simulink)

# registerCFunctionEntry

Create function entry based on specified parameters and register in code replacement table

## Syntax

```
entry = registerCFunctionEntry(hTable,priority,numInputs,
functionName,inputType,implementationName,outputType,headerFile,
genCallback,genFileName)
```

## Description

`entry = registerCFunctionEntry(hTable,priority,numInputs, functionName,inputType,implementationName,outputType,headerFile, genCallback,genFileName)` provides a quick way to create and register a code replacement function entry.

This function can be used only if your function entry meets the following conditions:

- The input arguments are of the same type.
- The input argument names and the return argument name follow the default Simulink naming convention:
  - For input argument names, `u1`, `u2`, ..., `u`*n*
  - For return argument, `y1`

## Examples

### Create C Function Entry in Table

This example shows how to use the `registerCFunctionEntry` function to create a C function entry for `sqrt` in a code replacement table.

```
hLib = RTW.TflTable;

hLib.registerCFunctionEntry(100, 1, 'sqrt', 'double', 'sqrt', ...
                             'double', '<math.h>', '', '');
```

## Input Arguments

**`hTable` — Handle to a code replacement table**
handle

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

Example: `hLib`

**`priority` — Specifies the search priority of the function entry**
integer 0..100

The *`priority`* specifies the search priority of the function entry, relative to other entries of the same function name and conceptual argument list within this table. Highest priority is 0, and lowest priority is 100. If the table provides two implementations for a function, the implementation with the higher priority shadows the one with the lower priority.

Example: `100`

**`numInputs` — Specifies the number of input arguments**
positive integer

Example: `1`

**`functionName` — Specifies the name of the function to replace**
character vector

The *`functionName`* specifies the name of the function to replace. The name must match a function name listed in “Code You Can Replace” in “What Is Code Replacement Customization?” (MATLAB code) or “What Is Code Replacement Customization?” (Simulink models).

Example: `'sqrt'`

**`inputType` — Specifies the data type of the input arguments**
character vector

This function requires that the input arguments are of the same type.

Example: `'double'`

**`implementationName` — Specifies the name of the implementation**
character vector

The *`implementationName`* specifies the name of the implementation. For example, if *`functionName`* is `'sqrt'`, *`implementationName`* can be `'sqrt'` or a different name.

Example: `'sqrt'`

**`outputType` — Specifies the data type of the return argument**
character vector

Example: `'double'`

**`headerFile` — Specifies the header file that declares the implementation function**
character vector

Example: `'<math.h>'`

**`genCallback` — Specifies callback that follows code generation**
`''` | `'RTW.copyFileToBuildDir'`

The *`genCallback`* specifies a callback that follows code generation. If you specify `'RTW.copyFileToBuildDir'`, and if this function entry is matched and used, the code generator calls function `RTW.copyFileToBuildDir` after code generation. This callback function copies additional header, source, or object files that you have specified for this function entry to the build folder. For more information, see "Specify Build Information for Replacement Code".

Example: `''`

**`genFileName` — Specifies `''`**
`''` | character vector

This argument is reserved for MathWorks developers.

Example: `''`

## Output Arguments

**`entry` — Handle to the created code replacement function entry**
handle

The *`entry`* is a handle to the created code replacement function entry. Specifying the return argument in the `registerCFunctionEntry` function call is optional.

## See Also

`registerCPromotableMacroEntry`

### Topics

"Define Code Replacement Mappings"

**Introduced in R2007b**

# registerCPPFunctionEntry

Create C++ function entry based on specified parameters and register in code replacement table

## Syntax

## Description

provides a quick way to create and register a code replacement C++ function entry.

This function can be used only if your C++ function entry meets the following conditions:

- The input arguments are of the same type.
- The input argument names and the return argument name follow the default Simulink naming convention:
  - For input argument names, `u1`, `u2`, ..., `u`*n*
  - For return argument, `y1`

When you register a code replacement library containing C++ function entries, you must specify the value `{'C++'}` for the `LanguageConstraint` property of the library registry entry. For more information, see "Register Code Replacement Mappings".

## Examples

#### Create C++ Function Entry in Table

This example shows how to use the `registerCPPFunctionEntry` function to create a C ++ function entry for `sin` in a code replacement table.

```
hLib = RTW.TflTable;
```

```
hLib.registerCPPFunctionEntry(100, 1, 'sin', 'single', 'sin', ...
                              'single', 'cmath', '', '', 'std');
```

## Input Arguments

**`hTable` — Handle to a code replacement table**
handle

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

Example: `hLib`

**`priority` — Specifies the search priority for the function entry**
integer 0..100

The *`priority`* specifies the search priority for the function entry, relative to other entries of the same function name and conceptual argument list within this table. Highest priority is 0, and lowest priority is 100. If the table provides two implementations for a function, the implementation with the higher priority shadows the one with the lower priority.

Example: `100`

**`numInputs` — Specifies the number of input arguments**
positive integer

Example: `1`

**`functionName` — Specifies the name of the function to replace**
character vector

The *`functionName`* specifies the name of the function to replace. The name must match a function listed in "Code You Can Replace from MATLAB Code" or "Code You Can Replace From Simulink Models".

Example: `'sin'`

**`inputType` — Specifies the data type of the input arguments**
character vector

This function requires that the input arguments are of the same type.

Example: `'double'`

**`implementationName` — Specifies the name of the implementation**
character vector

The *`implementationName`* specifies the name of the implementation. For example, if *`functionName`* is `'sqrt'`, *`implementationName`* can be `'sqrt'` or a different name.

Example: `'sqrt'`

**`outputType` — Specifies the data type of the return argument**
character vector

Example: `'double'`

**`headerFile` — Specifies the header file that declares the implementation function**
character vector

Example: `'<math.h>'`

**`genCallback` — Specifies callback that follows code generation**
`''` | `'RTW.copyFileToBuildDir'`

The *`genCallback`* specifies a callback that follows code generation. If you specify `'RTW.copyFileToBuildDir'`, and if this function entry is matched and used, the code generator calls function `RTW.copyFileToBuildDir` after code generation. This callback function copies additional header, source, or object files that you have specified for this function entry to the build folder. For more information, see “Specify Build Information for Replacement Code”.

Example: `''`

**`genFileName` — Specifies `''`**
`''`

This argument is reserved for MathWorks developers.

Example: `''`

**`nameSpace` — Specifies the C++ namespace in which the implementation function is defined**
character vector

The *`nameSpace`* specifies the C++ namespace in which the implementation function is defined. If this function entry is matched, the software emits the namespace in the

generated function code (for example, `std::sin(tfl_cpp_U.In1)`). If you specify `''`, the software does not emit a namespace designation in the generated code.

Example: `'std'`

## Output Arguments

**`entry` — Handle to the created C++ function entry**
handle

The *`entry`* is a handle to the created C++ function entry. Specifying the return argument in the `registerCPPFunctionEntry` function call is optional.

## See Also

`enableCPP` | `setNameSpace`

### Topics

"Define Code Replacement Mappings"

**Introduced in R2010a**

# registerCPromotableMacroEntry

Create promotable code replacement macro entry based on specified parameters and register in code replacement table (for `abs` function replacement only)

## Syntax

```
entry = registerCPromotableMacroEntry(hTable,priority,numInputs,
functionName,inputType,implementationName,outputType,headerFile,
genCallback,genFileName)
```

## Description

`entry = registerCPromotableMacroEntry(hTable,priority,numInputs, functionName,inputType,implementationName,outputType,headerFile, genCallback,genFileName)` creates a promotable macro entry based on specified parameters and registers the entry in the code replacement table. A promotable macro entry promotes the output data type based on the target word size.

This function provides a quick way to create and register a promotable macro entry. This function can be used only if your code replacement function entry meets the following conditions:

- The input arguments are of the same type.
- The input argument names and the return argument name follow the default Simulink naming convention:
  - For input argument names, `u1`, `u2`, ..., `u`*n*
  - For return argument, `y1`

Use this function only for `abs` function replacement. For other functions supported for replacement, use `registerCFunctionEntry`.

## Examples

### Create Promotable Macro Entry in Table

This example shows how to use the `registerCPromotableMacroEntry` function to create a promotable macro entry for `abs` in a code replacement table.

```
hLib = RTW.TflTable;

hLib.registerCPromotableMacroEntry(100, 1, 'abs', ...
    'double', 'abs_prime', ...
    'double', '<math_prime.h>', '', '');
```

## Input Arguments

**`hTable` — Handle to a code replacement table**
handle

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

Example: `hLib`

**`priority` — Specifies the search priority for the function entry**
integer 0..100

The *`priority`* specifies the search priority for the function entry, relative to other entries of the same function name and conceptual argument list within this table. Highest priority is 0, and lowest priority is 100. If the table provides two implementations for a function, the implementation with the higher priority shadows the one with the lower priority.

Example: `100`

**`numInputs` — Specifies the number of input arguments**
positive integer

Example: `1`

**`functionName` — Specifies the name of the function to replace**
character vector

The *`functionName`* specifies the name of the function to be replaced. Specify `'abs'`. Use this function only for `abs` function replacement.

Example: `'abs'`

**inputType — Specifies the data type of the input arguments**
character vector

This function requires that the input arguments are of the same type.

Example: `'double'`

**implementationName — Specifies the name of the implementation**
character vector

The *implementationName* specifies the name of the implementation. For example, assuming *functionName* is `'abs'`, *implementationName* can be `'abs'` or a different name of your choosing.

Example: `'abs'`

**outputType — Specifies the data type of the return argument**
character vector

Example: `'double'`

**headerFile — Specifies the header file that declares the implementation function**
character vector

Example: `'<math.h>'`

**genCallback — Specifies callback that follows code generation**
`''` | `'RTW.copyFileToBuildDir'`

The *genCallback* specifies a callback that follows code generation. If you specify `'RTW.copyFileToBuildDir'`, and if this function entry is matched and used, the code generator calls function `RTW.copyFileToBuildDir` after code generation. This callback function copies additional header, source, or object files that you have specified for this function entry to the build folder. For more information, see “Specify Build Information for Replacement Code”.

Example: `''`

**genFileName — Specifies ''**
`''` | character vector

This argument is reserved for MathWorks developers.

Example: `''`

## Output Arguments

**`entry` — Handle to the created promotable macro entry**
handle

The *`entry`* is a handle to the created promotable macro entry. Specifying the return argument in the `registerCPromotableMacroEntry` function call is optional.

## See Also

`registerCFunctionEntry`

### Topics

"Define Code Replacement Mappings"

**Introduced in R2007b**

# regread

Values from processor registers

## Syntax

```
reg = regread(IDE_Obj,'regname','represent',timeout)
reg = regread(IDE_Obj,'regname','represent')
reg = regread(IDE_Obj,'regname')
```

## IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

## Description

`reg = regread(IDE_Obj,'regname','represent',timeout)` reads the data value in the `regname` register of the target processor and returns the value in `reg` as a double-precision value. For convenience, `regread` converts each return value to the MATLAB `double` datatype. Making this conversion lets you manipulate the data in MATLAB. Character vector `regname` specifies the name of the source register on the target. The IDE handle, `IDE_Obj`, defines the target to read from. Valid entries for `regname` depend on your target processor.

**Note** `regread` does not read 64-bit registers, like the `cycle` register on Blackfin processors.

Register names are not case-sensitive — `a0` is the same as `A0`.

For example, MPC5500 processors provide the following register names that are valid entries for `regname`.

| Register Names | Register Contents |
| --- | --- |
| 'acc' | Accumulator A register |
| sprg0 through sprg7 | SPR registers |

For example, TMS320C6xxx processors provide the following register names that are valid entries for regname.

| Register Names | Register Contents |
| --- | --- |
| A0, A1, A2,..., A15 | General purpose A registers |
| B0, B1, B2,..., B15 | General purpose B registers |
| PC, ISTP, IFR, IRP, NRP, AMR, CSR | Other general purpose 32-bit registers |
| A1:A0, A2:A1,..., B15:B14 | 64-bit general purpose register pairs |

**Note** Use read (called a direct memory read) to read memory-mapped registers.

The represent input argument defines the format of the data stored in regname. Input argument represent takes one of three input character vectors.

| represent Value | Description |
| --- | --- |
| '2scomp' | Source register contains a signed integer value in two's complement format. This is the default setting when you omit the represent argument. |
| 'binary' | Source register contains an unsigned binary integer. |
| 'ieee' | Source register contains a floating point 32-bit or 64-bit value in IEEE floating-point format. Use this only when you are reading from 32 and 64 bit registers on the target. |

To limit the time that regread spends transferring data from the target processor, the optional argument timeout tells the data transfer process to stop after timeout seconds. timeout is defined as the number of seconds allowed to complete the read operation. You might find this useful for limiting prolonged data transfer operations. If you omit the *timeout* argument, regread defaults to the global time-out defined in IDE_Obj.

`reg = regread(IDE_Obj,'regname','represent')` does not set the global time-out value. The time-out value in `IDE_Obj` applies.

`reg = regread(IDE_Obj,'regname')` does not define the format of the data in `regname`.

### Reading and Writing Register Values

Register variables can be difficult to read and write because the registers which hold their value are not dedicated to storing just the variable values.

Registers are used as temporary storage locations during execution. When this temporary storage process occurs, the value of the variable is temporarily stored somewhere on the stack and returned later. Therefore, getting the values of register variables during program execution may return unexpected answers.

Values that you write to register variables and local variables during intermediate times in program operation may not get reflected in the register.

To see if the result is consistent, write a line of code that uses the variable. For example:

```
register int a = 100;
int b;
 ...
b = a + 2;
```

Reading the register assigned to `a` may return an incorrect value for `a` but if `b` returns the expected 102 result, nothing is wrong with the code or the software.

## Examples

### For CCS IDE

For the C5xxx processor family, most registers are memory-mapped and consequently are available using `read` and `write`. However, use `regread` to read the PC register. The following command shows how to read the PC register. To identify the processor, `IDE_Obj` is a link for CCS IDE.

```
regread(IDE_Obj,'PC','binary')
```

To tell MATLAB software what datatype you are reading, the character vector `'binary'` indicates that the `PC` register contains a value stored as an unsigned binary integer.

In response, MATLAB software displays

```
ans  =

       33824
```

For processors in the C6xxx family, `regread` lets you access processor registers directly. To read the value in general purpose register A0, type the following function.

```
treg = regread(IDE_Obj,'A0','2scomp');
```

`treg` now contains the two's complement representation of the value in `A0`.

Now read the value stored in register B2 as an unsigned binary integer, by typing

```
regread(IDE_Obj,'B2','binary');
```

## See Also

`read` | `regwrite` | `write`

**Introduced in R2011a**

# regwrite

Write data values to registers on processor

## Syntax

```
regwrite(IDE_Obj,'regname',value,'represent',timeout)
regwrite(IDE_Obj,'regname',value,'represent')
regwrite(IDE_Obj,'regname',value,)
```

## IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

## Description

`regwrite(IDE_Obj,'regname',value,'represent',timeout)` writes the data in `value` to the `regname` register of the target processor. `regwrite` converts `value` from its representation in the MATLAB workspace to the representation specified by `represent`. The `represent` input argument defines the format of the data when it is stored in `regname`. Input argument `represent` takes one of three input character vectors.

| represent Value | Description |
|---|---|
| '2scomp' | Write `value` to the destination register as a signed integer value in two's complement format. This is the default setting when you omit the `represent` argument. |
| 'binary' | Write `value` to the destination register as an unsigned binary integer. |

| represent Value | Description |
| --- | --- |
| 'ieee' | Write value to the destination registers as a floating point 32-bit or 64-bit value in IEEE floating-point format. Use this only when you are writing to 32- and 64-bit registers on the target. |

**Note** Use write to write memory-mapped registers. This action is also called a *direct memory write*.

Character vector regname specifies the name of the destination register on the target. IDE handle, IDE_Obj defines the target to write value to. Valid entries for regname depend on your target processor. Register names are not case-sensitive — a0 is the same as A0.

For example, MPC5500 processors provide the following register names that are valid entries for regname.

| Register Names | Register Contents |
| --- | --- |
| 'acc' | Accumulator A register |
| sprg0 | SPR registers |

For example, C6xxx processors provide the following register names that are valid entries for regname.

| Register Names | Register Contents |
| --- | --- |
| A0, A1, A2,..., A15 | General purpose A registers |
| B0, B1, B2,..., B15 | General purpose B registers |
| PC, ISTP, IFR, IRP, NRP, AMR, CSR | Other general purpose 32-bit registers |
| A1:A0, A2:A1,..., B15:B14 | 64-bit general purpose register pairs |

Other processors provide other register sets. Refer to the documentation for your target processor to determine the registers for the processor.

To limit the time that regwrite spends transferring data to the target processor, the optional argument timeout tells the data transfer process to stop after timeout

seconds. `timeout` is defined as the number of seconds allowed to complete the write operation. You might find this useful for limiting prolonged data transfer operations.

If you omit the `timeout` input argument in the syntax, `regwrite` defaults to the global time-out defined in `IDE_Obj`. If the write operation exceeds the time specified, `regwrite` returns with a time-out error. Generally, time-out errors do not stop the register write process. The write process stops while waiting for the IDE to respond that the write operation is complete.

`regwrite(`*`IDE_Obj`*`,'regname',value,'represent')` omits the `timeout` input argument and does not change the time-out value specified in `IDE_Obj`.

`regwrite(`*`IDE_Obj`*`,'regname',value,)` omits the `represent` input argument. Writing the data does not reformat the data written to `regname`.

### Reading and Writing Register Values

Register variables can be difficult to read and write because the registers which hold their value are not dedicated to storing just the variable values.

Registers are used as temporary storage locations during execution. When this temporary storage process occurs, the value of the variable is temporarily stored somewhere on the stack and returned later. Therefore, getting the values of register variables during program execution may return unexpected answers.

Values that you write to register variables and local variables during intermediate times in program operation may not get reflected in the register.

To see if the result is consistent, write a line of code that uses the variable. For example:

```
register int a = 100;
int b;
 ...
b = a + 2;
```

Reading the register assigned to `a` may return an incorrect value for `a` but if `b` returns the expected 102 result, nothing is wrong with the code or the software.

## Examples

To write a new value to the PC register on a C5xxx family processor, enter

```
regwrite(IDE_Obj,'pc',hex2dec('100'),'binary')
```

specifying that you are writing the value 256 (the decimal value of 0x100) to register `pc` as binary data.

To write a 64-bit value to a register pair, such as B1:B0, the following syntax specifies the value as a character vector, representation, and target registers.

```
regwrite(IDE_Obj,'b1:b0',hex2dec('1010'),'ieee')
```

Registers `B1:B0` now contain the value 4112 in double-precision format.

## See Also

`read` | `regread` | `write`

**Introduced in R2011a**

# reload

Reload most recent program file to processor signal processor

## Syntax

```
s = reload(IDE_Obj,timeout)
s = reload(IDE_Obj)
```

## IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

## Description

`s = reload(IDE_Obj,timeout)` resends the most recently loaded program file to the processor. If you have not loaded a program file in the current session (so there is no previously loaded file), `reload` returns the null entry `[]` in `s` indicating that it could not load a file to the processor. Otherwise, `s` contains the full path name to the program file. After you reset your processor or after an event produces changes in your processor memory, use `reload` to restore the program file to the processor for execution.

To limit the time the IDE spends trying to reload the program file to the processor, *`timeout`* specifies how long the load process can take. If the load process exceeds the timeout limit, the IDE stops trying to load the program file and returns an error stating that the time period expired. Exceeding the allotted time for the reload operation usually indicates that the reload was complete but the IDE did not receive confirmation before the timeout period passed.

`s = reload(IDE_Obj)` reloads the most recent program file, using the *`timeout`* value set when you created link `IDE_Obj`, the global timeout setting.

### Using reload with Multiprocessor Boards

When your board contains more than one processor, `reload` calls the reloading function for each processor represented by `IDE_Obj`, reloading the most recently loaded program on each processor.

This action is the same as calling `reload` for each processor individually through IDE handle objects for each one.

## Examples

After you create an object that connects to the IDE, use the available methods to reload your most recently loaded project. If you have not loaded a project in this session, `reload` returns an error and an empty value for `s`. Loading a project eliminates the error. First, create an IDE handle object, such as IDE_Obj, using the constructor for your IDE.

```
s = reload(IDE_Obj,23)
Warning: No action taken - load a valid Program file before
you reload...

s  =

     ''

open((IDE_Obj,'D:\ti\tutorial\sim62xx\gelsolid\hellodsp.pjt','project')

build(IDE_Obj)

load(IDE_Obj,'hellodsp.pjt') #This file extension varies by IDE
halt(IDE_Obj)
s = reload(IDE_Obj,23)

s  =

D:\ti\tutorial\sim62xx\gelsolid\Debug\hellodsp.out
```

## See Also

`cd` | `load` | `open`

**Introduced in R2011a**

# remove

Remove file, project, or breakpoint

## Syntax

```
remove(IDE_Obj,filename,filetype)
remove(IDE_Obj,addr,debugtype,timeout)
remove(IDE_Obj,filename,line,debugtype,timeout)
remove(IDE_Obj,all,break)
```

## IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

## Description

`remove(`*`IDE_Obj`*`,`*`filename`*`,`*`filetype`*`)` deletes a file from the active project in the IDE or deletes the project.

`remove(`*`IDE_Obj`*`,`*`addr`*`,`*`debugtype`*`,`*`timeout`*`)` removes a debug point from an address in the program.

`remove(`*`IDE_Obj`*`,`*`filename`*`,`*`line`*`,`*`debugtype`*`,`*`timeout`*`)` removes a debug point from a line in a source file.

`remove(`*`IDE_Obj`*`,`*`all`*`,`*`break`*`)` removes the breakpoints and waits for completion.

## Input Arguments

**`IDE_Obj`**

Enter the name of the IDE link handle for your IDE. Create an IDE link handle before you use the `remove` method. .

**`filename`**

Replace *`filename`* with the name of the file you are removing, or the source file from which you are removing debug points. If the file is not located in the active project, MATLAB returns a warning instead of completing the action.

**`filetype`**

To remove a project, enter `'project'`. To remove a source file, enter `'text'`.

**Default:** `'text'`

**`addr`**

Enter the memory address of the debug point. Enter `'all'` to remove the breakpoints.

**`debugtype`**

Enter the type of debug point to remove. The IDE provide several types of debug points. Refer to the IDE help documentation for information on their respective behavior.

**Default:** `'break'` (breakpoint)

**`line`**

Enter the line number of the debug point located in a file.

**`timeout`**

Enter a time limit, in seconds, for the method to complete an action.

## Examples

After you have a project in the IDE, you can delete files from it using `remove` from the MATLAB software command line. For example, build a project and load the resulting `.out` file. With the project build complete, load your `.out` file by typing

```
load(IDE_Obj,'filename.out')
```

Now remove one file from your project

```
remove(IDE_Obj,'filename')
```

You see in the IDE that the file no longer appears.

## See Also

add | cd | open

**Introduced in R2011a**

# removeInheritedCheck

**Class:** rtw.codegenObjectives.Objective
**Package:** rtw.codegenObjectives

Remove inherited checks

## Syntax

`removeInheritedCheck(obj, checkID)`

## Description

`removeInheritedCheck(obj, checkID)` removes an inherited check from the objective definition. Use this method when you create a new objective from an existing objective.

When the user selects multiple objectives, if another selected objective includes this check, the Code Generation Advisor displays the check.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a code generation objective object previously created. |
| *checkID* | Unique identifier of the check that you remove from the new objective. |

## Examples

Remove the **Identify questionable code instrumentation (data I/O)** check from the objective.

```
removeInheritedCheck(obj, 'mathworks.codegen.CodeInstrumentation');
```

)

## See Also

`Simulink.ModelAdvisor`

### Topics

"Create Custom Code Generation Objectives"
"About IDs" (Simulink)

# removeInheritedParam

**Class:** rtw.codegenObjectives.Objective
**Package:** rtw.codegenObjectives

Remove inherited parameters

## Syntax

`removeInheritedParam(*obj*, *paramName*)`

## Description

`removeInheritedParam(*obj*, *paramName*)` removes an inherited parameter from this objective. Use this method when you create a new objective from an existing objective.

When the user selects multiple objectives, if another objective includes the parameter, the Code Generation Advisor reviews the parameter value using **Check model configuration settings against code generation objectives**.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a code generation objective object previously created. |
| *paramName* | Parameter that you want to remove from the objective. |

## Examples

Remove `DefaultParameterBehavior` from the objective.

```
removeInheritedParam(obj, 'DefaultParameterBehavior');
```

## See Also

`get_param`

## Topics

“Create Custom Code Generation Objectives”

# report

Open code execution profiling report and specify display of time measurements.

## Syntax

```
report(myExecutionProfile)
report(myExecutionProfile, Name1, Value1, Name2, Value2, ...)
report(myExecutionProfile, 'Units', 'Seconds', 'ScaleFactor',
'1e-06', 'NumericFormat', '%0.3f')
```

## Description

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

`report(myExecutionProfile)` opens the code execution profiling report using default display options.

`report(myExecutionProfile, Name1, Value1, Name2, Value2, ...)` opens the report with display options specified by the name-value *character vector* pairs.

`report(myExecutionProfile, 'Units', 'Seconds', 'ScaleFactor', '1e-06', 'NumericFormat', '%0.3f')` displays time in microseconds ($10^{-6}$ seconds) with a precision of three decimal places.

| Name-Value Pair | Details |
| --- | --- |
| `'Units'`, `'Seconds'` or `'Units'`, `'Ticks'` | Time measurements displayed in seconds or timer ticks.<br><br>Default:<br><br>• SIL simulation on Windows — Seconds<br>• SIL simulation on non-Windows — Timer ticks<br>• PIL simulation — Seconds, if number of timer ticks per second has been specified by the target connectivity configuration. Otherwise, ticks. |
| `'ScaleFactor'`, *`Value`* | Scale factor for displayed measurements. For example, to display measurements in microseconds, use the name-value pair `'ScaleFactor'`, `'1e-6'`.<br><br>*`Value`* must be a character vector representation of a number that is a power of 10. For example, `'1'`, `'1e-6'`, or `'1e-9'`. Default value is `'1e-9'`.<br><br>To specify the scale factor, you must also specify `'Units'`, `'Seconds'`. |
| `'NumericFormat'`, *`Convention`* | Numeric format for displayed measurements. Use the *decimal* convention utilized by the ANSI® C function `sprintf`, for example, `'%1.2f'`. Default is `'%0.0f'`.<br><br>To specify the numeric format, you must also specify `'Units'`, `'Seconds'`. |

# See Also

`annotate` | `display`

## Topics

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

**Introduced in R2011b**

# report

Open code execution profiling report and specify display of time measurements.

## Syntax

```
report(myExecutionProfile)
report(myExecutionProfile, Name1, Value1, Name2, Value2, ...)
report(myExecutionProfile, 'Units', 'Seconds', 'ScaleFactor',
'1e-06', 'NumericFormat', '%0.3f')
```

## Description

`report(myExecutionProfile)` opens the code execution profiling report using default display options.

`report(myExecutionProfile, Name1, Value1, Name2, Value2, ...)` opens the report with display options specified by the name-value *character vector* pairs.

`report(myExecutionProfile, 'Units', 'Seconds', 'ScaleFactor', '1e-06', 'NumericFormat', '%0.3f')` displays time in microseconds ($10^{-6}$ seconds) with a precision of three decimal places.

*`myExecutionProfile`* is a workspace variable that you create using `getCoderExecutionProfile`.

| Name-Value Pair | Details |
|---|---|
| 'Units','Seconds' or 'Units','Ticks' | Time measurements displayed in seconds or timer ticks.<br><br>Default:<br><br>• SIL execution on Windows — Seconds<br>• SIL execution on non-Windows — Timer ticks<br>• PIL execution — Seconds, if number of timer ticks per second has been specified by the target connectivity configuration. Otherwise, ticks. |
| 'ScaleFactor',*Value* | Scale factor for displayed measurements. For example, to display measurements in microseconds, use the name-value pair 'ScaleFactor','1e-6'.<br><br>*Value* must be a character vector representation of a number that is a power of 10. For example, '1', '1e-6', or '1e-9'. Default value is '1e-9'.<br><br>To specify the scale factor, you must also specify 'Units','Seconds'. |
| 'NumericFormat',*Convention* | Numeric format for displayed measurements. Use the *decimal* convention utilized by the ANSI C function sprintf, for example, '%1.2f'. Default is '%0.0f'.<br><br>To specify the numeric format, you must also specify 'Units','Seconds'. |

# See Also

Sections | TimerTicksPerSecond | getCoderExecutionProfile

## Topics

"Generate Execution Time Profile"
"Analyze Execution Time Data"

**Introduced in R2011b**

# reset

Stop program execution and reset processor

## Syntax

`reset(IDE_Obj,timeout)`

## IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

## Description

`reset(IDE_Obj,timeout)` stops the program executing on the processor and asynchronously performs a processor reset, returning the processor register contents to their power-up settings. `reset` returns immediately after the processor halt.

The optional *`timeout`* argument sets the number of seconds MATLAB waits for the processor to halt. If you omit the timeout argument, timeout defaults to the timeout value of the IDE handle object.

## See Also

`halt` | `load` | `run`

**Introduced in R2011a**

# restart

Reload most recent program file to processor signal processor

## Syntax

```
restart(IDE_Obj)
restart(IDE_Obj,timeout)
```

## IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

## Description

`restart(IDE_Obj)` issues a restart command in the IDE debugger. The behavior of the restart process depends on the processor. Refer to the documentation for your IDE for details about using restart with various processors.

When `IDE_Obj` is an array that contains more than one processor, each processor calls `restart` in sequence.

`restart(IDE_Obj,timeout)` adds the optional `timeout` input argument. `timeout` defines an upper limit in seconds on the period the restart routine waits for completion of the restart process. If the time-out period is exceeded, `restart` returns control to MATLAB with a time-out error. In general, `restart` causes the processor to initiate a restart, even if the time-out period expires. The time-out error indicates that the restart confirmation was not received before the time-out period elapsed.

## See Also

`halt` | `isrunning` | `run`

**Introduced in R2011a**

# rtIOStreamClose

Shut down communications channel

## Syntax

`errFlg = rtIOStreamClose(streamID)`

## Description

`errFlg = rtIOStreamClose(streamID)` shuts down the communications channel and cleans up associated resources.

## Examples

### Close Communications Channel

This code from `rtiostreamtest.c` detects errors when closing the communications channel.

```
static int closeServer(void)
{
    const int errorOccurred = rtIOStreamClose(streamID);
    if (errorOccurred == RTIOSTREAM_ERROR)
    {
        return errorOccurred;
    }
    return RTIOSTREAM_NO_ERROR;
}
```

## Input Arguments

**`streamID` — Stream handle**
scalar integer

Handle to the stream returned by a previous call to `rtIOStreamOpen`.

## Output Arguments

**`errFlg` — Error flag**
0 | -1

If the function runs without errors, it returns zero. Otherwise, it returns -1.

The `rtiostream.h` file defines these macros:

```
#define RTIOSTREAM_ERROR (-1)
#define RTIOSTREAM_NO_ERROR (0)
```

## See Also
`rtIOStreamOpen` | `rtIOStreamSend` | `rtIOStreamRecv` | `rtiostream_wrapper`

### Topics
"Create PIL Target Connectivity Configuration for MATLAB"
"Create PIL Target Connectivity Configuration for Simulink"
"Create a Target Communication Channel for Processor-In-The-Loop (PIL) Simulation"
"Configure Processor-In-The-Loop (PIL) for a Custom Target"

**Introduced in R2009a**

# rtIOStreamOpen

Initialize communications channel

## Syntax

`streamID = rtIOStreamOpen(argCount, argValues)`

## Description

`streamID = rtIOStreamOpen(argCount, argValues)` initializes a communication stream to allow the exchange of data between the development computer and target processor.

## Examples

### Initialize Communications Channel

This code from `rtiostreamtest.c` initializes a communication stream and checks for errors.

```
static int openServer(int rtArgc, void * rtArgv [])
{
    streamID = rtIOStreamOpen(rtArgc, rtArgv);
    if (streamID == RTIOSTREAM_ERROR)
    {
        return streamID;
    }
    return RTIOSTREAM_NO_ERROR;
}
```

## Input Arguments

**`argCount` — Argument count**
scalar integer

Number of elements in `argValues` array.

**`argValues` — Driver parameters**
array of pointers to character vectors

Parameters for the communications driver.

## Output Arguments

**`streamID` — Stream handle**
positive integer | -1

If the function initializes a communication stream, it returns a positive integer that represents the stream handle. Otherwise, it returns -1, which indicates an error.

The `rtiostream.h` file defines this macro:

```
#define RTIOSTREAM_ERROR (-1)
```

## See Also

`rtIOStreamSend` | `rtIOStreamRecv` | `rtIOStreamClose` | `rtiostream_wrapper`

### Topics

"Create PIL Target Connectivity Configuration for MATLAB"
"Create PIL Target Connectivity Configuration for Simulink"
"Create a Target Communication Channel for Processor-In-The-Loop (PIL) Simulation"
"Configure Processor-In-The-Loop (PIL) for a Custom Target"

**Introduced in R2009a**

# rtIOStreamRecv

Receive data through communication channel

## Syntax

errFlg = rtIOStreamRecv(streamID, dest, size, receivedDataSize)

## Description

errFlg = rtIOStreamRecv(streamID, dest, size, receivedDataSize) receives data through a communication channel.

## Examples

### Send and Receive Data from Processor

This code from rtiostreamtest.c shows how to send and receive data from a target processor.

```
static void blockingIO(int send, unsigned long numMemUnits)
{
    size_t sizeToTransfer = (size_t) numMemUnits;
    size_t sizeTransferred;
    IOUnit_T * ioPtr = (IOUnit_T *) &buff[0];
    int status;

    if (numMemUnits > BUFFER_SIZE)
    {
        AckCode = stat_notEnoughSpace;
        AckArg0 = BUFFER_SIZE;
        return;
    }

#ifdef HOST_WORD_ADDRESSABLE_TESTING
   /* map to bytes */
   sizeToTransfer *= MEM_UNIT_BYTES;
#endif

   while (sizeToTransfer > 0) {
```

```
            sizeTransferred = 0;
            /* Do the low level call */
            status = send ?
               rtIOStreamSend(streamID, ioPtr, sizeToTransfer, &sizeTransferred) :
               rtIOStreamRecv(streamID, ioPtr, sizeToTransfer, &sizeTransferred);

            if (status != RTIOSTREAM_NO_ERROR) {
               if (AckCode == stat_OK) {
                  AckCode = stat_RTIOSTREAM_ERROR;
                  AckArg0 = data_counter;
               }
               return;
            }
            else {
               sizeToTransfer -= sizeTransferred;
               ioPtr += sizeTransferred;
            }
         }
      }
```

## Input Arguments

### `streamID` — Stream handle

scalar integer

Handle to the stream returned by a previous call to `rtIOStreamOpen`.

### `dest` — Data destination

void pointer

Pointer to the start of the buffer for received data.

### `size` — Size of data to copy

size_t

Size of data to copy into the destination buffer. For byte-addressable architectures, size is measured in bytes. Some DSP architectures are not byte-addressable. In these cases, size is measured in number of WORDs, where `sizeof(WORD) == 1`.

### `receivedDataSize` — Size of data received

size_t

Number of units of data received and copied into the buffer `dest`. If no data is copied, value is zero.

## Output Arguments

**`errFlg` — Error flag**

0 | -1

If the function runs without errors, it returns zero. Otherwise, it returns -1.

The `rtiostream.h` file defines these macros:

```
#define RTIOSTREAM_ERROR (-1)
#define RTIOSTREAM_NO_ERROR (0)
```

## See Also

`rtIOStreamSend` | `rtIOStreamOpen` | `rtIOStreamClose` | `rtiostream_wrapper`

### Topics

"Create PIL Target Connectivity Configuration for MATLAB"
"Create PIL Target Connectivity Configuration for Simulink"
"Create a Target Communication Channel for Processor-In-The-Loop (PIL) Simulation"
"Configure Processor-In-The-Loop (PIL) for a Custom Target"

**Introduced in R2009a**

# rtIOStreamSend

Send data through communication channel

## Syntax

```
errFlag = rtIOStreamSend(streamID, src, size, sizeSent)
```

## Description

`errFlag = rtIOStreamSend(streamID, src, size, sizeSent)` sends data through a communication stream.

The API for `rtIOStream` functions is independent of the physical layer across which you send the data, for example, RS232, Ethernet, or Controller Area Network (CAN). The choice of physical layer affects the achievable data rates for communication between your development computer and target processor.

For a processor-in-the-loop (PIL) application, there is no minimum data rate requirement. The higher the data rate is, the faster the simulation runs.

A communications device driver can require additional hardware-specific or channel-specific configuration parameters. For example:

- A CAN channel can require the specification of the CAN node that is used.
- A TCP/IP channel can require the configuration of a port or static IP address.
- A CAN channel can require the specification of the CAN message ID and priority.

When you implement the `rtIOStream` driver functions, provide this configuration data, for example, by hard-coding the data or by supplying arguments to `rtIOStreamOpen`.

# Examples

## Send and Receive Data from Processor

This code from `rtiostreamtest.c` shows how to send and receive data from a target processor.

```
static void blockingIO(int send, unsigned long numMemUnits)
{
    size_t sizeToTransfer = (size_t) numMemUnits;
    size_t sizeTransferred;
    IOUnit_T * ioPtr = (IOUnit_T *) &buff[0];
    int status;

    if (numMemUnits > BUFFER_SIZE)
    {
        AckCode = stat_notEnoughSpace;
        AckArg0 = BUFFER_SIZE;
        return;
    }

#ifdef HOST_WORD_ADDRESSABLE_TESTING
   /* map to bytes */
   sizeToTransfer *= MEM_UNIT_BYTES;
#endif

   while (sizeToTransfer > 0) {
      sizeTransferred = 0;
      /* Do the low level call */
      status = send ?
         rtIOStreamSend(streamID, ioPtr, sizeToTransfer, &sizeTransferred) :
         rtIOStreamRecv(streamID, ioPtr, sizeToTransfer, &sizeTransferred);

      if (status != RTIOSTREAM_NO_ERROR) {
         if (AckCode == stat_OK) {
            AckCode = stat_RTIOSTREAM_ERROR;
            AckArg0 = data_counter;
         }
         return;
      }
      else {
         sizeToTransfer -= sizeTransferred;
         ioPtr += sizeTransferred;
      }
   }
}
```

## Input Arguments

### `streamID` — Stream handle

scalar integer

Handle to the stream returned by a previous call to `rtIOStreamOpen`.

### `src` — Data source

constant void pointer

Pointer to the start of the buffer that contains a data array for transmission.

### `size` — Size of data to transmit

size_t

Size of data to transmit from the source buffer. For byte-addressable architectures, size is measured in bytes. Some DSP architectures are not byte-addressable. In these cases, size is measured in number of WORDs, where `sizeof(WORD) == 1`.

## Output Arguments

### `errFlag` — Error flag

0 | -1

If the function runs without errors, it returns zero. Otherwise, it returns -1.

The `rtiostream.h` file defines these macros:

```
#define RTIOSTREAM_ERROR (-1)
#define RTIOSTREAM_NO_ERROR (0)
```

### `sizeSent` — Size of transmitted data

size_t pointer

Size of transmitted data, which is less than or equal to `size`. If data is not transmitted, value is zero.

## See Also

`rtIOStreamOpen` | `rtIOStreamClose` | `rtIOStreamRecv` | `rtiostream_wrapper`

## Topics

"Create PIL Target Connectivity Configuration for MATLAB"
"Create PIL Target Connectivity Configuration for Simulink"
"Create a Target Communication Channel for Processor-In-The-Loop (PIL) Simulation"
"Configure Processor-In-The-Loop (PIL) for a Custom Target"

**Introduced in R2009a**

# rtiostream_wrapper

Test `rtiostream` shared library functions in MATLAB

## Syntax

```
streamID = rtiostream_wrapper(sharedLib, 'open')
[errFlag,transmittedDataSize] = rtiostream_wrapper(sharedLib,'send',
streamID,data,dataSize)
[errFlag,receivedData,receivedDataSize] = rtiostream_wrapper(
sharedLib,'recv',streamID,dataSize)
streamID = rtiostream_wrapper( ___ ,Name,Value)
errFlag = rtiostream_wrapper(sharedLib,'close',streamID)
rtiostream_wrapper(sharedLib,'unloadlibrary')
```

## Description

`streamID = rtiostream_wrapper(sharedLib, 'open')` opens an `rtiostream` communication channel or stream through a shared library.

`[errFlag,transmittedDataSize] = rtiostream_wrapper(sharedLib,'send', streamID,data,dataSize)` transmits data from a workspace variable through the open communication channel or stream.

`[errFlag,receivedData,receivedDataSize] = rtiostream_wrapper( sharedLib,'recv',streamID,dataSize)` receives workspace variable data from the open communication channel or stream.

`streamID = rtiostream_wrapper( ___ ,Name,Value)` specifies additional options using one or more name-value pair arguments. These arguments are implementation-dependent, that is, they are specific to the shared library that you use.

`errFlag = rtiostream_wrapper(sharedLib,'close',streamID)` closes the `rtiostream` communication channel or stream.

`rtiostream_wrapper(sharedLib,'unloadlibrary')` unloads the shared library, clearing persistent data.

# Examples

## Open Communication Channels

These examples use the supplied TCP/IP and serial communication drivers to open communication channels.

Open `rtiostream` `stationA` as a TCP/IP server:

```
stationA = rtiostream_wrapper('libmwrtiostreamtcpip.dll','open',...
                              '-client', '0',...
                              '-blocking', '0',...
                              '-port', port_number);
```

Opens `rtiostream` `StationB` as a TCP/IP client:

```
stationB = rtiostream_wrapper('libmwrtiostreamtcpip.dll','open',...
                              '-client','1',...
                              '-blocking', '0',...
                              '-port', port_number,...
                              '-hostname','localhost');
```

If you use the supplied development computer driver for serial communications (as an alternative to the drivers for TCP/IP), specify the bit rate when you open a channel with a specific port. For example, open channel `stationA` with port `COM1` and bit rate of 9600:

```
stationA = rtiostream_wrapper('libmwrtiostreamserial.dll','open',...
                              '-port','COM1',...
                              '-baud','9600');
```

# Input Arguments

**`sharedLib` — Shared library**
character vector

Shared library that implements required `rtIOStream` functions, `rtIOStreamOpen`, `rtIOStreamSend`, `rtIOStreamRecv`, and `rtIOStreamClose`. Must be on system path. Specify one of these values:

- *`libTCPIP`* — For TCP/IP communication. Value depends on your operating system, for example, `'libmwrtiostreamtcpip.dll'`
- *`libSerial`* — For serial communication, for example, `'libmwrtiostreamserial.dll'`.

**`streamID` — Stream handle**
scalar integer

Handle to `rtiostream` communication stream.

**`data` — Workspace variable**
array

Array that contains data for transmission.

**`dataSize` — Size of data to transmit or receive**
scalar integer

Size of workspace variable data to transmit or receive, in bytes.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: '-hostname','localhost'

**TCP/IP Communication**

**`-client` — Stream type**
0 | 1

Open `rtiostream` as TCP/IP server or client:

- 0 — TCP/IP server
- 1 — TCP/IP client

**`-port` — Port number**
scalar integer

Port for TCP/IP communication.

**`-hostname` — Development computer**
character vector

Identifier for your development computer, for example, `'localhost'`.

**`-blocking` — Call behavior**
0 | 1

Call behavior when receiving data:

- 0 — Polling mode. If data is available, call returns with data. If data is not available, call returns without waiting.
- 1 — Blocking mode. If data is available, call returns with data. If data is not available, call waits for data. Use `recv_timeout_secs` to specify the waiting period.

Default is 0 unless the preprocessor macro `define VXWORKS` exists. In this case, the default is 1.

**`-recv_timeout_secs` — Waiting period**
positive integer | 0 | -1 | -2 | -3

Waiting period of call that receives data:

- *X*, a positive integer — Wait for *X* seconds.
- 0 — No waiting period.
- -1 — Wait indefinitely.
- -2 — Wait for default period.
- -3 — Wait 10 ms.

Default for client connections is to wait 1 second. Default for server connections is to wait indefinitely.

#### Serial Communication

**`-port` — Port number**
scalar integer

COM port for serial communication.

**`-baud` — Bit rate**
scalar integer

Bit rate for serial communication port.

## Output Arguments

**`streamID` — Stream handle**
scalar integer

If the function opens the communications stream, it returns a handle to the stream. Otherwise, it returns -1.

**`errFlag` — Error flag**
0 | -1

If the function runs without errors, it returns zero. Otherwise, it returns -1.

**`transmittedDataSize` — Size of transmitted data**
scalar integer

Size of data transmitted through communication stream. Can be less than `dataSize`, the number of bytes specified for transmission.

**`receivedData` — Workspace variable received**
array

Array that contains received data.

**`receivedDataSize` — Size of received data**
scalar integer

Number of bytes received from communication stream. Can be less than `dataSize`, the number of bytes specified for transmission.

## See Also

`rtIOStreamOpen` | `rtIOStreamSend` | `rtIOStreamRecv` | `rtIOStreamClose`

### Topics

"Create PIL Target Connectivity Configuration for MATLAB"
"Create PIL Target Connectivity Configuration for Simulink"
"Create a Target Communication Channel for Processor-In-The-Loop (PIL) Simulation"
"Configure Processor-In-The-Loop (PIL) for a Custom Target"

**Introduced in R2008b**

# rtw.codegenObjectives.Objective class

**Package:** rtw.codegenObjectives

Customize code generation objectives

## Description

An `rtw.codegenObjectives.Objective` object creates a code generation objective.

## Construction

| | |
|---|---|
| rtw.codegenObjectives.Objective | Create custom code generation objectives |

## Methods

| | |
|---|---|
| addCheck | Add checks |
| addParam | Add parameters |
| excludeCheck | Exclude checks |
| modifyInheritedParam | Modify inherited parameter values |
| register | Register objective |
| removeInheritedCheck | Remove inherited checks |
| removeInheritedParam | Remove inherited parameters |
| setObjectiveName | Specify objective name |

## Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects (MATLAB) in the MATLAB Programming Fundamentals documentation.

## Examples

Create a custom objective named `Reduce RAM Example`. The following code is the contents of the `sl_customization.m` file that you create.

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

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

end

function addObjectives

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

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

% Add additional checks to the objective
% The Code Generation Advisor automatically includes 'Check model
% configuration settings against code generation objectives' in every
% objective.
addCheck(obj, 'mathworks.design.UnconnectedLinesPorts');
addCheck(obj, 'mathworks.design.Update');

%Register the objective
register(obj);

end
```

## See Also

### Topics

"Create Custom Code Generation Objectives"

# rtw.codegenObjectives.Objective

**Class:** rtw.codegenObjectives.Objective
**Package:** rtw.codegenObjectives

Create custom code generation objectives

## Syntax

```
obj = rtw.codegenObjectives.Objective('objID')
obj = rtw.codegenObjectives.Objective('objID', 'base_objID')
```

## Description

`obj = rtw.codegenObjectives.Objective('objID')` creates an objective object, `obj`.

`obj = rtw.codegenObjectives.Objective('objID', 'base_objID')` creates an object, `obj`, for a new objective that is identical to an existing objective. You can then modify the new objective to meet your requirements.

## Input Arguments

*objID* — A permanent, unique identifier for the objective.

- You must have

  *objID*.
- The value of *objID* must remain constant.
- When you refresh your customizations, if *objID* is not unique, Simulink generates an error.

*base_objID* — The identifier of the objective that you want to base the new objective on.

## Examples

Create a new objective:

```
obj = rtw.codegenObjectives.Objective('ex_ram_1');
```

Create a new objective based on the existing `Execution efficiency` objective:

```
obj = rtw.codegenObjectives.Objective('ex_my_efficiency_1', 'Execution efficiency');
```

## See Also

### Topics

“Create Custom Code Generation Objectives”

# RTW.configSubsystemBuild

**Package:** RTW

Configure C function prototype or C++ class interface for right-click build of specified subsystem

## Syntax

`RTW.configSubsystemBuild(`*`block`*`)`

## Description

`RTW.configSubsystemBuild(`*`block`*`)` opens a graphical user interface where you can configure either C function prototype information or C++ class interface information for right-click builds of a specified nonvirtual subsystem. A dialog box opens based on the **Language** and **Code interface packaging** values selected for your model on the **Code Generation** and **Code Generation > Interface** panes of the Configuration Parameters dialog box.

To configure and generate C++ class interfaces for a nonvirtual subsystem, you must

- Select the system target file `ert.tlc` for the model.
- Select the **Language** parameter value `C++` for the model.
- Select the **Code interface packaging** parameter value `C++ class` for the model.
- Make sure that the subsystem is convertible to a Model block using the function `Simulink.SubSystem.convertToModelReference`. For referenced model conversion requirements, see the Simulink reference page `Simulink.SubSystem.convertToModelReference`.

## Input Arguments

| | |
|---|---|
| *`block`* | Character vector specifying the name of a nonvirtual subsystem block in an ERT-based Simulink model. |

## See Also

### Topics

“Customize Function Interfaces for Nonvirtual Subsystems”
“Customize Generated C Function Interfaces”
“Configure C++ Class Interfaces for Nonvirtual Subsystems”
“Customize Generated C++ Class Interfaces”

**Introduced in R2008b**

# rtw.connectivity.ComponentArgs

Provide parameters for each target connectivity component

## Description

An `rtw.connectivity.ComponentArgs` object provides functions for getting information about the source component and the target application.

## Creation

### Description

`compArgs = rtw.connectivity.ComponentArgs (componentPath, componentCodePath, componentCodeName, applicationCodePath)` returns a handle to an `rtw.connectivity.ComponentArgs` object.

## Object Functions

| Function | Description |
| --- | --- |
| `getComponentPath` | `*cmpPath* = *compArgs*.getComponentPath` returns the system path of the source component. For example:<br>• For MATLAB, the path of the function that is under test.<br>• For Simulink, the path of the referenced model that is under test. |

| Function | Description |
|---|---|
| `getComponentCodePath` | *`cmpCodePath`* `=` *`compArgs`*`.getComponentCodePath` returns the code generation folder path associated with the source component. For example:<br>• For MATLAB, the code generation folder of the MATLAB function that is under test.<br>• For Simulink, the code generation folder of the referenced model that is under test. |
| `getComponentCodeName` | *`cmpCodeName`* `=` *`compArgs`*`.getComponentCodeName` returns the component name used for code generation. |
| `getApplicationCodePath` | *`appCodePath`* `=` *`compArgs`*`.getApplicationCodePath` returns the folder path associated with the target application, for example, the path associated with the PIL application. |
| `getParam` | *`settingOrParameterValue`* `=` *`compArgs`*`.getParam(`*`settingOrParameterName`*`)` returns:<br>• For MATLAB, the value of the specific MATLAB Coder setting for the generated code.<br>• For Simulink, the value of the specific model configuration parameter for the generated code. The function does not load the model. |

# Examples

## Using `rtw.connectivity.ComponentArgs` in PIL Connectivity

For an example that shows how to use this object in setting up PIL connectivity, see:

- For MATLAB, “Processor-in-the-Loop Execution From Command Line”
- For Simulink, “Configure Processor-In-The-Loop (PIL) for a Custom Target”

# See Also

`rtw.connectivity.Config`

## Topics

"Create PIL Target Connectivity Configuration for MATLAB"
"Create PIL Target Connectivity Configuration for Simulink"

**Introduced in R2008b**

# rtw.connectivity.Config

Define connectivity implementation that comprises builder, launcher, and communicator components

## Description

The `rtw.connectivity.Config` class specifies the actions required for running a processor-in-the-loop (PIL) simulation.

## Creation

### Description

`rtw.connectivity.Config(componentArgs, builder, launcher, communicator)` creates an `rtw.connectivity.Config` object with these arguments:

- *`componentArgs`* – `rtw.connectivity.ComponentArgs` object
- *`builder`* – `rtw.connectivity.Builder` object, for example, `rtw.connectivity.MakefileBuilder` object.
- *`launcher`* – `rtw.connectivity.Launcher` object
- *`communicator`* – `rtw.connectivity.Communicator`, for example, `rtw.connectivity.RtIOStreamHostCommunicator` object.

To define a connectivity implementation:

1. Create a subclass of `rtw.connectivity.Config` that creates instances of your connectivity component classes:

   - `rtw.connectivity.MakefileBuilder`
   - `rtw.connectivity.Launcher`
   - `rtw.connectivity.RtIOStreamHostCommunicator`
2. Define the constructor for your subclass:

   ```
   function this = myConfig(componentArgs)
   ```

When the software creates an instance of your subclass of `rtw.connectivity.Config`, it provides an instance of the `rtw.connectivity.ComponentArgs` class as the only constructor argument. If you want to test your subclass of `rtw.connectivity.Config` manually, you can create an `rtw.connectivity.ComponentArgs` object to pass as a constructor argument.

3. After instantiating the builder, launcher and communicator objects in your subclass, call the constructor of the superclass `rtw.connectivity.Config` to define your complete target connectivity configuration. For example:

```
this@rtw.connectivity.Config(componentArgs,...
builder, launcher, communicator);
```

4. Optionally, for execution-time profiling, use the `setTimer` method to register your hardware timer. For example, if you specified the timer in a code replacement table, insert the following line:

```
this.setTimer('myCrlTable')
```

*`myCrlTable`* is the name of the code replacement table, which must be in a location on the MATLAB search path.

Register your subclass name, for example, `myPIL.ConnectivityConfig` by using the class `rtw.connectivity.ConfigRegistry`. The PIL infrastructure instantiates your subclass as required. The `rtwTargetInfo.m` file (for MATLAB) or `sl_customization.m` mechanism (for Simulink) specifies a suitable connectivity configuration for use with a particular PIL component (and its configuration set). The subclass can also perform additional validation on construction. For example, you can use the component path returned by the `getComponentPath` method of the `componentArgs` constructor argument to query and validate parameters associated with the PIL component under test.

# Examples

## Using `rtw.connectivity.Config` in PIL Connectivity

For an example that shows how to use this object in setting up PIL connectivity, see:

- For MATLAB, “Processor-in-the-Loop Execution From Command Line”

- For Simulink, “Configure Processor-In-The-Loop (PIL) for a Custom Target”

## See Also

`rtw.connectivity.MakefileBuilder` | `rtw.connectivity.Launcher` | `rtw.connectivity.RtIOStreamHostCommunicator` | `rtw.connectivity.ComponentArgs`

### Topics

“Create PIL Target Connectivity Configuration for MATLAB”
“Create PIL Target Connectivity Configuration for Simulink”
“Specify Hardware Timer”
“Specify Hardware Timer”
“SIL and PIL Limitations”

**Introduced in R2008b**

# rtw.connectivity.ConfigRegistry

Register connectivity configuration

## Description

Register your connectivity configuration with MATLAB or Simulink.

## Creation

### Description

`config = rtw.connectivity.ConfigRegistry` returns a handle to an `rtw.connectivity.ConfigRegistry` object.

To create this class:

- For MATLAB, use an `rtwTargetInfo.m` file, which you must place on the MATLAB search path. In the `rtwTargetInfo.m` file, a call to `registerTargetInfo` registers the connectivity configuration.
- For Simulink, use an `sl_customization.m` file, which you must place on the MATLAB search path. When Simulink starts, it reads the file, and registers your connectivity configuration through a call to `registerTargetInfo` in the file.

Through the first two properties of this class, you can specify for your connectivity configuration:

- A unique name.
- An associated connectivity implementation class, which is a subclass of `rtw.connectivity.Config`.

Through the remaining properties, you can define:

- For MATLAB, the code that is compatible with the connectivity implementation class.
- For Simulink, the set of models that are compatible with the connectivity implementation class.

A comparison of the union of these properties against the MATLAB Coder configuration settings or Simulink model parameters determines compatibility. For example with Simulink, whether the `SystemTargetFile`, `TemplateMakefile`, and `HardwareBoard` properties jointly match the corresponding model parameters.

## Properties

**`ConfigName` — Name**
character vector

Unique name for configuration.

**`ConfigClass` — Class name**
character vector

Full class name of the connectivity implementation that you want to register.

**`SystemTargetFile` — System target files (Simulink)**
cell array of character vectors | {}

For Simulink, system target files that support the `rtw.connectivity.ConfigRegistry` object you create. A comparison of this cell array against the `SystemTargetFile` configuration parameter of the model determines whether the created object is valid for use. An empty cell array matches any system target file.

**`TemplateMakefile` — Template makefiles (Simulink)**
cell array of character vectors | {}

For Simulink, template makefiles that support the `rtw.connectivity.ConfigRegistry` object you create. A comparison of this cell array against the `TemplateMakefile` configuration parameter of the model determines whether the created object is valid for use. An empty cell array matches any template makefile and non-makefile target (`GenerateMakefile`: off).

If you use a toolchain to build the generated code, do not specify the `TemplateMakefile` configuration parameter. Instead, specify the `Toolchain` configuration parameter.

**`Toolchain` — Toolchains**
cell array of character vectors | {}

Toolchains that support the `rtw.connectivity.ConfigRegistry` object you create:

- For MATLAB, a comparison of this cell array against the MATLAB Coder `Toolchain` configuration setting determines whether the created object is valid for use.
- For Simulink, a comparison of this cell array against the `Toolchain` configuration parameter of the model determines whether the created object is valid for use. If you do not use a toolchain to build the generated code, do not specify the `Toolchain` configuration parameter. Instead, specify the `TemplateMakefile` configuration parameter.

An empty cell array matches any toolchain.

**`HardwareBoard` — Hardware boards**
cell array of character vectors | {}

Hardware boards that support the `rtw.connectivity.ConfigRegistry` object you create:

- For MATLAB, a comparison of this cell array against the MATLAB Coder `HardwareBoard` configuration setting determines whether the created object is valid for use.
- For Simulink, a comparison of this cell array against the `HardwareBoard` configuration parameter of the model determines whether the created object is valid for use.

An empty cell array matches any hardware board.

**`TargetHWDeviceType` — Hardware device types**
cell array of character vectors | {}

Hardware device types that support the `rtw.connectivity.ConfigRegistry` object you create:

- For MATLAB, a comparison of this cell array against the MATLAB Coder `TargetHWDeviceType` configuration setting determines whether the created object is valid for use.
- For Simulink, a comparison of this cell array against the `TargetHWDeviceType` configuration parameter of the model determines whether the created object is valid for use.

An empty cell array matches any hardware device type.

# Examples

## Create `rtwTargetInfo.m` File

This code is an example `rtwTargetInfo.m` file. Use the function syntax exactly as shown.

```
function rtwTargetInfo(tr)
% Register PIL connectivity config: mypil.ConnectivityConfig

tr.registerTargetInfo(@loc_createConfig);

% local function
function config = loc_createConfig

% Create object for connectivity configuration
config = rtw.connectivity.ConfigRegistry;
% Assign connectivity configuration name
config.ConfigName = 'My PIL Example';
% Associate the connectivity configuration with the connectivity
% API implementation
config.ConfigClass = 'mypil.ConnectivityConfig';

% Specify toolchains for host-based PIL
config.Toolchain =  rtw.connectivity.Utils.getHostToolchainNames;

% Through the HardwareBoard and TargetHWDeviceType properties,
% define compatible code for the target connectivity configuration
config.HardwareBoard = {};
config.TargetHWDeviceType = {'Generic->32-bit x86 compatible' ...
                             'Generic->Custom' ...
                             'Intel->x86-64 (Windows64)', ...
                             'Intel->x86-64 (Mac OS X)', ...
                             'Intel->x86-64 (Linux 64)'};
```

The function performs the following steps:

1. Creates an instance of the `rtw.connectivity.ConfigRegistry` class. For example:

   ```
   config = rtw.connectivity.ConfigRegistry;
   ```
2. Assigns a connectivity configuration name to the `ConfigName` property of the object. For example:

```
config.ConfigName = 'My PIL Example';
```

3. Associates the connectivity configuration with the connectivity API implementation created in step 1. For example:

```
config.ConfigClass = 'mypil.ConnectivityConfig';
```

4. Defines compatible code for this target connectivity configuration, by setting the `HardwareBoard` and `TargetHWDeviceType` properties of the object. For example:

```
config.HardwareBoard = {};  % Any hardware board
config.TargetHWDeviceType = {'Generic->32-bit x86 compatible' ...
                             'Generic->Custom' ...
                             'Intel->x86-64 (Windows64)', ...
                             'Intel->x86-64 (Mac OS X)', ...
                             'Intel->x86-64 (Linux 64)'};
```

## Create sl_customization.m File

This code is an example of an `sl_customization.m` file. Use the `sl_customization.m` file structure, and call the `registerTargetInfo` function exactly as shown.

```
function sl_customization(cm)
% SL_CUSTOMIZATION for PIL connectivity config:...
% mypil.ConnectivityConfig

% Copyright 2008 The MathWorks, Inc.

cm.registerTargetInfo(@loc_createConfig);

% local function
function config = loc_createConfig

config = rtw.connectivity.ConfigRegistry;
config.ConfigName = 'My PIL Example';
config.ConfigClass = 'mypil.ConnectivityConfig';

% Match only ert.tlc
config.SystemTargetFile = {'ert.tlc'};

% If you use a toolchain to build your generated code,
% specify the config.Toolchain property to match your
% Simulink model toolchain setting. Otherwise, for a
% non-toolchain approach, match the TMF
```

```
config.TemplateMakefile = {'ert_default_tmf' ...
                           'ert_unix.tmf', ...
                           'ert_vcx64.tmf', ...
                           'ert_lcc.tmf'};

% Match hardware boards and hardware device types
config.HardwareBoard = {};  % Any hardware board
config.TargetHWDeviceType = {'Generic->32-bit x86 compatible' ...
                             'Generic->Custom' ...
                             'Intel->x86-64 (Windows64)', ...
                             'Intel->x86-64 (Mac OS X)', ...
                             'Intel->x86-64 (Linux 64)'};
```

You must configure the file to perform the following steps when Simulink starts:

1. Create an instance of the `rtw.connectivity.ConfigRegistry` class. For example:

   ```
   config = rtw.connectivity.ConfigRegistry;
   ```
2. Assign a connectivity configuration name to the `ConfigName` property of the object. For example:

   ```
   config.ConfigName = 'My PIL Example';
   ```
3. Associate the connectivity configuration with the connectivity API implementation (created in step 1). For example:

   ```
   config.ConfigClass = 'mypil.ConnectivityConfig';
   ```
4. Define compatible models for this target connectivity configuration, by setting these properties of the properties of the object:

   - `SystemTargetFile`
   - `Toolchain` or `TemplateMakefile`
   - `HardwareBoard`
   - `TargetHWDeviceType`

   For example:

   ```
   config.SystemTargetFile = {'ert.tlc'};
   config.TemplateMakefile = {'ert_default_tmf' ...
                              'ert_unix.tmf', ...
                              'ert_vcx64.tmf', ...
   ```

```
                              'ert_lcc.tmf'};
config.HardwareBoard = {};
config.TargetHWDeviceType = {'Generic->32-bit x86 compatible' ...
                              'Generic->Custom' ...
                              'Intel->x86-64 (Windows64)', ...
                              'Intel->x86-64 (Mac OS X)', ...
                              'Intel->x86-64 (Linux 64)'};
```

### Using `rtw.connectivity.ConfigRegistry` in PIL Connectivity

For an example that shows how to use this object in setting up PIL connectivity, see:

- For MATLAB, “Processor-in-the-Loop Execution From Command Line”
- For Simulink, “Configure Processor-In-The-Loop (PIL) for a Custom Target”

## See Also

`rtw.connectivity.Config`

### Topics

“Create PIL Target Connectivity Configuration for MATLAB”
“Create PIL Target Connectivity Configuration for Simulink”
“Register Code Replacement Mappings”

**Introduced in R2008b**

# rtw.connectivity.Launcher

Control downloading, starting, and resetting of a target application

## Description

The `rtw.connectivity.Launcher` class, which runs on your development computer, controls execution of an application on the target processor.

## Creation

### Description

`rtw.connectivity.Launcher(componentArgs)` controls the download, start, and reset of an application, for example, a PIL application.

Make a subclass and implement the `startApplication` and `stopApplication` methods.

You can implement a destructor method that cleans up resources (for example, a handle to a third-party download tool) when the object is cleared from memory.

## Object Functions

| Function | Description |
| --- | --- |
| `getComponentArgs` | `componentArgs = obj.getComponentArgs` returns the `rtw.connectivity.ComponentArgs` object associated with the launcher object. |
| `setExe` | `setExe(exe)` specifies the application that runs on the target processor. |
| `getExe` | `exe=getExe()` returns the application that is running on the target processor. |

| Function | Description |
| --- | --- |
| `startApplication` | `obj.startApplication` is an abstract method that you implement in a subclass. Called by MATLAB or Simulink to start execution of the target application.<br><br>MATLAB or Simulink calls the `setExe` method, which specifies the target application to run. To obtain this application, use the `getExe` method. For example:<br><br>`exe = getExe()`<br><br>The `startApplication` method resets the application to its initial state by ensuring that external and static (global) variables are zero-initialized. |
| `stopApplication` | `obj.stopApplication` is an abstract method that you must implement in a subclass.<br><br>Called by MATLAB to stop execution of the target application. |
| `getBuilder` | `builder = obj.getBuilder` returns the `rtw.connectivity.Builder` object associated with the launcher object. |

## Examples

### Using `rtw.connectivity.Launcher` in PIL Connectivity

For an example that shows how to use this object in setting up PIL connectivity, see:

- For MATLAB, “Processor-in-the-Loop Execution From Command Line”
- For Simulink, “Configure Processor-In-The-Loop (PIL) for a Custom Target”

## See Also

`rtw.connectivity.MakefileBuilder` | `rtw.connectivity.RtIOStreamHostCommunicator` | `rtw.connectivity.ComponentArgs`

## Topics

“Create PIL Target Connectivity Configuration for MATLAB”
“Create PIL Target Connectivity Configuration for Simulink”

**Introduced in R2008b**

# rtw.connectivity.MakefileBuilder

Configure toolchain-based build process

## Description

Control toolchain-based build process for the creation of a PIL application.

## Creation

### Description

`rtw.connectivity.MakefileBuilder(componentArgs, targetApplicationFramework, exeExtension)` creates an object with these arguments:

- *`componentArgs`* - An `rtw.connectivity.ComponentArgs` object
- *`TargetApplicationFramework`* - An `rtw.pil.RtIOStreamApplicationFramework` object. For example, `myPIL.TargetFramework`.
- *`exeExtension`* - Name extension of executable file for target system.The extension depends on the toolchain defined by `rtw.connectivity.ConfigRegistry`. For an embedded target, the extension can be, for example, `'.elf'`, `'.abs'`, `'.sre'`, or `'.hex'`. For a Windows development computer target, the extension is `'.exe'`. For a UNIX® development computer target, the extension is empty, `''`.

## Examples

### Using `rtw.connectivity.MakefileBuilder` in PIL Connectivity

For an example that shows how to use this object in setting up PIL connectivity, see:

- For MATLAB, “Processor-in-the-Loop Execution From Command Line”
- For Simulink, “Configure Processor-In-The-Loop (PIL) for a Custom Target”

## See Also

`rtw.connectivity.ComponentArgs` | `rtw.pil.RtIOStreamApplicationFramework`

### Topics

“Create PIL Target Connectivity Configuration for MATLAB”
“Create PIL Target Connectivity Configuration for Simulink”

**Introduced in R2008b**

# rtw.connectivity.RtIOStreamHostCommunicator

Configure development computer communications with target processor

## Description

Configure communications between your development computer and the target processor by loading and initializing a shared library that implements the `rtiostream` functions.

## Creation

### Description

`rtw.connectivity.RtIOStreamHostCommunicator(componentArgs, launcher, rtiostreamLib)` creates an object by using these arguments:

- *`componentArgs`* -- `rtw.connectivity.ComponentArgs` object.
- *`launcher`* -- `rtw.connectivity.Launcher` object.
- *`rtiostreamLib`* -- `rtiostream` shared library that implements the development computer part of communications between the development computer and the target processor.

The object loads and initializes the shared library.

For your development computer, Embedded Coder provides a shared library for these communication protocols:

- TCP/IP
- serial

You must provide drivers for the target processors.

For other communication protocols, for example, USB, you must provide a shared library for the development computer and drivers for the target processors.

To create your instance of `rtw.connectivity.RtIOStreamHostCommunicator`, you have these options:

- Instantiate `rtw.connectivity.RtIOStreamHostCommunicator` directly, providing custom arguments for the `rtiostream` shared library.
- Create a subclass of `rtw.connectivity.RtIOStreamHostCommunicator`. Consider this option when more complex configuration is required. For example, when:
  - The subclass `rtw.connectivity.HostTCPIPCommunicator` includes additional code to determine the number of the TCP/IP port that the executable application serves.
  - You use a subclass to specify a serial port number.
  - You specify verbose or silent operation.

# Object Functions

| Function | Description |
| --- | --- |
| `setTimeoutRecvSecs` | `hostCommunicator.setTimeoutRecvSecs(`*`timeout`*`)` sets the timeout value for reading data. You can configure data reading to time out if no new data is received for a period greater than `timeout` seconds. |
| `setInitCommsTimeout` | `hostCommunicator.setInitCommsTimeout(`*`timeout`*`)` sets the timeout value for initial setup of the communications channel. For some target processors, you might need to set a timeout value for initial setup of the communications channel. For example, the target processor can take a few seconds to open its side of the communications channel. If you set a nonzero timeout value, the communicator repeatedly tries to open the communications channel until the timeout value is reached. |

## Examples

### Using `rtw.connectivity.RtIOStreamHostCommunicator` in PIL Connectivity

For an example that shows how to use this object in setting up PIL connectivity, see:

- For MATLAB, "Processor-in-the-Loop Execution From Command Line"
- For Simulink, "Configure Processor-In-The-Loop (PIL) for a Custom Target"

## See Also

`rtiostream_wrapper` | `rtw.connectivity.ComponentArgs` | `rtw.connectivity.Launcher`

### Topics

"Create PIL Target Connectivity Configuration for MATLAB"
"Create PIL Target Connectivity Configuration for Simulink"

**Introduced in R2008b**

# RTW.getClassInterfaceSpecification

**Package:** RTW

Get handle to model-specific C++ class interface control object

## Syntax

`obj = RTW.getClassInterfaceSpecification(modelName)`

## Description

`obj = RTW.getClassInterfaceSpecification(modelName)` returns a handle to a model-specific C++ class interface control object.

## Input Arguments

| | |
|---|---|
| *modelName* | Character vector specifying the name of a loaded ERT-based Simulink model. |

## Output Arguments

| | |
|---|---|
| `obj` | Handle to the C++ class interface control object associated with the specified model. If the model does not have an associated C++ class interface control object, the function returns `[]`. |

## Alternatives

The **Configure C++ Class Interface** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Configure C++ class interface dialog box, where you can flexibly control the C++ class interfaces that are generated for your model. Once you validate and apply your changes, you can generate code based on your C

++ class interface modifications. See “Customize C++ Class Interfaces Using Graphical Interfaces”.

## See Also

### Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

**Introduced in R2014a**

# RTW.getFunctionSpecification

**Package:** RTW

Get handle to model-specific C prototype function control object

## Syntax

`obj = RTW.getFunctionSpecification(modelName)`

## Description

*obj* = RTW.getFunctionSpecification(*modelName*) returns a handle to the model-specific C function prototype control object.

## Input Arguments

| | |
|---|---|
| *modelName* | Character vector specifying the name of a loaded ERT-based Simulink model. |

## Output Arguments

| | |
|---|---|
| *obj* | Handle to the model-specific C prototype function control object associated with the specified model. If the model does not have an associated function control object, the function returns `[]`. |

## Alternatives

The **Configure Model Functions** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Model Interface dialog box, which provides you flexible control over the C function prototypes that are generated for your

model. Once you validate and apply your changes, you can generate code based on your C function prototype modifications. See “Customize Generated C Function Interfaces”.

## See Also

### Topics

“Customize Generated C Function Interfaces”

**Introduced in R2008a**

# RTW.ModelCPPArgsClass class

**Package:** RTW
**Superclasses:**

Control C++ class interfaces for models using I/O arguments style step method

## Description

The `ModelCPPArgsClass` class provides objects that describe C++ class interfaces for models using an I/O arguments style step method. Use the `attachToModel` method to attach a C++ class interface to a loaded ERT-based Simulink model.

## Construction

| | |
|---|---|
| RTW.ModelCPPArgsClass | Create C++ class interface object for configuring model class with I/O arguments style step method |

## Methods

See the methods of the base class `RTW.ModelCPPClass`, plus the following methods.

| | |
|---|---|
| getArgCategory | Get argument category for Simulink model port from model-specific C++ class interface |
| getArgName | Get argument name for Simulink model port from model-specific C++ class interface |
| getArgPosition | Get argument position for Simulink model port from model-specific C++ class interface |
| getArgQualifier | Get argument type qualifier for Simulink model port from model-specific C++ class interface |
| runValidation | Validate model-specific C++ class interface against Simulink model |
| setArgCategory | Set argument category for Simulink model port in model-specific C++ class interface |
| setArgName | Set argument name for Simulink model port in model-specific C++ class interface |
| setArgPosition | Set argument position for Simulink model port in model-specific C++ class interface |
| setArgQualifier | Set argument type qualifier for Simulink model port in model-specific C++ class interface |

## Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects (MATLAB).

## Alternatives

The **Configure C++ Class Interface** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Configure C++ class interface dialog box, where you can flexibly control the C++ class interfaces that are generated for your model. Once you validate and apply your changes, you can generate code based on your C++ class interface modifications. See “Customize C++ Class Interfaces Using Graphical Interfaces”.

# See Also

## Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

# RTW.ModelCPPArgsClass

**Class:** RTW.ModelCPPArgsClass
**Package:** RTW

Create C++ class interface object for configuring model class with I/O arguments style step method

## Syntax

```
obj = RTW.ModelCPPArgsClass
```

## Description

*obj* = RTW.ModelCPPArgsClass returns a handle, *obj*, to a newly created object of class RTW.ModelCPPArgsClass.

## Output Arguments

| | |
|---|---|
| *obj* | Handle to a newly created C++ class interface object for configuring a model class with an I/O arguments style step method. The object has not yet been configured or attached to an ERT-based Simulink model. |

## Alternatives

The **Configure C++ Class Interface** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Configure C++ class interface dialog box, where you can flexibly control the C++ class interfaces that are generated for your model. See “Customize C++ Class Interfaces Using Graphical Interfaces”.

## See Also

### Topics

"Customize C++ Class Interfaces Programmatically"
"Configure Step Method for Model Class"
"Customize Generated C++ Class Interfaces"

# RTW.ModelCPPClass class

**Package:** RTW

Control C++ class interfaces for models

## Description

The `ModelCPPClass` class is the base class for the classes `RTW.ModelCPPArgsClass` and `RTW.ModelCPPDefaultClass`, which provide objects that describe C++ class interfaces for models using either an I/O arguments style step method or a default style step method. Use the `attachToModel` method to attach a C++ class interface to a loaded ERT-based Simulink model.

## Construction

To access the methods of this class, use the constructor for either `RTW.ModelCPPArgsClass` or `RTW.ModelCPPDefaultClass`.

## Methods

| | |
|---|---|
| attachToModel | Attach model-specific C++ class interface to loaded ERT-based Simulink model |
| getClassName | Get class name from model-specific C++ class interface |
| getDefaultConf | Get default configuration information for model-specific C++ class interface from Simulink model |
| getNamespace | Get namespace from model-specific C++ class interface |
| getNumArgs | Get number of step method arguments from model-specific C++ class interface |
| getStepMethodName | Get step method name from model-specific C++ class interface |
| setClassName | Set class name in model-specific C++ class interface |
| setNamespace | Set namespace in model-specific C++ class interface |
| setStepMethodName | Set step method name in model-specific C++ class interface |

## Alternatives

The **Configure C++ Class Interface** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Configure C++ class interface dialog box, where you can flexibly control the C++ class interfaces that are generated for your model. Once you validate and apply your changes, you can generate code based on your C ++ class interface modifications. See “Customize C++ Class Interfaces Using Graphical Interfaces”.

## See Also

### Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

# RTW.ModelCPPDefaultClass class

**Package:** RTW
**Superclasses:**

Control C++ class interfaces for models using default model step method

## Description

The `ModelCPPDefaultClass` class provides objects that describe C++ class interfaces for models using a default model step method. Use the `attachToModel` method to attach a C++ class interface to a loaded ERT-based Simulink model.

## Construction

| | |
|---|---|
| RTW.ModelCPPDefaultClass | Create C++ class interface object for configuring model class with default model step method |

## Methods

See the methods of the base class `RTW.ModelCPPClass`, plus the following method.

| | |
|---|---|
| runValidation | Validate model-specific C++ class interface against Simulink model |

## Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects (MATLAB).

## Alternatives

The **Configure C++ Class Interface** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Configure C++ class interface dialog

box, where you can flexibly control the C++ class interfaces that are generated for your model. Once you validate and apply your changes, you can generate code based on your C ++ class interface modifications. See "Customize C++ Class Interfaces Using Graphical Interfaces".

## See Also

### Topics

"Customize C++ Class Interfaces Programmatically"
"Configure Step Method for Model Class"
"Customize Generated C++ Class Interfaces"

# RTW.ModelCPPDefaultClass

**Class:** RTW.ModelCPPDefaultClass
**Package:** RTW

Create C++ class interface object for configuring model class with default model step method

## Syntax

`obj = RTW.ModelCPPDefaultClass`

## Description

`obj = RTW.ModelCPPDefaultClass` returns a handle, `obj`, to a newly created object of class `RTW.ModelCPPDefaultClass`.

## Output Arguments

| | |
|---|---|
| *obj* | Handle to a newly created C++ class interface object for configuring a model class with a default model step method. The object has not yet been configured or attached to an ERT-based Simulink model. |

## Alternatives

The **Configure C++ Class Interface** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Configure C++ class interface dialog box, where you can flexibly control the C++ class interfaces that are generated for your model. See “Customize C++ Class Interfaces Using Graphical Interfaces”.

# See Also

## Topics

"Customize C++ Class Interfaces Programmatically"
"Configure Step Method for Model Class"
"Customize Generated C++ Class Interfaces"

# RTW.ModelSpecificCPrototype class

**Package:** RTW

Describe signatures of functions for model

## Description

A `ModelSpecificCPrototype` object describes the signatures of the step and initialization functions for a model. You must use this in conjunction with the `attachToModel` method.

## Construction

| | |
|---|---|
| RTW.ModelSpecificCPrototype | Create model-specific C prototype object |

## Methods

| | |
|---|---|
| addArgConf | Add argument configuration information for Simulink model port to model-specific C function prototype |
| attachToModel | Attach model-specific C function prototype to loaded ERT-based Simulink model |
| getArgCategory | Get argument category for Simulink model port from model-specific C function prototype |
| getArgName | Get argument name for Simulink model port from model-specific C function prototype |
| getArgPosition | Get argument position for Simulink model port from model-specific C function prototype |
| getArgQualifier | Get argument type qualifier for Simulink model port from model-specific C function prototype |
| getDefaultConf | Get default configuration information for model-specific C function prototype from Simulink model |
| getFunctionName | Get function name from model-specific C function prototype |
| getNumArgs | Get number of function arguments from model-specific C function prototype |
| getPreview | Get model-specific C function prototype code preview |
| runValidation | Validate model-specific C function prototype against Simulink model |
| setArgCategory | Set argument category for Simulink model port in model-specific C function prototype |
| setArgName | Set argument name for Simulink model port in model-specific C function prototype |
| setArgPosition | Set argument position for Simulink model port in model-specific C function prototype |
| setArgQualifier | Set argument type qualifier for Simulink model port in model-specific C function prototype |
| setFunctionName | Set function name in model-specific C function prototype |

## Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects (MATLAB).

## Examples

The code below creates a function control object, `a`, and uses it to add argument configuration information to the model.

```
% Open the rtwdemo_counter model and specify the System Target File
rtwdemo_counter
set_param(gcs,'SystemTargetFile','ert.tlc')

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

%% Add argument configuration information for Input and Output ports
addArgConf(a,'Input','Pointer','inputArg','const *')
addArgConf(a,'Output','Pointer','outputArg','none')

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

## Alternatives

You can create a function control object using the Model Interface dialog box.

## See Also

`RTW.ModelSpecificCPrototype.addArgConf`

### Topics

“Customize Generated C Function Interfaces”

# RTW.ModelSpecificCPrototype

**Class:** RTW.ModelSpecificCPrototype
**Package:** RTW

Create model-specific C prototype object

## Syntax

*obj* = RTW.ModelSpecificCPrototype

## Description

*obj* = RTW.ModelSpecificCPrototype creates a handle, *obj*, to an object of class RTW.ModelSpecificCPrototype.

## Output Arguments

| | |
|---|---|
| *obj* | Handle to model specific C prototype object. |

## Examples

Create a function control object, a, and use it to add argument configuration information to the model:

```
% Open the rtwdemo_counter model and specify the System Target File
rtwdemo_counter
set_param(gcs,'SystemTargetFile','ert.tlc')

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

%% Add argument configuration information for Input and Output ports
addArgConf(a,'Input','Pointer','inputArg','const *')
addArgConf(a,'Output','Pointer','outputArg','none')

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

## Alternatives

The **Configure Model Functions** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Model Interface dialog box, which provides you flexible control over the C function prototypes that are generated for your model. See “Customize Generated C Function Interfaces”.

## See Also

`RTW.ModelSpecificCPrototype.addArgConf`

### Topics

“Customize Generated C Function Interfaces”

# rtw.pil.RtIOStreamApplicationFramework

Configure target-side communications

## Description

Specify target-specific libraries and source files that are required to build the executable file. The libraries and source files must include the device drivers that implement the target-side of the `rtiostream` communications channel.

## Creation

### Description

`appFrameObj = rtw.pil.RtIOStreamApplicationFramework(componentArgs)` returns an object that provides access to an `RTW.BuildInfo` object containing PIL-specific files (including a PIL `main` function). `rtw.connectivity.MakefileBuilder` combines these files with the PIL component libraries to create the PIL application.

Make a subclass of `rtw.pil.RtIOStreamApplicationFramework`. In addition:

- Use the `addPILMain` method to specify a `main` function, which is required to build the PIL application.
- To the `RTW.BuildInfo` object, add data that is required for the implementation of the `rtiostream` target communications interface by using provided functions:
  - Source file names - `addSourceFiles`
  - Source file paths - `addSourcePaths`
  - Include file names - `addIncludeFiles`
  - Include file paths - `addIncludePaths`
  - Libraries - `addLinkObjects`
  - Preprocessor macro definitions - `addDefines`
  - Compiler options - `addCompileFlags`

- Linker options - `addLinkFlags`

## Object Functions

| Function | Description |
|---|---|
| `getComponentArgs` | `componentArgs = appFrameObj.getComponentArgs` returns the `rtw.connectivity.ComponentArgs` object associated with `appFrameObj`. |
| `getBuildInfo` | `buildInfo = appFrameObj.getBuildInfo` returns the `RTW.BuildInfo` object associated with `appFrameObj`. |
| `addPILMain` | To build the PIL application, a `main` function is required. Use this method to add one of the two provided files to the application framework.<br><br>To specify a `main` function that is adapted for on-target PIL and suitable for most PIL implementations, enter:<br><br>`appFrameObj.addPILMain('target');`<br><br>To specify a `main` function that is adapted for PIL on your development computer, enter:<br><br>`appFrameObj.addPILMain('host');`<br><br>Alternatively, you can specify your own `main` function:<br><br>`componentArgs = appFrameObj.getComponentArgs;`<br>`buildInfo = appFrameObj.getBuildInfo;`<br>`buildInfo.addSourcePaths(`*`pathToMyMainC`*`);`<br>`buildInfo.addSourceFiles(`*`myMainC`*`);` |

## Examples

### Using `rtw.pil.RtIOStreamApplicationFramework` in PIL Connectivity

For an example that shows how to use this object in setting up PIL connectivity, see:

- For MATLAB, “Processor-in-the-Loop Execution From Command Line”
- For Simulink, “Configure Processor-In-The-Loop (PIL) for a Custom Target”

## See Also

`rtiostream_wrapper` | `rtw.connectivity.ComponentArgs`

### Topics

“Create PIL Target Connectivity Configuration for MATLAB”
“Create PIL Target Connectivity Configuration for Simulink”
“Build Information Object” (Simulink Coder)

**Introduced in R2008b**

# run

**Class:** cgv.CGV
**Package:** cgv

Execute CGV object

## Syntax

`result = cgvObj.run()`

## Description

`result = cgvObj.run()` executes the model once for each input data that you added to the object. *`result`* is a boolean value that indicates whether the run completed without execution error. *`cgvObj`* is a handle to a `cgv.CGV` object.

After each execution of the model, the object captures and writes the following metadata to a file in the output folder:
`ErrorDetails` — If errors occur, the error information.
`status` — The execution status.
`ver` — Version information for MathWorks® products.
`hostname` — Name of computer.
`dateTime` — Date and time of execution.
`warnings` — If warnings occur, the warning messages.
`username` — Name of user.
`runtime` — The amount of time that lapsed for the execution.

## Tips

- Only call `run` once for each `cgv.CGV` object.
- The cgv.CGV methods that set up the object are ignored after a call to `run`. See the `cgv.CGV` for details.
- You can call `run` once without first calling `addInputData`. However, it is recommended that you first save the required data for execution to a MAT-file,

including the model inputs and parameters. Then use `cgv.CGV.addInputData` to pass the MAT-file to the CGV object before calling `run`.

- The `cgv.CGV` object supports callback functions that you can define and add to the `cgv.CGV` object. These callback functions are called during `cgv.CGV.run()` in the following order:

| Callback function | Add to object using... | `cgv.CGV.run()` executes callback function... |
| --- | --- | --- |
| HeaderReportFcn | `addHeaderReportFcn` | Before executing input data in `cgv.CGV` |
| PreExecReportFcn | `addPreExecReportFcn` | Before executing each input data file in `cgv.CGV` |
| PreExecFcn | `addPreExecFcn` | Before executing each input data file in `cgv.CGV` |
| PostExecReportFcn | `addPostExecReportFcn` | After executing each input data file in `cgv.CGV` |
| PostExecFcn | `addPostExecFcn` | After executing each input data file in `cgv.CGV` |
| TrailerReportFcn | `addTrailerReportFcn` | After the input data is executed in `cgv.CGV` |

## See Also

### Topics

"Verify Numerical Equivalence with CGV"

# runValidation

**Class:** RTW.ModelCPPArgsClass
**Package:** RTW

Validate model-specific C++ class interface against Simulink model

## Syntax

`[status, msg] = runValidation(obj)`

## Description

`[status, msg] = runValidation(obj)` runs a validation check of the specified model-specific C++ class interface against the ERT-based Simulink model to which it is attached.

Before calling this function, you must call either `attachToModel`, to attach a function prototype to a loaded model, or `RTW.getClassInterfaceSpecification`, to get the handle to a function prototype previously attached to a loaded model.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a model-specific C++ class interface control object, such as a handle previously returned by `obj = RTW.ModelCPPArgsClass` on page 1-534 or `obj = RTW.getClassInterfaceSpecification (modelName)`. |

## Output Arguments

| | |
|---|---|
| *status* | Boolean value; true for a valid configuration, false otherwise. |

| | |
|---|---|
| *msg* | If *status* is false, *msg* contains a character vector of information describing why the configuration is invalid. |

## Alternatives

To validate a C++ class interface in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. Click the **Validate** button to validate your current model step function configuration. The **Validation** pane displays status and an explanation of failures. For more information, see “Configure Step Method for Your Model Class”.

## See Also

### Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

# runValidation

**Class:** RTW.ModelCPPDefaultClass
**Package:** RTW

Validate model-specific C++ class interface against Simulink model

## Syntax

`[status, msg] = runValidation(obj)`

## Description

`[status, msg] = runValidation(obj)` runs a validation check of the specified model-specific C++ class interface against the ERT-based Simulink model to which it is attached.

Before calling this function, you must call either `attachToModel`, to attach a function prototype to a loaded model, or `RTW.getClassInterfaceSpecification`, to get the handle to a function prototype previously attached to a loaded model.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a model-specific C++ class interface control object, such as a handle previously returned by `obj = RTW.ModelCPPDefaultClass` on page 1-540 or `obj = RTW.getClassInterfaceSpecification (modelName)`. |

## Output Arguments

| | |
|---|---|
| *status* | Boolean value; true for a valid configuration, false otherwise. |

| | |
|---|---|
| *msg* | If *status* is false, *msg* contains a character vector of information describing why the configuration is invalid. |

## Alternatives

To validate a C++ class interface in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. Click the **Validate** button to validate your current model step function configuration. The **Validation** pane displays status and an explanation of failures. For more information, see “Configure Step Method for Your Model Class”.

## See Also

### Topics
“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

# runValidation

**Class:** RTW.ModelSpecificCPrototype
**Package:** RTW

Validate model-specific C function prototype against Simulink model

## Syntax

```
[status, msg] = runValidation(obj)
```

## Description

`[status, msg] = runValidation(obj)` runs a validation check of the specified model-specific C function prototype against the ERT-based Simulink model to which it is attached.

Before calling this function, you must call either `attachToModel`, to attach a function prototype to a loaded model, or `RTW.getFunctionSpecification`, to get the handle to a function prototype previously attached to a loaded model.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a model-specific C prototype function control object previously returned by `obj = RTW.ModelSpecificCPrototype` or `obj = RTW.getFunctionSpecification (modelName)`. |

## Output Arguments

| | |
|---|---|
| *status* | True for a valid configuration; false otherwise. |
| *msg* | If *status* is false, *msg* contains a character vector explaining why the configuration is invalid. |

## Alternatives

Click the **Validate** button in the Model Interface dialog box to run a validation check of the specified model-specific C function prototype against the ERT-based Simulink model to which it is attached. See “Configure C Step Function Arguments”.

## See Also

### Topics

“Customize Generated C Function Interfaces”

# setArgCategory

**Class:** RTW.ModelCPPArgsClass
**Package:** RTW

Set argument category for Simulink model port in model-specific C++ class interface

## Syntax

`setArgCategory(obj, portName, category)`

## Description

`setArgCategory(obj, portName, category)` sets the category — `'Value'`, `'Pointer'`, or `'Reference'` — of the argument corresponding to a specified Simulink model inport or outport in a specified model-specific C++ class interface.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a model-specific C++ class interface control object, such as a handle previously returned by `obj = RTW.ModelCPPArgsClass` on page 1-534 or `obj = RTW.getClassInterfaceSpecification (modelName)`. |
| *portName* | Character vector specifying the unqualified name of an inport or outport in your Simulink model. |
| *category* | Character vector specifying the argument category — `'Value'`, `'Pointer'`, or `'Reference'` — to be set for the specified Simulink model port. |

**Note** If you change the argument category for an outport from `'Pointer'` to `'Value'`, the change causes the argument to move to the first argument position when `attachToModel` or `runValidation` is called.

## Alternatives

To set argument categories in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments step method` view of this dialog box, click the **Get Default Configuration** button to display step method argument categories that you can examine and modify. For more information, see “Configure Step Method for Your Model Class”.

## See Also

### Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

# setArgCategory

**Class:** RTW.ModelSpecificCPrototype
**Package:** RTW

Set argument category for Simulink model port in model-specific C function prototype

## Syntax

`setArgCategory(obj, portName, category)`

## Description

`setArgCategory(obj, portName, category)` sets the category, `'Value'` or `'Pointer'`, of the argument corresponding to a specified Simulink model inport or outport in a specified model-specific C function prototype.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a model-specific C prototype function control object previously returned by `obj = RTW.ModelSpecificCPrototype` or `obj = RTW.getFunctionSpecification(modelName)`. |
| *portName* | Character vector specifying the unqualified name of an inport or outport in your Simulink model. |
| *category* | Character vector specifying the argument category, `'Value'` or `'Pointer'`, that you set for the specified Simulink model port. |

**Note** If you change the argument category for an outport from `'Pointer'` to `'Value'`, it causes the argument to move to the first argument position when you call `RTW.ModelSpecificCPrototype.attachToModel` or `RTW.ModelSpecificCPrototype.runValidation`.

## Alternatives

Use the **Step function arguments** table in the Model Interface dialog box to specify argument categories. See “Configure C Step Function Arguments”.

## See Also

### Topics

“Customize Generated C Function Interfaces”

# setArgName

**Class:** RTW.ModelCPPArgsClass
**Package:** RTW

Set argument name for Simulink model port in model-specific C++ class interface

## Syntax

`setArgName(obj, portName, argName)`

## Description

`setArgName(obj, portName, argName)` sets the argument name that corresponds to a specified Simulink model inport or outport in a specified model-specific C++ class interface.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a model-specific C++ class interface control object, such as a handle previously returned by `obj = RTW.ModelCPPArgsClass` on page 1-534 or `obj = RTW.getClassInterfaceSpecification (modelName)`. |
| *portName* | Character vector specifying the name of an inport or outport in your Simulink model. |
| *argName* | Character vector specifying the argument name to set for the specified Simulink model port. The argument must be a valid C identifier. |

## Alternatives

To set argument names in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface**

button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments step method` view of this dialog box, click the **Get Default Configuration** button to display step method argument names that you can examine and modify. For more information, see “Configure Step Method for Your Model Class”.

# See Also

## Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

# setArgName

**Class:** RTW.ModelSpecificCPrototype
**Package:** RTW

Set argument name for Simulink model port in model-specific C function prototype

## Syntax

`setArgName(obj, portName, argName)`

## Description

`setArgName(obj, portName, argName)` sets the argument name corresponding to a specified Simulink model inport or outport in a specified model-specific C function prototype.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a model-specific C prototype function control object previously returned by `obj = RTW.ModelSpecificCPrototype` or `obj = RTW.getFunctionSpecification (modelName)`. |
| *portName* | Character vector specifying the name of an inport or outport in your Simulink model. |
| *argName* | Character vector specifying the argument name to set for the specified Simulink model port. The argument must be a valid C identifier. |

## Alternatives

Use the **Step function arguments** table in the Model Interface dialog box to specify argument names. See “Configure C Step Function Arguments”.

## See Also

### Topics

"Customize Generated C Function Interfaces"

# setArgPosition

**Class:** RTW.ModelCPPArgsClass
**Package:** RTW

Set argument position for Simulink model port in model-specific C++ class interface

## Syntax

`setArgPosition(obj, portName, position)`

## Description

`setArgPosition(obj, portName, position)` sets the position — 1 for first, 2 for second, etc. — of the argument that corresponds to a specified Simulink model inport or outport in a specified model-specific C++ class interface. The specified argument is then moved to the specified position, and other arguments shifted by one position accordingly.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a model-specific C++ class interface control object, such as a handle previously returned by `obj = RTW.ModelCPPArgsClass` on page 1-534 or `obj = RTW.getClassInterfaceSpecification (modelName)`. |
| *portName* | Character vector specifying the name of an inport or outport in your Simulink model. |
| *position* | Integer specifying the argument position — 1 for first, 2 for second, etc. — to be set for the specified Simulink model port. The value must be greater than or equal to 1 and less than or equal to the number of function arguments. |

## Alternatives

To set argument positions in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments step method` view of this dialog box, click the **Get Default Configuration** button to display step method argument positions that you can examine and modify. For more information, see "Configure Step Method for Your Model Class".

## See Also

### Topics

"Customize C++ Class Interfaces Programmatically"
"Configure Step Method for Model Class"
"Customize Generated C++ Class Interfaces"

# setArgPosition

**Class:** RTW.ModelSpecificCPrototype
**Package:** RTW

Set argument position for Simulink model port in model-specific C function prototype

## Syntax

`setArgPosition(obj, portName, position)`

## Description

`setArgPosition(obj, portName, position)` sets the position — 1 for first, 2 for second, etc. — of the argument corresponding to a specified Simulink model inport or outport in a specified model-specific C function prototype. The specified argument moves to the specified position, and other arguments shift by one position accordingly.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a model-specific C prototype function control object previously returned by `obj = RTW.ModelSpecificCPrototype` or `obj = RTW.getFunctionSpecification (modelName)`. |
| *portName* | Character vector specifying the name of an inport or outport in your Simulink model. |
| *position* | Integer specifying the argument position — 1 for first, 2 for second, etc. — to be set for the specified Simulink model port. The value must be greater than or equal to 1 and less than or equal to the number of function arguments. |

## Alternatives

Use the **Step function arguments** table in the Model Interface dialog box to specify argument position. See “Configure C Step Function Arguments”.

## See Also

### Topics

“Customize Generated C Function Interfaces”

# setArgQualifier

**Class:** RTW.ModelCPPArgsClass
**Package:** RTW

Set argument type qualifier for Simulink model port in model-specific C++ class interface

## Syntax

```
setArgQualifier(obj, portName, qualifier)
```

## Description

`setArgQualifier(obj, portName, qualifier)` sets the type qualifier — `'none'`, `'const'`, `'const *'`, `'const * const'`, or `'const &'` — of the argument that corresponds to a specified Simulink model inport or outport in a specified model-specific C++ class interface.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a model-specific C++ class interface control object, such as a handle previously returned by `obj = RTW.ModelCPPArgsClass` on page 1-534 or `obj = RTW.getClassInterfaceSpecification (modelName)`. |
| *portName* | Character vector specifying the name of an inport or outport in your Simulink model. |
| *qualifier* | Character vector specifying the argument type qualifier — `'none'`, `'const'`, `'const *'`, `'const * const'`, or `'const &'` — to be set for the specified Simulink model port. |

## Alternatives

To set argument qualifiers in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface**

button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments step method` view of this dialog box, click the **Get Default Configuration** button to display step method argument qualifiers that you can examine and modify. For more information, see "Configure Step Method for Your Model Class".

## See Also

### Topics

"Customize C++ Class Interfaces Programmatically"
"Configure Step Method for Model Class"
"Customize Generated C++ Class Interfaces"

# setArgQualifier

**Class:** RTW.ModelSpecificCPrototype
**Package:** RTW

Set argument type qualifier for Simulink model port in model-specific C function prototype

## Syntax

`setArgQualifier(`*`obj`*`, `*`portName`*`, `*`qualifier`*`)`

## Description

`setArgQualifier(`*`obj`*`, `*`portName`*`, `*`qualifier`*`)` sets the type qualifier — `'none'`, `'const'`, `'const *'`, or `'const * const'`— of the argument corresponding to a specified Simulink model inport or outport in a specified model-specific C function prototype.

## Input Arguments

| | |
|---|---|
| *`obj`* | Handle to a model-specific C prototype function control object previously returned by *`obj`*` = RTW.ModelSpecificCPrototype` or *`obj`*` = RTW.getFunctionSpecification (`*`modelName`*`)`. |
| *`portName`* | Character vector specifying the name of an inport or outport in your Simulink model. |
| *`qualifier`* | Character vector specifying the argument type qualifier — `'none'`, `'const'`, `'const *'`, or `'const * const'`— to be set for the specified Simulink model port. |

## Alternatives

Use the **Step function arguments** table in the Model Interface dialog box to specify argument qualifiers. See “Configure C Step Function Arguments”.

## See Also

### Topics

“Customize Generated C Function Interfaces”

# setClassName

**Class:** RTW.ModelCPPClass
**Package:** RTW

Set class name in model-specific C++ class interface

## Syntax

`setClassName(obj, clsName)`

## Description

`setClassName(obj, clsName)` sets the class name in the specified model-specific C++ class interface.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a model-specific C++ class interface control object, such as a handle previously returned by `obj = RTW.ModelCPPArgsClass` on page 1-534, `obj = RTW.ModelCPPDefaultClass` on page 1-540, or `obj = RTW.getClassInterfaceSpecification (modelName)`. |
| *clsName* | Character vector specifying a new name for the class described by the specified model-specific C++ class interface. The argument must be a valid C/C++ identifier. |

## Alternatives

To set the model class name in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments`

`step method` view of this dialog box, click the **Get Default Configuration** button to display the model class name, which you can examine and modify. In the `Default step method` view, you can examine and modify the model class name without having to click a button. For more information, see “Configure Step Method for Your Model Class”.

## See Also

### Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

# setFunctionName

**Class:** RTW.ModelSpecificCPrototype
**Package:** RTW

Set function name in model-specific C function prototype

## Syntax

`setFunctionName(obj, fcnName, fcnType)`

## Description

`setFunctionName(obj, fcnName, fcnType)` sets the step or initialization function name in the specified function control object.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a model-specific C prototype function control object previously returned by `obj = RTW.ModelSpecificCPrototype` or `obj = RTW.getFunctionSpecification(modelName)`. |
| *fcnName* | Character vector specifying a new name for the function described by the function control object. The argument must be a valid C identifier. |
| *fcnType* | Optional. Character vector specifying which function to name. Valid options are `'step'` and `'init'`. If *fcnType* is not specified, sets the step function name. |

## Alternatives

Use the **Initialize function name** and **Step function name** fields in the Model Interface dialog box to specify function names. See “Configure C Step Function Arguments”.

## See Also

### Topics

“Customize Generated C Function Interfaces”

# setMode

**Class:** cgv.CGV
**Package:** cgv

Specify mode of execution

## Syntax

```
cgvObj.setMode(connectivity)
```

## Description

`cgvObj.setMode(connectivity)` specifies the mode of execution for the `cgv.CGV` object, *`cgvObj`*. The default value for the execution mode is set to either `normal` or `sim`.

## Input Arguments

**`connectivity`**

Specify mode of execution

| Value | Description |
|---|---|
| `sim` or `normal` (default) | Mode of execution is normal simulation. |
| `sil` | Mode of execution is SIL. |
| `pil` | Mode of execution is PIL. |

## Examples

After running a `cgv.CGV` object, copy the object. Before rerunning the object, call `setMode` to change the execution mode to `sil` for an existing `cgv.CGV` object.

```
cgvModel = 'rtwdemo_cgv';
cgvObj1 = cgv.CGV(cgvModel, 'connectivity', 'sim');
```

```
cgvObj1.run();
cgvObj2 = cgvObj1.copySetup()
cgvObj2.setMode('sil');
cgvObj2.run();
```

## See Also

`cgv.CGV.copySetup` | `cgv.CGV.run`

### Topics

"Verify Numerical Equivalence with CGV"

# setNameSpace

Set namespace for C++ function entry in code replacement table

## Syntax

`setNameSpace(hEntry,nameSpace)`

## Description

`setNameSpace(hEntry,nameSpace)` specifies the namespace for a C++ function entry in a code replacement table.

During code generation, if the function entry is matched, the software emits the namespace in the generated function code (for example, `std::sin(tfl_cpp_U.In1)`).

If you created the function entry by using *`hEntry`* `= RTW.TflCFunctionEntry` or *`hEntry`* = *`MyCustomFunctionEntry`* (did not use `registerCPPFunctionEntry`), before calling the `setNameSpace` function, enable C++ support for the function entry by calling the `enableCPP` function.

## Examples

### Set Namespace for Implementation Function

This example shows how to use the `setNameSpace` function to set the namespace for the `sin` implementation function to `std`.

```
fcn_entry = RTW.TflCFunctionEntry;
fcn_entry.setTflCFunctionEntryParameters( ...
                               'Key',                      'sin', ...
                               'Priority',                 100, ...
                               'ImplementationName',       'sin', ...
                               'ImplementationHeaderFile', 'cmath' );
```

```
fcn_entry.enableCPP();
fcn_entry.setNameSpace('std');
```

## Input Arguments

**`hEntry` — Handle to a code replacement function entry**
handle

The *`hEntry`* is a handle to a code replacement function entry previously returned by one of the following:

- *`hEntry`* `= RTW.TflCFunctionEntry`
- *`hEntry = MyCustomFunctionEntry`*, where *`MyCustomFunctionEntry`* is a class derived from `RTW.TflCFunctionEntry`
- A call to the `registerCPPFunctionEntry` function

Example: `fcn_entry`

**`nameSpace` — Specifies the namespace in which the implementation function for the C++ function entry is defined**
character vector

Example: `'std'`

## See Also

`enableCPP` | `registerCPPFunctionEntry`

### Topics

"Define Code Replacement Mappings"

**Introduced in R2010a**

# setNamespace

**Class:** RTW.ModelCPPClass
**Package:** RTW

Set namespace in model-specific C++ class interface

## Syntax

`setNamespace(obj, nsName)`

## Description

`setNamespace(obj, nsName)` sets the namespace in the specified model-specific C++ class interface.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a model-specific C++ class interface control object, such as a handle previously returned by `obj = RTW.ModelCPPArgsClass` on page 1-534, `obj = RTW.ModelCPPDefaultClass` on page 1-540, or `obj = RTW.getClassInterfaceSpecification (modelName)`. |
| *nsName* | Character vector specifying a namespace for the class described by the specified model-specific C++ class interface. The argument must be a valid C/C++ identifier. |

## Alternatives

To set the model namespace in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, where you can display and configure the namespace for your model class. In the `I/O arguments step`

`method` view of this dialog box, click the **Get Default Configuration** button to display the model namespace, which you can examine and modify. In the `Default step method` view, you can examine and modify the model namespace without having to click a button. For more information, see “Configure Step Method for Your Model Class”.

# See Also

## Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

# setObjectiveName

**Class:** rtw.codegenObjectives.Objective
**Package:** rtw.codegenObjectives

Specify objective name

## Syntax

`setObjectiveName(obj, objName)`

## Description

`setObjectiveName(obj, objName)` specifies a name for the objective. The Configuration Set Objectives dialog box displays the name of the objective.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a code generation objective object previously created. |
| *objName* | Optional character vector that indicates the name of the objective. If you do not specify an objective name, the Configuration Set Objectives dialog box displays the objective ID for the objective name. |

## Examples

Name the objective Reduce RAM Example:

```
setObjectiveName(obj, 'Reduce RAM Example');
```

## See Also

### Topics
"Create Custom Code Generation Objectives"

# setOutputDir

**Class:** cgv.CGV
**Package:** cgv

Specify folder

## Syntax

```
cgvObj.setOutputDir('path')
cgvObj.setOutputDir('path', 'overwrite', 'on')
```

## Description

*`cgvObj`*`.setOutputDir('`*`path`*`')` is an optional method that specifies a location where the object writes the output and metadata files for execution. *`cgvObj`* is a handle to a `cgv.CGV` object. *`path`* is the absolute or relative path to the folder. If the path does not exist, the object attempts to create the folder. If you do not call `setOutputDir`, the object uses the current working folder.

*`cgvObj`*`.setOutputDir('`*`path`*`', 'overwrite', '`*`on`*`')` includes the property and value pair to allow read-only files in the working directory to be overwritten. The default value for `'overwrite'` is `'`*`off`*`'`.

## See Also

### Topics

"Verify Numerical Equivalence with CGV"

# setOutputFile

**Class:** cgv.CGV
**Package:** cgv

Specify output data file name

## Syntax

*cgvObj*.setOutputFile(*InputIndex*,*OutputFile*)

## Description

*cgvObj*.setOutputFile(*InputIndex*,*OutputFile*) is an optional method that changes the default file name for the output data. *cgvObj* is a handle to a cgv.CGV object. *InputIndex* is a unique numeric identifier that specifies which output data to write to the file. The *InputIndex* is associated with specific input data.*OutputFile* is the name of the file, with or without the .mat extension.

## See Also

### Topics

"Verify Numerical Equivalence with CGV"

# setReservedIdentifiers

Register reserved identifiers to associate with code replacement library

## Syntax

setReservedIdentifiers(hTable,ids)

## Description

setReservedIdentifiers(hTable,ids) registers reserved identifier structures in a code replacement table.

In a code replacement table, the code generator registers each function implementation name defined by a table entry as a reserved identifier. You can register additional reserved identifiers for the table on a per-header-file basis. Providing additional reserved identifiers can help prevent duplicate symbols and other identifier-related compile and link issues.

The setReservedIdentifiers function lets you register up to four reserved identifier structures in a code replacement table. One set of reserved identifiers can be associated with a code replacement library, while the other three (if present) must be associated with libraries named ANSI_C, ISO_C, ISO_C++, or GNU.

For information about generating a list of reserved identifiers for the code replacement library that you use to generate code, see “Reserved Identifiers and Code Replacement”.

## Examples

### Register Reserved Identifier Structures

This example shows how to use the setReservedIdentifiers function to register four reserved identifier structures, for 'ANSI_C', 'ISO_C','ISO_C++', and 'My Custom CRL', respectively.

```
hLib = RTW.TflTable;

% Create and register CRL entries here

.
.
.

% Create and register reserved identifiers
d{1}.LibraryName = 'ANSI_C';
d{1}.HeaderInfos{1}.HeaderName = 'math.h';
d{1}.HeaderInfos{1}.ReservedIds = {'a', 'b'};
d{1}.HeaderInfos{2}.HeaderName = 'foo.h';
d{1}.HeaderInfos{2}.ReservedIds = {'c', 'd'};

d{2}.LibraryName = 'ISO_C';
d{2}.HeaderInfos{1}.HeaderName = 'math.h';
d{2}.HeaderInfos{1}.ReservedIds = {'a', 'b'};
d{2}.HeaderInfos{2}.HeaderName = 'foo.h';
d{2}.HeaderInfos{2}.ReservedIds = {'c', 'd'};

d{3}.LibraryName = 'ISO_C++';
d{3}.HeaderInfos{1}.HeaderName = 'math.h';
d{3}.HeaderInfos{1}.ReservedIds = {'a', 'b'};
d{3}.HeaderInfos{2}.HeaderName = 'foo.h';
d{3}.HeaderInfos{2}.ReservedIds = {'c', 'd'};

d{4}.LibraryName = 'My Custom CRL';
d{4}.HeaderInfos{1}.HeaderName = 'my_math_lib.h';
d{4}.HeaderInfos{1}.ReservedIds = {'y1', 'u1'};
d{4}.HeaderInfos{2}.HeaderName = 'my_oper_lib.h';
d{4}.HeaderInfos{2}.ReservedIds = {'foo', 'bar'};

setReservedIdentifiers(hLib, d);
```

## Input Arguments

**hTable — Handle to a code replacement table**

handle

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

Example: `hLib`

**`ids` — Specifies reserved keywords to register for library**
structure

The *`ids`* is a structure specifying reserved keywords to be registered for a library. The structure must contain:

- `LibraryName` element, a character vector that specifies `'ANSI_C'`, `'ISO_C'`, `'ISO_C++'`, `'GNU'`.
- `HeaderInfos` element, a structure or cell array of structures containing:
  - `HeaderName` element, a character vector that specifies the header file in which the identifiers are declared.
  - `ReservedIds` element, a cell array of character vectors that specifies the names of the identifiers to be registered as reserved keywords.

Example: `d`

# See Also

## Topics
"Reserved Identifiers and Code Replacement"

**Introduced in R2008a**

# setStepMethodName

**Class:** RTW.ModelCPPClass
**Package:** RTW

Set step method name in model-specific C++ class interface

## Syntax

`setStepMethodName(`*`obj`*`, `*`fcnName`*`)`

## Description

`setStepMethodName(`*`obj`*`, `*`fcnName`*`)` sets the step method name in the specified model-specific C++ class interface.

## Input Arguments

| | |
|---|---|
| *obj* | Handle to a model-specific C++ class interface control object, such as a handle previously returned by *obj* = `RTW.ModelCPPArgsClass` on page 1-534, *obj* = `RTW.ModelCPPDefaultClass` on page 1-540, or *obj* = `RTW.getClassInterfaceSpecification` (*modelName*). |
| *fcnName* | Character vector specifying a new name for the step method described by the specified model-specific C++ class interface. The argument must be a valid C/C++ identifier. |

## Alternatives

To set the step method name in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments`

`step method` view of this dialog box, click the **Get Default Configuration** button to display the step method name, which you can examine and modify. In the `Default step method` view, you can examine and modify the step method name without having to click a button. For more information, see “Configure Step Method for Your Model Class”.

## See Also

### Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

# setTflCFunctionEntryParameters

Set specified parameters for function entry in code replacement table

## Syntax

`setTflCFunctionEntryParameters(hEntry,varargin)`

## Description

`setTflCFunctionEntryParameters(hEntry,varargin)` sets specified parameters for a function entry in a code replacement table.

## Examples

### Specify Parameters for Function Entry

This example shows how to use the `setTflCFunctionEntryParameters` function to set specified parameters for a code replacement function entry for `sqrt`.

```
fcn_entry = RTW.TflCFunctionEntry;
fcn_entry.setTflCFunctionEntryParameters( ...
                                'Key',                      'sqrt', ...
                                'Priority',                 100, ...
                                'ImplementationName',       'sqrt', ...
                                'ImplementationHeaderFile', '<math.h>' );
```

## Input Arguments

**`hEntry` — Handle to a code replacement function entry**
handle

The *hEntry* is a handle to a code replacement function entry previously returned by *hEntry* = RTW.TflCFunctionEntry or *hEntry* = *MyCustomFunctionEntry*, where *MyCustomFunctionEntry* is a class derived from RTW.TflCFunctionEntry.

Example: fcn_entry

**varargin — Name-value pairs of arguments for function entry**
name-value pairs

Example: 'Key','sqrt'

## Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'Key','sqrt'

**AcceptExprInput — Selects whether implementation function accepts expression inputs**
true | false

The *AcceptExprInput* value flags the code generator that the implementation function described by this entry accepts expression inputs. The default value is true if ImplType equals FCN_IMPL_FUNCT and false if ImplType equals FCN_IMPL_MACRO.

If the value is true, expression inputs are integrated into the generated code in a form similar to this form:

```
rtY.Out1 = mySin(rtU.In1 + rtU.In2);
```

If the value is false, a temporary variable is generated for the expression input:

```
real_T rtb_Sum;

rtb_Sum = rtU.In1 + rtU.In2;
rtY.Out1 = mySin(rtb_Sum);
```

Example: 'AcceptExprInput',true

**AdditionalHeaderFiles — Specifies additional header files for table entry**
{} (default) | array of character vectors

The *AdditionalHeaderFiles* value specifies additional header files for a code replacement table entry. The vectors can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalHeaderFiles',{}`

**`AdditionalIncludePaths` — Specifies additional include paths for table entry**
`{}` (default) | array of character vectors

The *AdditionalIncludePaths* value specifies the full path of additional include paths for a code replacement entry. The character vectors can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalIncludePaths',{}`

**`AdditionalLinkObjs` — Specifies additional link objects for table entry**
`{}` (default) | array of character vectors

The *AdditionalLinkObjs* value specifies additional link objects for a code replacement table entry. The character vectors can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalLinkObjs',{}`

**`AdditionalLinkObjsPaths` — Specifying additional link object paths for table entry**
`{}` (default) | array of character vectors

The *AdditionalLinkObjsPaths* value specifies the full path of additional link object paths for a code replacement entry. The character vectors can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector. The default is `{}`.

Example: `'AdditionalLinkObjsPaths',{}`

**`AdditionalSourceFiles` — Specifies additional source files for table entry**
`{}` (default) | array of character vectors

The *AdditionalSourceFiles* value specifies additional source files for a code replacement table entry. The character vectors can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalSourceFiles',{}`

**AdditionalSourcePaths — Specifies additional source paths for table entry**
`{}` (default) | array of character vectors

The *AdditionalSourcePaths* value specifies the full path of additional source paths for a code replacement entry. The character vectors can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalSourcePaths',{}`

**AdditionalCompileFlags — Specifies additional compiler flags for table entry**
`{}` (default) | array of character vectors

The *AdditionalCompileFlags* value specifies additional flags required to compile the source files defined for a code replacement table entry. The default is `{}`.

Example: `'AdditionalCompileFlags',{}`

**AdditionalLinkFlags — Specifies additional linker flags for table entry**
`{}` (default) | array of character vectors

The *AdditionalLinkFlags* value specifies additional flags required to link the compiled files for a code replacement table entry.

Example: `'AdditionalLinkFlags',{}`

**ArrayLayout — Specifies layout of array storage for table entry**
`'COLUMN_MAJOR'` (default) | `'ROW_MAJOR'` | `'COLUMN_AND_ROW'`

The *ArrayLayout* value specifies the order of array elements in memory supported by the replacement implementation. By default, the replacement implementation supports column-major data layout. For `ROW-MAJOR`, the replacement implementation supports row-major data layout. For `COLUMN_AND_ROW`, the replacement implementation supports column-major and row-major data layouts.

Example: `'ArrayLayout','ROW_MAJOR'`

### `EntryInfoAlgorithm` — Specifies computation or approximation method to match for table entry

`'RTW_DEFAULT'` | `'RTW_NEWTON_RAPHSON'` | `'RTW_CORDIC'` | `'RTW_UNSPECIFIED'`

The *`EntryInfoAlgorithm`* value specifies a computation or approximation method, configured for the specified math function, that must be matched in order for function replacement to occur. Code replacement libraries support function replacement based on computation or approximation method for the math functions `rSqrt`, `sin`, `cos`, and `sincos`. The valid arguments for each supported function are listed in the table.

| Function | Argument | Meaning |
|---|---|---|
| `rSqrt` | `RTW_DEFAULT` | Match the default computation method, `Exact` |
| | `RTW_NEWTON_RAPHSON` | Match the `Newton-Raphson` computation method |
| | `RTW_UNSPECIFIED` | Match a computation method |
| `sin`<br>`cos`<br>`sincos` | `RTW_CORDIC` | Match the `CORDIC` approximation method |
| | `RTW_DEFAULT` | Match the default approximation method, `None` |
| | `RTW_UNSPECIFIED` | Match an approximation method |

Example: `'EntryInfoAlgorithm','RTW_DEFAULT'`

### `GenCallback` — Specifies callback that follows code generation

`''` (default) | `'RTW.copyFileToBuildDir'`

The *`GenCallback`* specifies a callback that follows code generation. If you specify `'RTW.copyFileToBuildDir'`, and if this function entry is matched and used, the code generator calls function `RTW.copyFileToBuildDir` after code generation. This callback function copies additional header, source, or object files that you have specified for this function entry to the build folder. For more information, see “Specify Build Information for Replacement Code”.

Example: `'GenCallback',''`

### `ImplementationHeaderFile` — Specifies the name of the header file that declares the implementation function

`{}` (default) | character vector

The *ImplementationHeaderFile* value specifies the name of the header file that declares the implementation function, for example, `'<math.h>'`. The character vector can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'ImplementationHeaderFile',{}`

**ImplementationHeaderPath — Specifies path to implementation header file**
`{}` (default) | character vector

The *ImplementationHeaderPath* value specifies the full path to the implementation header file. The character vector can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'ImplementationHeaderPath',{}`

**ImplementationName — Specifies name of implementation function**
`''` (default) | character vector

The *ImplementationName* value specifies the name of the implementation function, for example, `'sqrt'`, which can match or differ from the `Key` name.

Example: `'ImplementationName',''`

**ImplementationSourceFile — Specifies name of implementation source file**
`''` (default) | character vector

The *ImplementationSourceFile* value specifies the name of the implementation source file. The character vector can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'ImplementationSourceFile',''`

**ImplementationSourcePath — Specifies path to implementation source file**
`''` (default) | character vector

The *ImplementationSourcePath* value specifies the full path to the implementation source file. The character vector can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'ImplementationSourcePath',''`

**`ImplType` — Specifies the type of entry**
`'FCN_IMPL_FUNCT'` (default) | `'FCN_IMPL_MACRO'`

Use `FCN_IMPL_FUNCT` for function or `FCN_IMPL_MACRO` for macro.

Example: `'ImplType','FCN_IMPL_FUNCT'`

**Key — Specifies name of function to replace**
character vector

The Key value specifies the name of the function to replace. The name must match a function name listed in “Code You Can Replace from MATLAB Code” or“Code You Can Replace From Simulink Models”.

Example: `'Key','sqrt'`

**`Priority` — Specifies the search priority for function entry**
`100` (default) | integer 0..100

The *`Priority`* value specifies the search priority for the function entry, relative to other entries of the same function name and conceptual argument list within this table. Highest priority is 0, and lowest priority is 100. If the table provides two implementations for a function, the implementation with the higher priority shadows the one with the lower priority.

Example: `'Priority',100`

**RoundingModes — Specifying rounding modes supported by implementation function**
`'RTW_ROUND_UNSPECIFIED'` (default) | `'RTW_ROUND_FLOOR'` | `'RTW_ROUND_CEILING'` | `'RTW_ROUND_ZERO'` | `'RTW_ROUND_NEAREST'` | `'RTW_ROUND_NEAREST_ML'` | `'RTW_ROUND_SIMPLEST'` | `'RTW_ROUND_CONV'` | array of character vectors

The *`RoundingModes`* value specifies one or more rounding modes supported by the implementation function.

Example: `'RoundingModes',{'RTW_ROUND_UNSPECIFIED'}`

**`SaturationMode` — Specifying saturation mode supported by implementation function**
`'RTW_SATURATE_UNSPECIFIED'` (default) | `'RTW_SATURATE_ON_OVERFLOW'` | `'RTW_WRAP_ON_OVERFLOW'`

The *`SaturationMode`* value specifies the saturation mode supported by the implementation function.

Example: `'SaturationMode','RTW_SATURATE_UNSPECIFIED'`

**`SideEffects` — Specifies whether to attempt to optimize away the implementation function**
`false` (default) | `true`

The *`SideEffects`* value flags the code generator not to optimize away the implementation function described by this entry. This parameter applies to implementation functions that return `void` but are not to be optimized away, such as a `memcpy` implementation or an implementation function that accesses global memory values. For those implementation functions only, you must include this parameter and specify the value `true`.

Example: `'SideEffects',false`

**`StoreFcnReturnInLocalVar` — Specifies whether to store the implementation function regardless expression folding settings**
`false` (default) | `true`

The *`StoreFcnReturnInLocalVar`* value flags the code generator that the return value of the implementation function described by this entry must be stored in a local variable regardless of other expression folding settings. If the value is `false`, other expression folding settings determine whether the return value is folded. Storing function returns in a local variable can increase the clarity of generated code. This example shows code generated with expression folding:

```
void sw_step(void)
{
    if (ssub(sadd(sw_U.In1, sw_U.In2), sw_U.In3) <=
        smul(ssub(sw_U.In4, sw_U.In5),sw_U.In6)) {
      sw_Y.Out1 = sw_U.In7;
    } else {
       sw_Y.Out1 = sw_U.In8;
    }
}
```

With `StoreFcnReturnInLocalVar` set to `true`, the generated code is potentially easier to understand and debug:

```
void sw_step(void)
{
    real32_T rtb_Switch;
    real32_T hoistedExpr;
    ......
    rtb_Switch = sadd(sw_U.In1, sw_U.In2);
    rtb_Switch = ssub(rtb_Switch, sw_U.In3);
    hoistedExpr = ssub(sw_U.In4, sw_U.In5);
    hoistedExpr = smul(hoistedExpr, sw_U.In6);
    if (rtb_Switch <= hoistedExpr) {
       sw_Y.Out1 = sw_U.In7;
    } else {
       sw_Y.Out1 = sw_U.In8;
    }
}
```

Example: `'StoreFcnReturnInLocalVar',false`

# See Also

`addAdditionalHeaderFile` | `addAdditionalIncludePath` | `addAdditionalLinkObj` | `addAdditionalLinkObjPath` | `addAdditionalSourceFile` | `addAdditionalSourcepath`

## Topics

"Specify Build Information for Replacement Code"
"Define Code Replacement Mappings"
"Code You Can Replace from MATLAB Code"
"Code You Can Replace From Simulink Models"

**Introduced in R2007b**

# setTflCOperationEntryParameters

Set specified parameters for operator entry in code replacement table

## Syntax

`setTflCOperationEntryParameters(hEntry,varargin)`

## Description

`setTflCOperationEntryParameters(hEntry,varargin)` sets specified parameters for an operator entry in a code replacement table.

## Examples

### Set Parameters for Addition Operator Entry

This example shows how to use the `setTflCOperationEntryParameters` function to set parameters for a code replacement operator entry for `uint8` addition that matches a cast-after-sum algorithm.

```
op_entry = RTW.TflCOperationEntry;
op_entry.setTflCOperationEntryParameters( ...
    'Key',                        'RTW_OP_ADD', ...
    'EntryInfoAlgorithm',         'RTW_CAST_AFTER_OP', ...
    'Priority',                   90, ...
    'SaturationMode',             'RTW_SATURATE_UNSPECIFIED', ...
    'RoundingModes',              {'RTW_ROUND_UNSPECIFIED'}, ...
    'ImplementationName',         'u8_add_u8_u8', ...
    'ImplementationHeaderFile', 'u8_add_u8_u8.h', ...
    'ImplementationSourceFile', 'u8_add_u8_u8.c');
```

### Set Parameters for Fixed-Point Division Operator Entry

This example shows how to use the `setTflCOperationEntryParameters` function to set parameters for a code replacement operator entry for fixed-point `int16` division. The table entry specifies a net scaling between the operator inputs and output to map a range of slope and bias values to a replacement function.

```
op_entry = RTW.TflCOperationEntryGenerator_NetSlope;
op_entry.setTflCOperationEntryParameters( ...
    'Key',                         'RTW_OP_DIV', ...
    'Priority',                    90, ...
    'SaturationMode',              'RTW_WRAP_ON_OVERFLOW', ...
    'RoundingModes',               {'RTW_ROUND_CEILING'}, ...
    'NetSlopeAdjustmentFactor',      1.0, ...
    'NetFixedExponent',              0.0, ...
    'ImplementationName',          's16_div_s16_s16', ...
    'ImplementationHeaderFile',    's16_div_s16_s16.h', ...
    'ImplementationSourceFile',    's16_div_s16_s16.c' );
```

### Set Parameters for Fixed-Point Addition Operator Entry

This example shows how to use the `setTflCOperationEntryParameters` function to set parameters for a code replacement operator entry for fixed-point `uint16` addition that matches a cast-after-sum algorithm. The table entry specifies equal slope and zero net bias across operator inputs and output to map relative slope and bias values (rather than a specific slope and bias combination) to a replacement function.

```
op_entry = RTW.TflCOperationEntryGenerator;
op_entry.setTflCOperationEntryParameters( ...
    'Key',                        'RTW_OP_ADD', ...
    'EntryInfoAlgorithm',         'RTW_CAST_AFTER_OP', ...
    'Priority',                    90, ...
    'SaturationMode',              'RTW_WRAP_ON_OVERFLOW', ...
    'RoundingModes',               {'RTW_ROUND_UNSPECIFIED'}, ...
    'SlopesMustBeTheSame',         true, ...
    'MustHaveZeroNetBias',         true, ...
    'ImplementationName',          'u16_add_SameSlopeZeroBias', ...
```

```
    'ImplementationHeaderFile', 'u16_add_SameSlopeZeroBias.h', ...
    'ImplementationSourceFile', 'u16_add_SameSlopeZeroBias.c');
```

## Input Arguments

**hEntry — Handle to a code replacement table entry**

handle

The *hEntry* is a handle to a code replacement table entry previously returned by one of the class instantiations in the table.

| Class Instantiation | Support |
| --- | --- |
| *hEntry* = RTW.TflCOperationEntry; | Supports operator replacement. |
| *hEntry* = RTW.TflCOperationEntryGenerator; | Provides parameters for fixed-point addition and subtraction that are not available in RTW.TflCOperationEntry (SlopesMustBeTheSame and MustHaveZeroNetBias). |
| *hEntry* = RTW.TflCOperationEntryGenerator_NetSlope; | Provides net slope parameters for fixed-point multiplication and division that are not available in RTW.TflCOperationEntry (NetSlopeAdjustmentFactor and NetFixedExponent). |
| *hEntry* = RTW.TflBlasEntryGenerator; | Supports replacement of nonscalar operators with MathWorks BLAS functions. |
| *hEntry* = RTW.TflCBlasEntryGenerator; | Supports replacement of nonscalar operators with ANSI/ISO® C BLAS functions. |
| *hEntry* = *MyCustomOperationEntry*;<br><br>(where *MyCustomOperationEntry* is a class derived from RTW.TflCOperationEntry) | Supports operator replacement using custom code replacement table entries. |

If you want to specify SlopesMustBeTheSame or MustHaveZeroNetBias for your operator entry, instantiate your table entry using *hEntry* =

`RTW.TflCOperationEntryGenerator` rather than *`hEntry`* `=` `RTW.TflCOperationEntry`. If you want to use `NetSlopeAdjustmentFactor` and `NetFixedExponent`, instantiate your table entry by using *`hEntry`* `=` `RTW.TflCOperationEntryGenerator_NetSlope`.

Example: `op_entry`

**`varargin` — Name-value pairs of arguments for function entry**
name-value pairs

Example: `'Key','RTW_OP_ADD'`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Key','RTW_OP_ADD'`

**`AcceptExprInput` — Specifies whether implementation function accepts expression inputs**
`true` | `false`

The *`AcceptExprInput`* value flags the code generator that the implementation function described by this entry accepts expression inputs. The default value is `true` if `ImplType` equals `FCN_IMPL_FUNCT` and `false` if `ImplType` equals `FCN_IMPL_MACRO`.

If the value is `true`, expression inputs are integrated into the generated code in a form similar to this form:

```
rtY.Out1 = mySin(rtU.In1 + rtU.In2);
```

If the value is `false`, a temporary variable is generated for the expression input:

```
real_T rtb_Sum;

rtb_Sum = rtU.In1 + rtU.In2;
rtY.Out1 = mySin(rtb_Sum);
```

Example: `'AcceptExprInput',true`

### `AdditionalHeaderFiles` — Specifies additional header files for table entry

`{}` (default) | array of character vectors

The *`AdditionalHeaderFiles`* value specifies additional header files for a code replacement table entry. The character vectors can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalHeaderFiles',{}`

### `AdditionalIncludePaths` — Specifies additional include paths for table entry

`{}` (default) | array of character vectors

The *`AdditionalIncludePaths`* value specifies the full path of additional include paths for a code replacement entry. The character vectors can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalIncludePaths',{}`

### `AdditionalLinkObjs` — Specifies additional link objects for table entry

`{}` (default) | array of character vectors

The *`AdditionalLinkObjs`* value specifies additional link objects for a code replacement table entry. The character vectors can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalLinkObjs',{}`

### `AdditionalLinkObjsPaths` — Specifies additional link object paths for table entry

`{}` (default) | array of character vectors

The *`AdditionalLinkObjsPaths`* value specifies the full path of additional link object paths for a code replacement entry. The character vectors can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalLinkObjsPaths',{}`

**AdditionalSourceFiles — specifies additional source files for table entry**
{} (default) | array of character vectors

The *AdditionalSourceFiles* value specifies additional source files for a code replacement table entry. The character vectors can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalSourceFiles',{}`

**AdditionalSourcePaths — Specifies additional source paths for table entry**
{} (default) | array of character vectors

The *AdditionalSourcePaths* value specifies the full path of additional source paths for a code replacement entry. The character vectors can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalSourcePaths',{}`

**AdditionalCompileFlags — Specifies additional compiler flags for table entry**
{} (default) | array of character vectors

The *AdditionalCompileFlags* value specifies additional flags required to compile the source files defined for a code replacement table entry.

Example: `'AdditionalCompileFlags',{}`

**AdditionalLinkFlags — Specifies additional linker flags for table entry**
{} (default) | array of character vectors

The *AdditionalLinkFlags* value specifies additional flags required to link the compiled files for a code replacement table entry.

Example: `'AdditionalLinkFlags',{}`

**ArrayLayout — Specifies layout of array storage for table entry**
`'COLUMN_MAJOR'` (default) | `'ROW_MAJOR'` | `'COLUMN_AND_ROW'`

The *ArrayLayout* value specifies the order of array elements in memory supported by the replacement implementation. By default, the replacement implementation supports column-major data layout. For `ROW-MAJOR`, the replacement implementation supports row-major data layout. For `COLUMN_AND_ROW`, the replacement implementation supports column-major and row-major data layouts.

Example: 'ArrayLayout','ROW_MAJOR'

**EntryInfoAlgorithm — Specifies math algorithm to match for table entry**
'RTW_CAST_BEFORE_OP' (default) | 'RTW_CAST_AFTER_OP'

The *EntryInfoAlgorithm* value specifies the algorithm for the specified math operation that must be matched for operation replacement to occur. Code replacement libraries support replacement based on the algorithm for math operations RTW_OP_ADD and RTW_OP_MINUS. Valid arguments for the supported functions are listed in the table. The arguments have the same meaning for both functions.

| Argument | Meaning |
| --- | --- |
| RTW_CAST_BEFORE_OP | Before performing the operation, type cast input values to the output data type. If the output type cannot exactly represent the input values, losses can occur as a result of the cast to the output type. Additional loss can occur when the result of the operation is cast to the final output type. |
| RTW_CAST_AFTER_OP | Compute the ideal result of the operation of inputs. Then, type cast the result to the output data type. Loss occurs during the type cast. This algorithm behaves similarly to the C language except when the signedness of the operands does not match. For example, when you add a signed long operation to an unsigned long operand, standard C language rules convert the signed long operand to an unsigned long operand. The result is a value that is not ideal. |

Example: 'EntryInfoAlgorithm','RTW_CAST_AFTER_OP'

**GenCallback — Specifies callback that follows code generation**
'' (default) | 'RTW.copyFileToBuildDir'

The *GenCallback* specifies a callback that follows code generation. If you specify 'RTW.copyFileToBuildDir', and if this function entry is matched and used, the code generator calls function RTW.copyFileToBuildDir after code generation. This callback function copies additional header, source, or object files that you have specified for this function entry to the build folder. For more information, see “Specify Build Information for Replacement Code”.

Example: `'GenCallback',''`

**`ImplementationHeaderFile` — Specifies name of header file that declares implementation function**

`''` (default) | character vector

The *`ImplementationHeaderFile`* value specifies the name of the header file that declares the implementation function. The character vector can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'ImplementationHeaderFile','<math.h>'`

**`ImplementationHeaderPath` — Specifies path to implementation header file**

`''` (default) | character vector

The *`ImplementationHeaderPath`* value specifies the full path to the implementation header file. The character vector can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'ImplementationHeaderPath',''`

**`ImplementationName` — Specifies name of implementation function**

`''` (default) | character vector

The *`ImplementationName`* value specifies the name of the implementation function, which can match or differ from the `Key` name.

Example: `'ImplementationName','sqrt'`

**`ImplementationSourceFile` — specifies name of implementation source file**

`''` (default) | character vector

The *`ImplementationSourceFile`* value specifies the name of the implementation source file. The character vector can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'ImplementationSourceFile',''`

**`ImplementationSourcePath` — Specifies path to implementation source file**

`''` (default) | character vector

The *ImplementationSourcePath* value specifies the full path to the implementation source file. The character vector can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'ImplementationSourcePath',''`

**`ImplType` — Specifies the type of table entry**
`'FCN_IMPL_FUNCT'` (default) | `'FCN_IMPL_MACRO'`

The *ImplType* value specifies the type of table entry. Use `FCN_IMPL_FUNCT` for function or `FCN_IMPL_MACRO` for macro.

Example: `'ImplType','FCN_IMPL_FUNCT'`

**Key — Specifies key for operator to replace**
character vector

The *Key* value specifies the key for the operator to replace. The key must match an operator key listed in “Code You Can Replace from MATLAB Code” or “Code You Can Replace From Simulink Models”.

Example: `'Key','RTW_OP_ADD'`

**`MustHaveZeroNetBias` — Specifies net bias requirement for concept arguments of Add/Minus entries**
`false` (default) | `true`

The *MustHaveZeroNetBias* value specifies whether a replacement match requires that the net bias for concept arguments of Add/Minus entries is zero. For Mul/Div/MulDiv/Shift/Cast entries, the value of this parameter is ignored, and the bias of all concept arguments for Mul/Div/MulDiv/Shift/Cast entries must be zero for replacement match.

Example: `'MustHaveZeroNetBias',true`

**`Priority` — Specifies the search priority of the function entry**
`100` (default) | integer value 0..100

The *Priority* value specifies the search priority of the function entry, relative to other entries of the same function name and conceptual argument list within this table. Highest priority is 0, and lowest priority is 100. If the table provides two implementations for a function, the implementation with the higher priority shadows the one with the lower priority.

Example: `'Priority',100`

**`RoundingModes` — Specifies rounding modes supported by implementation function**

`'RTW_ROUND_UNSPECIFIED'` (default) | `'RTW_ROUND_FLOOR'` | `'RTW_ROUND_CEILING'` | `'RTW_ROUND_ZERO'` | `'RTW_ROUND_NEAREST'` | `'RTW_ROUND_NEAREST_ML'` | `'RTW_ROUND_CONV'` | `'RTW_ROUND_SIMPLEST'` | array of character vectors

The *`RoundingModes`* value specifies one or more rounding modes supported by the implementation function.

Example: `'RoundingModes',{'RTW_ROUND_UNSPECIFIED'}`

**`SaturationMode` — specifies saturation mode supported by implementation function**

`'RTW_SATURATE_UNSPECIFIED'` (default) | `'RTW_SATURATE_ON_OVERFLOW'` | `'RTW_WRAP_ON_OVERFLOW'` | character vector

The *`SaturationMode`* value specifies the saturation mode supported by the implementation function.

Example: `'SaturationMode','RTW_SATURATE_UNSPECIFIED'`

**`SideEffects` — Specifies whether to attempt to optimize away the implementation function**

`false` (default) | `true`

The *`SideEffects`* value flags the code generator not to optimize away the implementation function described by this entry. This parameter applies to implementation functions that return `void` but are not to be optimized away, such as a `memcpy` implementation or an implementation function that accesses global memory values. For those implementation functions only, you must include this parameter and specify the value `true`.

Example: `'SideEffects',false`

**`SlopesMustBeTheSame` — Specifies slope requirement for concept arguments of Mul/Div/MulDiv/Shift/Cast entries**

`false` (default) | `true`

The *`SlopesMustBeTheSame`* value specifies whether a replacement match requires that the slope is the same for all concept arguments of Mul/Div/MulDiv/Shift/Cast entries. For

Add/Minus entries, the value of this parameter is ignored, and the slope of concept arguments for Add/Subtract entries must always be the same for replacement match.

Example: `'SlopesMustBeTheSame',true`

#### `StoreFcnReturnInLocalVar` — Specifies whether to store the implementation function regardless expression folding settings

`false` (default) | `true`

The *`StoreFcnReturnInLocalVar`* value flags the code generator that the return value of the implementation function described by this entry must be stored in a local variable regardless of other expression folding settings. If the value is `false`, other expression folding settings determine whether the return value is folded. Storing function returns in a local variable can increase the clarity of generated code. This example shows code generated with expression folding:

```
void sw_step(void)
{
    if (ssub(sadd(sw_U.In1, sw_U.In2), sw_U.In3) <=
        smul(ssub(sw_U.In4, sw_U.In5),sw_U.In6)) {
      sw_Y.Out1 = sw_U.In7;
    } else {
       sw_Y.Out1 = sw_U.In8;
    }
}
```

With `StoreFcnReturnInLocalVar` set to `true`, the generated code is potentially easier to understand and debug:

```
void sw_step(void)
{
    real32_T rtb_Switch;
    real32_T hoistedExpr;
    ......
    rtb_Switch = sadd(sw_U.In1, sw_U.In2);
    rtb_Switch = ssub(rtb_Switch, sw_U.In3);
    hoistedExpr = ssub(sw_U.In4, sw_U.In5);
    hoistedExpr = smul(hoistedExpr, sw_U.In6);
    if (rtb_Switch <= hoistedExpr) {
       sw_Y.Out1 = sw_U.In7;
    } else {
       sw_Y.Out1 = sw_U.In8;
    }
}
```

Example: `'StoreFcnReturnInLocalVar',false`

## See Also

`addAdditionalHeaderFile` | `addAdditionalIncludePath` | `addAdditionalLinkObj` | `addAdditionalLinkObjPath` | `addAdditionalSourceFile` | `addAdditionalSourcepath`

### Topics

"Specify Build Information for Replacement Code"
"Define Code Replacement Mappings"
"Scalar Operator Code Replacement"
"Addition and Subtraction Operator Code Replacement"
"Small Matrix Operation to Processor Code Replacement"
"Code You Can Replace from MATLAB Code"
"Code You Can Replace From Simulink Models"

**Introduced in R2007b**

# setTflCSemaphoreEntryParameters

Set specified parameters for semaphore entry in code replacement table

## Syntax

`setTflCSemaphoreEntryParameters(hEntry,varargin)`

## Description

`setTflCSemaphoreEntryParameters(hEntry,varargin)` sets specified parameters for a semaphore entry in a code replacement table.

## Examples

### Specify Semaphore Initialization Parameters for Table Entry

This example shows how to use the `setTflCSemaphoreEntryParameters` function to set specified parameters for a code replacement table entry for a semaphore initialization replacement.

```
sem_entry = RTW.TflCSemaphoreEntry;
sem_entry.setTflCSemaphoreEntryParameters( ...
    'Key',                        'RTW_SEM_INIT', ...
    'Priority',                   100, ...
    'ImplementationName',         'mySemCreate', ...
    'ImplementationHeaderFile', 'mySem.h', ...
    'ImplementationSourceFile', 'mySem.c', ...
```

```
    'GenCallback',              'RTW.copyFileToBuildDir', ...
    'SideEffects',              true);
```

## Input Arguments

**`hEntry` — Handle to semaphore entry**
handle

The *`hEntry`* is a handle to a code replacement library semaphore entry previously returned by *`hEntry`* `= RTW.TflCSemaphoreEntry;`.

Example: `sem_entry`

**`varargin` — Name-value pairs of arguments for function entry**
name-value pairs

Example: `'Key','RTW_SEM_INIT'`

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Key','RTW_SEM_INIT'`

**`AcceptExprInput` — Specifies whether implementation function accepts expression inputs**
`true` | `false`

The *`AcceptExprInput`* value flags the code generator that the implementation function described by this entry accepts expression inputs. The default value is `true` if `ImplType` equals `FCN_IMPL_FUNCT` and `false` if `ImplType` equals `FCN_IMPL_MACRO`.

If the value is `true`, expression inputs are integrated into the generated code in a form similar to this form:

```
rtY.Out1 = mySin(rtU.In1 + rtU.In2);
```

If the value is `false`, a temporary variable is generated for the expression input:

```
real_T rtb_Sum;

rtb_Sum = rtU.In1 + rtU.In2;
rtY.Out1 = mySin(rtb_Sum);
```

Example: `'AcceptExprInput',true`

**`AdditionalHeaderFiles` — Specifies additional header files for table entry**
`{}` (default) | array of character vectors

The *`AdditionalHeaderFiles`* value specifies additional header files for a code replacement table entry. The character vectors can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalHeaderFiles',{}`

**`AdditionalIncludePaths` — Specifies additional include paths for table entry**
`{}` (default) | array of character vectors

The *`AdditionalIncludePaths`* value specifies the full path of additional include paths for a code replacement entry. The character vectors can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalIncludePaths',{}`

**`AdditionalLinkObjs` — Specifies additional link objects for table entry**
`{}` (default) | array of character vectors

The *`AdditionalLinkObjs`* value specifies additional link objects for a code replacement table entry. The character vectors can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalLinkObjs',{}`

**`AdditionalLinkObjsPaths` — Specifies additional link object paths for table entry**
`{}` (default) | array of character vectors

The *`AdditionalLinkObjsPaths`* value specifies the full path of additional link object paths for a code replacement entry. The character vectors can include tokens. For

example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalLinkObjsPaths',{}`

**`AdditionalSourceFiles` — specifies additional source files for table entry**
`{}` (default) | array of character vectors

The *`AdditionalSourceFiles`* value specifies additional source files for a code replacement table entry. The character vectors can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalSourceFiles',{}`

**`AdditionalSourcePaths` — Specifies additional source paths for table entry**
`{}` (default) | array of character vectors

The *`AdditionalSourcePaths`* value specifies the full path of additional source paths for a code replacement entry. The character vectors can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector. The default is `{}`.

Example: `'AdditionalSourcePaths',{}`

**`AdditionalCompileFlags` — Specifies additional compiler flags for table entry**
`{}` (default) | array of character vectors

The *`AdditionalCompileFlags`* value specifies additional flags required to compile the source files defined for a code replacement table entry.

Example: `'AdditionalCompileFlags',{}`

**`AdditionalLinkFlags` — Specifies additional linker flags for table entry**
`{}` (default) | array of character vectors

The *`AdditionalLinkFlags`* value specifies additional flags required to link the compiled files for a code replacement table entry.

Example: `'AdditionalLinkFlags',{}`

**`GenCallback` — Specifies callback that follows code generation**
`''` (default) | `'RTW.copyFileToBuildDir'`

The *GenCallback* specifies a callback that follows code generation. If you specify 'RTW.copyFileToBuildDir', and if this function entry is matched and used, the function RTW.copyFileToBuildDir is called after code generation. This callback function copies additional header, source, or object files that you have specified for this function entry to the build folder. For more information, see “Specify Build Information for Replacement Code”.

Example: 'GenCallback',''

**ImplementationHeaderFile — Specifies name of header file that declares implementation function**

'' (default) | character vector

The *ImplementationHeaderFile* value specifies the name of the header file that declares the implementation function. The character vector can include tokens. For example, in the token $mytoken$, mytoken is a variable defined as a character vector in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: 'ImplementationHeaderFile','<math.h>'

**ImplementationHeaderPath — Specifies path to implementation header file**

'' (default) | character vector

The *ImplementationHeaderPath* value specifies the full path to the implementation header file. The character vector can include tokens. For example, in the token $mytoken $, mytoken is a variable defined as a character vector in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: 'ImplementationHeaderPath',''

**ImplementationName — Specifies name of implementation function**

'' (default) | character vector

The *ImplementationName* value specifies the name of the implementation function, which can match or differ from the Key name.

Example: 'ImplementationName','sqrt'

**ImplementationSourceFile — specifies name of implementation source file**

'' (default) | character vector

The *ImplementationSourceFile* value specifies the name of the implementation source file. The character vector can include tokens. For example, in the token $mytoken

$, `mytoken` is a variable defined as a character vector in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'ImplementationSourceFile',''`

**`ImplementationSourcePath` — Specifies path to implementation source file**
`''` (default) | character vector

The *`ImplementationSourcePath`* value specifies the full path to the implementation source file. The character vector can include tokens. For example, in the token `$mytoken` $, `mytoken` is a variable defined as a character vector in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'ImplementationSourcePath',''`

**`ImplType` — Specifies the type of table entry**
`'FCN_IMPL_FUNCT'` (default) | `'FCN_IMPL_MACRO'`

The *`ImplType`* value specifies the type of table entry. Use `FCN_IMPL_FUNCT` for function or `FCN_IMPL_MACRO` for macro.

Example: `'ImplType','FCN_IMPL_FUNCT'`

**Key — Specifies key for operator to replace**
character vector

The *Key* value specifies the key for the operator to replace. The name must match a function name listed in “Code You Can Replace from MATLAB Code” or “Code You Can Replace From Simulink Models”.

Example: `'Key','RTW_OP_ADD'`

**`Priority` — Specifies the search priority of the function entry**
`100` (default) | integer value 0..100

The *`Priority`* value specifies the search priority of the function entry, relative to other entries of the same function name and conceptual argument list within this table. Highest priority is 0, and lowest priority is 100. The default is 100. If the table provides two implementations for a function, the implementation with the higher priority shadows the one with the lower priority.

Example: `'Priority',100`

**RoundingModes — Specifies rounding modes supported by implementation function**

'RTW_ROUND_UNSPECIFIED' (default) | 'RTW_ROUND_FLOOR' | 'RTW_ROUND_CEILING' | 'RTW_ROUND_ZERO' | 'RTW_ROUND_NEAREST' | 'RTW_ROUND_NEAREST_ML' | 'RTW_ROUND_CONV' | 'RTW_ROUND_SIMPLEST' | array of character vectors

The *RoundingModes* value specifies one or more rounding modes supported by the implementation function.

Example: 'RoundingModes',{'RTW_ROUND_UNSPECIFIED'}

**SaturationMode — specifies saturation mode supported by implementation function**

'RTW_SATURATE_UNSPECIFIED' (default) | 'RTW_SATURATE_ON_OVERFLOW' | 'RTW_WRAP_ON_OVERFLOW' | character vector

The *SaturationMode* value specifies the saturation mode supported by the implementation function.

Example: 'SaturationMode','RTW_SATURATE_UNSPECIFIED'

**SideEffects — Specifies whether to attempt to optimize away the implementation function**

false (default) | true

The *SideEffects* value flags the code generator not to optimize away the implementation function described by this entry. This parameter applies to implementation functions that return void but are not to be optimized away, such as a memcpy implementation or an implementation function that accesses global memory values. For those implementation functions only, you must include this parameter and specify the value true.

Example: 'SideEffects',false

**StoreFcnReturnInLocalVar — Specifies whether to store the implementation function regardless expression folding settings**

false (default) | true

The *StoreFcnReturnInLocalVar* value flags the code generator that the return value of the implementation function described by this entry must be stored in a local variable regardless of other expression folding settings. If the value is false, other expression folding settings determine whether the return value is folded. Storing function returns in

a local variable can increase the clarity of generated code. This example shows code generated with expression folding:

```
void sw_step(void)
{
    if (ssub(sadd(sw_U.In1, sw_U.In2), sw_U.In3) <=
        smul(ssub(sw_U.In4, sw_U.In5),sw_U.In6)) {
      sw_Y.Out1 = sw_U.In7;
    } else {
       sw_Y.Out1 = sw_U.In8;
    }
}
```

With `StoreFcnReturnInLocalVar` set to `true`, the generated code is potentially easier to understand and debug:

```
void sw_step(void)
{
    real32_T rtb_Switch;
    real32_T hoistedExpr;
    ......
    rtb_Switch = sadd(sw_U.In1, sw_U.In2);
    rtb_Switch = ssub(rtb_Switch, sw_U.In3);
    hoistedExpr = ssub(sw_U.In4, sw_U.In5);
    hoistedExpr = smul(hoistedExpr, sw_U.In6);
    if (rtb_Switch <= hoistedExpr) {
       sw_Y.Out1 = sw_U.In7;
    } else {
       sw_Y.Out1 = sw_U.In8;
    }
}
```

Example: `'StoreFcnReturnInLocalVar',false`

# See Also

`addAdditionalHeaderFile` | `addAdditionalIncludePath` | `addAdditionalLinkObj` | `addAdditionalLinkObjPath` | `addAdditionalSourceFile` | `addAdditionalSourcepath`

## Topics

"Specify Build Information for Replacement Code"

"Define Code Replacement Mappings"
"Code You Can Replace from MATLAB Code"
"Code You Can Replace From Simulink Models"
"Mutex and Semaphore Functions"

**Introduced in R2013a**

# coder.MATLABCodeTemplate.setTokenValue

**Class:** coder.MATLABCodeTemplate
**Package:** coder

Set value of token for code generation template

## Syntax

`setTokenValue(tokenName,tokenValue)`

## Description

`setTokenValue(tokenName,tokenValue)` sets the value of a token for a code generation template.

## Input Arguments

**`tokenName`**

The name of the token

**Default:**

**`tokenValue`**

The value of the token

**Default:** empty

## Examples

Create a MATLABCodeTemplate object from a custom template. Set the value for a custom token in the template.

```
newObj = coder.MATLABCodeTemplate('myCGTFile');
% Create a MATLABCodeTemplate object from a custom template file
newObj.setTokenValue('myCustomToken', 'myValue');
% Set the value of a custom token in the file
newObj.getTokenValue('myCustomToken')
% Check value of the custom token
```

## See Also

`coder.MATLABCodeTemplate.emitSection` | `coder.MATLABCodeTemplate.getCurrentTokens` | `coder.MATLABCodeTemplate.getTokenValue`

### Topics

“Generate Custom File and Function Banners for C/C++ Code”
“Code Generation Template Files for MATLAB Code”

# run

Execute program loaded on processor

## Syntax

```
run(IDE_Obj)
run(IDE_Obj,'runopt')
run(IDE_Obj,…,timeout)
```

## IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

## Description

`run(IDE_Obj)` runs the program file loaded on the referenced processor, returning immediately after the processor starts running. Program execution starts from the location of program counter (PC). Usually, the program counter is positioned at the top of the executable file. However, if you stopped a running program with `halt`, the program counter may be anywhere in the program. `run` starts the program from the program counter current location.

If `IDE_Obj` references more the one processor, each processors calls `run` in sequence.

`run(IDE_Obj,'runopt')` includes the parameter `runopt` that defines the action of the `run` method. The options for `runopt` are listed in the following table.

| runopt Value | Description |
| --- | --- |
| `'run'` | Executes the run and waits to confirm that the processor is running, and then returns to MATLAB. |

| runopt Value | Description |
|---|---|
| `'runtohalt'` | Executes the run but then waits until the processor halts before returning. The halt can be the result of the PC reaching a breakpoint, or by direct interaction with the IDE, or by the normal program exit process. |
| `'tohalt'` | Waits until the running program has halted. Unlike the other options, this selection does not execute a run, it simply waits for the running program to halt. |
| `'main'` | This option resets the program and executes a run until the start of function `'main'`. |
| `'tofunc'` | This option must be followed by an extra parameter *funname*, the name of the function to run to:<br><br>`run(IDE_Obj,'tofunc',funcname)`<br><br>This executes a run from the present PC location until the start of function *funcname* is reached. If *funcname* is not along the program's normal execution path, *funcname* is not reached and the method times out. |

In the `'run'` and `'runtohalt'` cases, a halt can be caused by a breakpoint, a direct interaction with the IDE, or by a normal program exit.

The following table shows the availability of the *runopt* options by IDE.

| | CCS IDE | VisualDSP++ IDE |
|---|---|---|
| `'run'` | Yes | Yes |
| `'runtohalt'` | Yes | Yes |
| `'tohalt'` | Yes | |
| `'main'` | Yes | |
| `'tofunc'` | Yes | |

`run(IDE_Obj,…,timeout)` adds input argument `timeout`, to allow you to set the time out to a value different from the global timeout value. The *timeout* value specifies how long, in seconds, MATLAB waits for the processor to start executing the loaded program before returning.

Most often, the `'run'` and `'runtohalt'` options cause the processor to initiate execution, even when a timeout is reached. The timeout indicates that the confirmation was not received before the timeout period elapsed.

## See Also

`halt` | `load` | `reset`

**Introduced in R2011a**

# save

Save file

## Syntax

`save(IDE_Obj,filename,filetype)`

## IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

## Description

Use `save(IDE_Obj,filename,filetype)` to save open files in the IDE project.

The *filename* argument defines the name of the file to save. When entering the file name, include the file extension.

The optional *filetype* argument defines the type of file to save. If you omit the *filetype* argument, *filetype* defaults to `'project'`. Except with VisualDSP++ IDE, `'project'` is the only supported option. Therefore, you can omit the *filetype* argument in most cases.

| | CCS IDE | VisualDSP++ IDE |
|---|---|---|
| `'project'` | Yes | Yes |
| `'projectgroup'` | No | Yes |

**Note** The `open` method does not support the `'text'` argument.

## Examples

To save the project files:

```
save(IDE_Obj,'all')
```

To save the `myproject` project:

```
save(IDE_Obj,'myproject')
```

To save the active project:

```
save(IDE_Obj,[])
```

For VisualDSP++ IDE, to save the projects in the project groups:

```
save(IDE_Obj,'all','projectgroup')
```

For VisualDSP++ IDE, to save the `myg.dpg` project group:

```
save(IDE_Obj,'myg.dpg','projectgroup')
```

For VisualDSP++ IDE, to save the active project in the project groups:

```
save(IDE_Obj,[],'projectgroup')
```

## See Also

`adivdsp` | `close` | `load`

**Introduced in R2011a**

# setbuildopt

Set active configuration build options

## Syntax

```
setbuildopt(IDE_Obj,tool,ostr)
setbuildopt(IDE_Obj,file,ostr)
```

## IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

## Description

Use `setbuildopt(`*`IDE_Obj`*`,`*`tool`*`,`*`ostr`*`)` to set the build options for a specific build tool in the current configuration. This replaces the switch settings that are applied when you invoke the command line `tool`. For example, a build tool could be a compiler, linker or assembler. To define the *`tool`* argument, first use the `getbuildopt` command to read a list of defined build tools.

If the VisualDSP++ and Code Composer Studio IDEs do not recognize the *`ostr`* argument, `setbuildopt` sets the switch settings to the default values for the build tool specified by `tool`.

Use `setbuildopt(`*`IDE_Obj`*`,`*`file`*`,`*`ostr`*`)` to configure the build options for a file you specify with the *`file`* argument. The source file must exist in the active project.

## See Also

`activate` | `getbuildopt`

**Introduced in R2011a**

# setAlgorithmParameters

Set algorithm parameters for lookup table function code replacement table entry

## Syntax

`setAlgorithmParameters(tableEntry, algParams)`

## Description

`setAlgorithmParameters(tableEntry, algParams)` sets the algorithm parameters for the lookup table function identified in the code replacement table entry `tableEntry`.

## Examples

### Set Algorithm Parameters for `prelookup` Function Table Entry

Create a code replacement table.

```
crTable = RTW.TflTable;
```

Create a table entry for a function.

```
tableEntry = RTW.TflCFunctionEntry;
```

Identify the table entry as an entry for the `prelookup` function.

```
setTflCFunctionEntryParameters(tableEntry, ...
    'Key', 'prelookup', ...
    'Priority', 100, ...
    'ImplementationName', 'Ifx_DpSearch_u8');
```

Get the algorithm parameter settings for the `prelookup` function table entry.

```
algParams = getAlgorithmParameters(tableEntry)
```

```
algParams =

  Prelookup with properties:

           ExtrapMethod: [1x1 coder.algorithm.parameter.ExtrapMethod]
```

```
              RndMeth: [1x1 coder.algorithm.parameter.RndMeth]
    IndexSearchMethod: [1x1 coder.algorithm.parameter.IndexSearchMethod]
    UseLastBreakpoint: [1x1 coder.algorithm.parameter.UseLastBreakpoint]
RemoveProtectionInput: [1x1 coder.algorithm.parameter.RemoveProtectionInput]
```

Display the valid values for parameter `UseLastBreakPoint` for the `prelookup` function.

```
algParams.UseLastBreakPoint
```

```
ans = 

  UseLastBreakpoint with properties:

       Name: 'UseLastBreakpoint'
    Options: {'off'  'on'}
    Primary: 0
      Value: {'off'  'on'}
```

Display the valid values for parameter `RemoveProtectionInput` for the `prelookup` function.

```
algParams.RemoveProtectionInput
```

```
ans = 

  RemoveProtectionInput with properties:

       Name: 'RemoveProtectionInput'
    Options: {'off'  'on'}
    Primary: 0
      Value: {'off'  'on'}
```

Set parameters `UseLastBreakPoint` and `RemoveProtectionInput` to `on` and `off`, respectively.

```
algParams.UseLastBreakPoint = 'on';
algParams.RemoveProtectionInput = 'off';
```

When you set each parameter, the algorithm parameter software checks for and reports errors for invalid syntax, parameter names, and values.

Update the parameter settings for the code replacement table entry.

```
setAlgorithmParameters(tableEntry, algParams);
```

Get the new algorithm parameter settings for the `prelookup` function table entry.

```
algParams = getAlgorithmParameters(tableEntry);
```

Examine the new value for `UseLastBreakPoint`.

```
algParams.UseLastBreakPoint
```

```
ans = 

  UseLastBreakpoint with properties:
```

```
       Name: 'UseLastBreakpoint'
    Options: {'off'  'on'}
    Primary: 0
      Value: {'on'}
```

Examine the new value for `RemoveProtectionInput`.

```
algParams.RemoveProtectionInput
```

```
ans = 

  RemoveProtectionInput with properties:

       Name: 'RemoveProtectionInput'
    Options: {'off'  'on'}
    Primary: 0
      Value: {'off'}
```

### Set Algorithm Parameters for `lookup2D` Function Table Entry

Create a code replacement table.

```
crTable = RTW.TflTable;
```

Create a table entry for a function.

```
tableEntry = RTW.TflCFunctionEntry;
```

Identify the table entry as an entry for the `lookup2D` function.

```
setTflCFunctionEntryParameters(tableEntry, ...
    'Key', 'lookup2D', ...
    'Priority', 100, ...
    'ImplementationName', 'myLookup2D');
```

Get the algorithm parameter settings for the `lookup2D` function table entry.

```
algParams = getAlgorithmParameters(tableEntry)
```

```
algParams = 

  Lookup with properties:

                 InterpMethod: [1x1 coder.algorithm.parameter.InterpMethod]
                 ExtrapMethod: [1x1 coder.algorithm.parameter.ExtrapMethod]
                      RndMeth: [1x1 coder.algorithm.parameter.RndMeth]
            IndexSearchMethod: [1x1 coder.algorithm.parameter.IndexSearchMethod]
            UseLastTableValue: [1x1 coder.algorithm.parameter.UseLastTableValue]
        RemoveProtectionInput: [1x1 coder.algorithm.parameter.RemoveProtectionInput]
    SaturateOnIntegerOverflow: [1x1 coder.algorithm.parameter.SaturateOnIntegerOverflow]
      SupportTunableTableSize: [1x1 coder.algorithm.parameter.SupportTunableTableSize]
              BPPower2Spacing: [1x1 coder.algorithm.parameter.BPPower2Spacing]
```

Display the valid values for algorithm parameter `IndexSearchMethod` for the `lookup2D` function.

```
algParams.IndexSearchMethod
```

```
ans =

  IndexSearchMethod with properties:

       Name: 'IndexSearchMethod'
    Options: {'Linear search'  'Binary search'  'Evenly spaced points'}
    Primary: 0
      Value: {'Binary search'  'Evenly spaced points'  'Linear search'}
```

Set parameter `IndexSearchMethod` to `Evenly spaced point`.

```
algParams.IndexSearchMethod = 'Evenly spaced point';
```

```
Error using coder.algorithm.parameter.validateValue (line 58)
Invalid value '{Evenly spaced point}' for algorithm parameter
'coder.algorithm.parameter.IndexSearchMethod'. Valid values are '{Linear
search, Binary search, Evenly spaced points}'.

Error in coder.algorithm.parameter.AlgorithmParameter/set.Value (line 49)
            obj.Value = coder.algorithm.parameter.validateValue(obj, val);

Error in coder.algorithm.parameter.AlgorithmParameter/setAP (line 36)
                obj.Value = value;

Error in coder.algorithm.parameterset.Lookup/set.IndexSearchMethod (line 39)
            obj.IndexSearchMethod = obj.IndexSearchMethod.setAP(value);
```

The code replacement software flags the ‘`s`’ that is missing from ‘`points`’.

Adjust the parameter setting.

```
algParams.IndexSearchMethod = 'Evenly spaced points';
```

Update the parameter settings for the code replacement table entry.

```
setAlgorithmParameters(tableEntry, algParams);
```

Get the updated algorithm parameter settings for the `lookup2D` function table entry.

```
algParams = getAlgorithmParameters(tableEntry);
```

Verify the new value of `IndexSearchMethod`.

```
algParams.IndexSearchMethod
```

```
ans =

  IndexSearchMethod with properties:

       Name: 'IndexSearchMethod'
    Options: {'Linear search'  'Binary search'  'Evenly spaced points'}
```

```
    Primary: 0
      Value: {'Evenly spaced points'}
```

## Input Arguments

**`tableEntry` — Code replacement table entry for a lookup table function**
object

Code replacement table entry that you previously created and represents a potential code replacement for a lookup table function. The entry must identify the lookup table function for which you are calling `setAlgorithmParameters`.

1. Create the entry. For example, call the function `RTW.TflCFunctionEntry`.

   ```
   tableEntry = RTW.TflCFunctionEntry;
   ```
2. Specify the name of the lookup table function for which you created the entry. Use the `Key` parameter in a call to `setTflCFunctionEntryParameters`. The following function call specifies the lookup table function `prelookup`.

   ```
   setTflCFunctionEntryParameters(tableEntry, ...
      'Key', 'prelookup', ...
      'Priority', 100, ...
      'ImplementationName', 'Ifx_DpSearch_u8');
   ```

**`algParams` — Algorithm parameter settings for a lookup table function**
object

Algorithm parameter settings for the lookup table function identified with the `Key` parameter in `tableEntry`.

## See Also

`RTW.TflCFunctionEntry` | `RTW.TflTable` | `addEntry` | `getAlgorithmParameters` | `setTflCFunctionEntryParameters`

### Topics

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

**Introduced in R2015a**

# symbol

Program symbol table from IDE

## Syntax

`s = symbol(IDE_Obj)`

## IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

## Description

`s = symbol(IDE_Obj)` returns the symbol table for the program loaded in the processor associated with the IDE handle object, *IDE_Obj*. The `symbol` method only applies after you load a processor program file. `s` is an array of structures where each row in `s` presents the symbol name and address in the table. Therefore, `s` has two columns; one is the symbol name, and the other is the symbol address and symbol page.

For CCS IDE, this table shows a few possible elements of `s`, and their interpretation.

| s Structure Field | Contents of the Specified Field |
|---|---|
| `s(1).name` | Character vector reflecting the symbol entry name. |
| `s(1).address(1)` | Address or value of symbol entry. |
| `s(1).address(2)` | Memory page for the symbol entry. For TI C6xxx processors, the page is 0. |

You can use field `address` in `s` as the `address` input argument to `read` and `write`.

It you use `symbol` and the symbol table does not exist, `s` returns empty and you get a warning message.

Symbol tables are a portion of a COFF object file that contains information about the symbols that are defined and used by the file. When you load a program to the processor, the symbol table resides in the IDE. While the IDE may contain more than one symbol table at a time, `symbol` accesses the symbol table belonging to the program you last loaded on the processor.

## Examples

Build and load a example program on your processor. Then use `symbol` to return the entries stored in the symbol table in the processor.

```
s = symbol(IDE_Obj);
```

`s` contains the symbols and their addresses, in a structure you can display with the following code:

```
for k = 1:length(s),disp(k),disp(s(k)),end;
```

MATLAB software lists the symbols from the symbol table in a column.

## See Also

`load` | `run`

**Introduced in R2011a**

# ticcs

Create handle object to interact with Code Composer Studio IDE

## Syntax

```
IDE_Obj = ticcs
IDE_Obj = ticcs('propertyname','propertyvalue',...)
```

## IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

## Description

`IDE_Obj = ticcs` returns a ticcs object in `IDE_Obj` that MATLAB software uses to communicate with the default processor. The output argument name you provide for `ticcs` cannot begin with an underscore, such as `_IDE_Obj`. If you do not use input arguments, `ticcs` constructs the object with default values for the properties. the IDE handles the communications between MATLAB software and the selected CPU. When you use the function, `ticcs` starts the IDE if it is not running. If `ticcs` opened an instance of the IDE when you issued the `ticcs` function, the IDE becomes invisible after the code generator creates the new object.

---

**Note** When `ticcs` creates the object `IDE_Obj`, it sets the working folder for the IDE to be the same as your MATLAB Current Folder. When you create files or projects in the IDE, or save files and projects, this working folder affects where you store the files and projects.

---

Each object that accesses the IDE comprises two objects—a `ticcs` object and an `rtdx` object—that include the following properties.

| Object | Property Name | Property | Default | Description |
|---|---|---|---|---|
| ticcs | 'apiversion' | API version | N/A | Defines the API version used to create the link. |
| | 'proctype' | Processor Type | N/A | Specifies the kind of processor on the board. |
| | 'procname' | Processor Name | CPU | Name given to the processor on the board to which this object links. |
| | 'status' | Running | No | Status of the program currently loaded on the processor. |
| | 'boardnum' | Board Number | 0 | Number that CCS assigns to the board. Used to identify the board. |
| | 'procnum' | Processor number | 0 | Number the CCS assigns to a processor on a board. |
| | 'timeout' | Default timeout | 10.0 s | Specifies how long MATLAB software waits for a response from CCS after issuing a request. This also applies when you try to construct a `ticcs` object. The create process waits for this timeout period for the connection to the processor to complete. If the timeout period expires, you get an error message that the connection to the processor failed and MATLAB software could not create the `ticcs` object. |
| rtdx | 'timeout' | Timeout | 10.0 s | Specifies how long CCS waits for a response from the processor after requesting data. |

| Object | Property Name | Property | Default | Description |
|---|---|---|---|---|
| | 'numchannels' | Number of open channels | 0 | The number of open channels using this link. |

`IDE_Obj = ticcs('propertyname','propertyvalue',...)` returns a handle in `IDE_Obj` that MATLAB software uses to communicate with the specified processor. CCS handles the communications between the MATLAB environment and the CPU.

MATLAB software treats input parameters to `ticcs` as property definitions. Each property definition consists of a property name/property value pair.

Two properties of the `ticcs` object are read only after you create the object:

- `'boardnum'` — The identifier for the installed board selected from the active boards recognized by CCS. If you have one board, use the default property value 0 to access the board.
- `'procnum'` — The identifier for the processor on the board defined by `boardnum`. On boards with more than one processor, use this value to specify the processor on the board. On boards with one processor, use the default property value 0 to specify the processor.

Given these two properties, the most common forms of the `ticcs` method are

```
IDE_Obj = ticcs('boardnum',value)
IDE_Obj = ticcs('boardnum',value,'procnum',value)
IDE_Obj = ticcs(...,'timeout',value)
```

which specify the board, and processor in the second example, as the processor.

The third example adds the `timeout` input argument and `value` to allow you to specify how long MATLAB software waits for the connection to the processor or the response to a command to return completed.

You do not need to specify the `boardnum` and `procnum` properties when you have one board with one processor installed. The default property values refer to the processor on the board.

---

**Note** Simulators are considered boards. If you defined both boards and simulators in the IDE, specify the `boardnum` and `procnum` properties to connect to specific boards or simulators. Use `ccsboardinfo` to determine the values for the `boardnum` and `procnum` properties.

---

Because these properties are read only after you create the handle, you must set these property values as input arguments when you use `ticcs`. You cannot change these values after the handle exists. After you create the handle, use the `get` function to retrieve the `boardnum` and `procnum` property values.

## Using ticcs with Multiple Processor Boards

When you create `ticcs` objects that access boards that contain more than one processor, such as the OMAP1510 platform, `ticcs` behaves a little differently.

For each of the `ticcs` syntaxes, the result of the method changes in the multiple processor case, as follows.

```
IDE_Obj = ticcs
IDE_Obj = ticcs('propertyname',propertyvalue)
IDE_Obj = ticcs('propertyname',propertyvalue,'propertyname',...
propertyvalue)
```

In the case where you do not specify a board or processor:

```
IDE_Obj = ticcs
Array of TICCS Objects:
 API version            : 1.2
 Board name             : OMAP 3.0 Platform Simulator [Texas
Instruments]
 Board number           : 0
 Processor 0 (element 1): TMS470R2127 (MPU, Not Running)
 Processor 1 (element 2): TMS320C5500 (DSP, Not Running)
```

Where you choose to identify your processor as an input argument to `ticcs`, for example, when your board contains two processors:

```
IDE_Obj = ticcs('boardnum',2)
Array of TICCS Objects:
 API version            : 1.2
 Board name            : OMAP 3.0 Platform Simulator [Texas Instruments]
 Board number           : 2
 Processor 0 (element 1) : TMS470R2127 (MPU, Not Running)
 Processor 1 (element 2) : TMS320C5500 (DSP, Not Running)
```

`IDE_Obj` returns a two element object handle with `IDE_Obj(1)` corresponding to the first processor and `IDE_Obj(2)` corresponding to the second.

You can include both the board number and the processor number in the `ticcs` syntax. For example:

```
IDE_Obj = ticcs('boardnum',2,'procnum',[0 1])
Array of TICCS Objects:
 API version          : 1.2
 Board name           : OMAP 3.0 Platform Simulator [Texas
Instruments]
```

```
 Board number         : 2
 Processor 0 (element 1) : TMS470R2127 (MPU, Not Running)
 Processor 1 (element 2) : TMS320C5500 (DSP, Not Running)
```

Enter `procnum` as either a single processor on the board (a single value in the input arguments to specify one processor) or a vector of processor numbers, as shown in the example, to select two or more processors.

## Support Coemulation and OMAP

Coemulation, defined by Texas Instruments to mean simultaneous debugging of two or more CPUs, allows you to coordinate your debugging efforts between two or more processors within one device. Efficient development with OMAPo hardware requires coemulation support. Instead of creating one `IDE_Obj` object when you issue the following command

```
IDE_Obj = ticcs
```

or your hardware that has multiple processors, the resulting `IDE_Obj` object comprises a vector of `IDE_Obj` objects `IDE_Obj(1)`, `IDE_Obj(2)`, and so on, each of which accesses one processor on your device, say an OMAP1510. When your processor has one processor, `IDE_Obj` is a single object. With a multiprocessor board, the `IDE_Obj` object returns the new vector of objects. For example, for board 2 with two processors,

```
IDE_Obj = ticcs
```

returns the following information about the board and processors:

```
IDE_Obj = ticcs('boardnum',2)
Array of TICCS Objects:
API version              : 1.2
 Board name               : OMAP 3.0 Platform Simulator [Texas
Instruments]
 Board number             : 2
 Processor 0  (element 1) : TMS470R2127 (MPU, Not Running)
 Processor 1  (element 2) : TMS320C5500 (DSP, Not Running)
```

Checking the existing boards shows that board 2 does have two processors:

```
ccsboardinfo

Board Board                             Proc Processor    Processor
Num   Name                              Num  Name         Type
 ---  ---------------------------------  ---  ---------------
2    OMAP 3.0 Platform Simulator [T ... 0    MPU          TMS470R2x
2    OMAP 3.0 Platform Simulator [T ... 1    DSP          TMS320C550
1    MGS3 Simulator [Texas Instruments] 0    CPU          TMS320C5500
0    ARM925 Simulator [Texas Instru ... 0    CPU          TMS470R2x
```

## Examples

On a system with three boards, where the third board has one processor and the first and second boards have two processors each, the following function:

```
IDE_Obj = ticcs('boardnum',1,'procnum',0);
```

returns an object that accesses the first processor on the second board. Similarly, the function

```
IDE_Obj = ticcs('boardnum',0,'procnum',1);
```

returns an object that refers to the second processor on the first board.

To access the processor on the third board, use

```
IDE_Obj = ticcs('boardnum',2);
```

which sets the default property value `procnum` = 0 to connect to the processor on the third board.

```
IDE_Obj = ticcs
TICCS Object:
 API version      : 1.2
 Processor type   : TMS320C6711
 Processor name   : CPU_1
 Running?         : No
 Board number     : 1
 Processor number : 0
 Default timeout  : 10.00 secs

  RTDX channels    : 0


Defined types : Void, Float, Double, Long, Int, Short, Char
```

## See Also

`ccsboardinfo`

### Introduced in R2011a

# visible

Set whether IDE window appears while IDE runs

## Syntax

`visible(IDE_Obj,state)`

## IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

## Description

Use `visible(IDE_Obj,state)` to make the IDE visible on the desktop or make it run in the background.

To run the IDE in the background so it is not visible on the desktop, enter `'0'` for the *state* argument.

To make the IDE visible on your system desktop, enter `'1'` for the *state* argument.

You can use methods to interact with a IDE handle object, such as `IDE_Obj`, while the IDE is in both states, visible and not visible. You can interact with the IDE GUI while the IDE is visible.

On the Microsoft Windows platform, if you make the IDE visible and look at the Windows Task Manager:

- While the IDE is visible (*state* is `1`), the IDE appears on the **Applications** page of Task Manager, and the `IDE_Obj_app.exe` process shows up on the **Processes** page as a running process.

- While the IDE is not visible (*state* is 0), the IDE disappears from the **Applications** page, but remains on the **Processes** page, with a process ID (PID), using CPU and memory resources.

## Examples

In MATLAB, use the constructor function to create a IDE handle object for your IDE. The constructor function creates a handle, such as `IDE_Obj`, and starts the IDE.

To get the visibility status of IDE_Obj, enter:

```
isvisible(IDE_Obj)

ans  =
     0
```

Now, change the visibility of the IDE to 1, and check its visibility again.

```
visible(IDE_Obj,1)
isvisible(IDE_Obj)

ans  =
     1
```

If you close MATLAB software while the IDE is not visible, the IDE remains running in the background. To close it, perform either of the following tasks:

- Start MATLAB software. Create a link to the IDE. Use the new link to make the IDE visible. Close the IDE.
- Open Microsoft Windows Task Manager. Click **Processes**. Find and highlight `IDE_Obj_app.exe`. Click **End Task**.

## See Also

`isvisible` | `load`

**Introduced in R2011a**

# write

Write data to processor memory block

## Syntax

```
mem = write(IDE_Obj,address,data)
mem = write(…,datatype)
mem = write(IDE_Obj,…,memorytype)
mem = write(IDE_Obj,…,timeout)
```

## IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

## Description

`mem = write(IDE_Obj,address,data)` writes `data`, a collection of values, to the memory space of the DSP processor referenced by `IDE_Obj`.

The *`data`* argument is a scalar, vector, or array of values to write to the memory of the processor. The block to write begins from the DSP memory location given by the input parameter `address`.

The method writes the data starting from *`address`* without regard to type-alignment boundaries in the DSP. Conversely, the byte ordering of the data type is automatically applied.

---

**Note** You cannot write data to processor memory while the processor is running.

---

The *`address`* argument is a decimal or hexadecimal representation of a memory address in the processor. The full memory address consist of two parts: the start address and the

memory type. The memory type value can be explicitly defined using a numeric vector representation of the address.

Alternatively, the `IDE_Obj` object has a default memory type value which is applied if the memory type value is not explicitly incorporated into the passed address parameter. In DSP processors with only a single memory type, by setting the `IDE_Obj` object memory type value to zero it is possible to specify the addresses using the abbreviated (implied memory type) format.

You provide the *`address`* argument either as a numerical value that is a decimal representation of the DSP memory address, or as a character vector that `write` converts to the decimal representation of the start address. (Refer to function `hex2dec` in the *MATLAB Function Reference* that `read` uses to convert the hexadecimal character vector to a decimal value).

The following examples show how `write` uses the *`address`* argument.

| address Parameter Value | Description |
|---|---|
| `131082` | Decimal address specification. The memory start address is 131082 and memory type is 0. This action is the same as specifying [131082 0]. |
| `[131082 1]` | Decimal address specification. The memory start address is 131082 and memory type is 1. |
| `'2000A'` | Hexadecimal address specification provided as a character vector entry. The memory start address is 131082 (converted to the decimal equivalent) and memory type is 0. |

It is possible to specify *`address`* as cell array, in which case you can use a combination of numbers and character vectors for the start address and memory type values. For example, the following are valid addresses from cell array `myaddress`:

```
myaddress1 myaddress1{1} = 131072; myadddress1{2} = 'Program(PM)
Memory';
```

```
myaddress2 myaddress2{1} = '20000'; myadddress2{2} = 'Program(PM)
Memory';
```

```
myaddress3 myaddress3{1} = 131072; myaddress3{2} = 0;
```

`mem = write(…,`*`datatype`*`)` where the *`datatype`* argument defines the interpretation of the raw values written to DSP memory. The *`datatype`* argument specifies the data format of the raw memory image. The data is written starting from `address` without regard to data type alignment boundaries in the DSP. The byte ordering of the data type is automatically applied. The following MATLAB data types are supported.

| MATLAB Data Type | Description |
| --- | --- |
| `double` | IEEE double-precision floating point value |
| `single` | IEEE single-precision floating point value |
| `uint8` | 8-bit unsigned binary integer value |
| `uint16` | 16-bit unsigned binary integer value |
| `uint32` | 32-bit unsigned binary integer value |
| `int8` | 8-bit signed two's complement integer value |
| `int16` | 16-bit signed two's complement integer value |
| `int32` | 32-bit signed two's complement integer value |

`write` does not coerce data type alignment. Some combinations of *`address`* and *`datatype`* will be difficult for the processor to use.

`mem = write(`*`IDE_Obj`*`,…,`*`memorytype`*`)` adds an optional *`memorytype`* argument. Object `IDE_Obj` has a default memory type value 0 that `write` applies if the memory type value is not explicitly incorporated into the passed address parameter. In processors with only a single memory type, it is possible to specify the addresses using the implied memory type format by setting the value of the `IDE_Obj` `memorytype` property to zero.

`mem = write(`*`IDE_Obj`*`,…,`*`timeout`*`)` adds the optional *`timeout`* argument, which the number of seconds MATLAB waits for the write process to complete. If the *`timeout`* period expires before the write process returns a completion message, MATLAB throws an error and returns. Usually the process works in spite of the error message.

## Using write with VisualDSP++ IDE

Blackfin and SHARC use different memory types. Blackfin processors have one memory type. SHARC processors provide five types. The following table shows the memory types for both processor families.

| String Entry for memorytype | Numerical Entry for memorytype | Processor Support |
|---|---|---|
| 'program(pm) memory' | 0 | Blackfin and SHARC |
| 'data(dm) memory' | 1 | SHARC |
| 'data(dm) short word memory' | 2 | SHARC |
| 'external data(dm) byte memory' | 3 | SHARC |
| 'boot(prom) memory' | 4 | SHARC |

# Examples

## Example with VisualDSP++ IDE

These three syntax examples show how to use `write` in some common ways. In the first example, write an array of 16-bit integers to location [131072 1].

```
write(IDE_Obj,[131072 1],int16([1:100]));
```

Now write a single-precision IEEE floating point value (32-bits) at address 2000A(Hex).

```
write(IDE_Obj,'2000A',single(23.5));
```

For the third example, write a 2-D array of integers in row-major format (standard C programming format) at address 131072 (decimal).

```
mlarr = int32([1:10;101:110]);
write(IDE_Obj,131072,mlarr');
```

# See Also

read | hex2dec

**Introduced in R2011a**

# writemsg

Write messages to specified RTDX channel

---

**Note** Support for `writemsg` on C5000 processors will be removed in a future version.

---

## Syntax

```
data = writemsg(rx,channelname,data)
data = writemsg(rx,channelname,data,timeout)
```

## IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

## Description

`data = writemsg(rx,channelname,data)` writes `data` to a channel associated with `rx`. `channelname` identifies the channel queue, which you must configure for write access beforehand. The messages must be the same type for a single write operation. `writemsg` takes the elements of matrix `data` in column-major order.

In `data = writemsg(rx,channelname,data,timeout)`, the optional argument, `timeout`, limits the time `writemsg` spends transferring messages from the processor. `timeout` is the number of seconds allowed to complete the write operation. You can use `timeout` limit prolonged data transfer operations. If you omit `timeout`, `writemsg` applies the global timeout period defined for the IDE handle object `IDE_Obj`.

`writemsg` supports the following data types: `uint8`, `int16`, `int32`, `single`, and `double`.

## Examples

After you load a program to your processor, configure a link in RTDX for write access and use `writemsg` to write data to the processor. Recall that the program loaded on the processor must define `ichannel` and the channel must be configured for write access.

```
IDE_Obj = ticcs;
rx = rtdx(IDE_Obj);
open(rx,'ichannel','w'); % Could use rx.open('ichannel','w')
enable(rx,'ichannel');
inputdata(1:25);
writemsg(rx,'ichannel',int16(inputdata));
```

As a further illustration, the following code snippet writes the messages in matrix `indata` to the write-enabled channel specified by `ichan`. The code in this example processes only when `ichan` is defined by the program on the processor and enabled for write access.

```
indata = [1 4 7; 2 5 8; 3 6 9];
writemsg(rtdx(IDE_Obj),'ichan',indata);
```

The matrix `indata` is written by column to `ichan`. The preceding function syntax is equivalent to

```
writemsg(rtdx(IDE_Obj),'ichan',[1:9]);
```

## See Also

`readmat` | `readmsg` | `write`

**Introduced in R2011a**

# xmakefilesetup

Configure code generator to produce makefiles

## Syntax

```
xmakefilesetup
```

## IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3
- Texas Instruments Code Composer Studio v4
- Texas Instruments Code Composer Studio v5

## Description

You can configure the code generator to produce and build your software using makefiles. This process can use the software build toolchains, such as compilers and linkers, associated with the preceding list of IDEs. However, the makefile build process does not use the graphical user interface of the IDE directly.

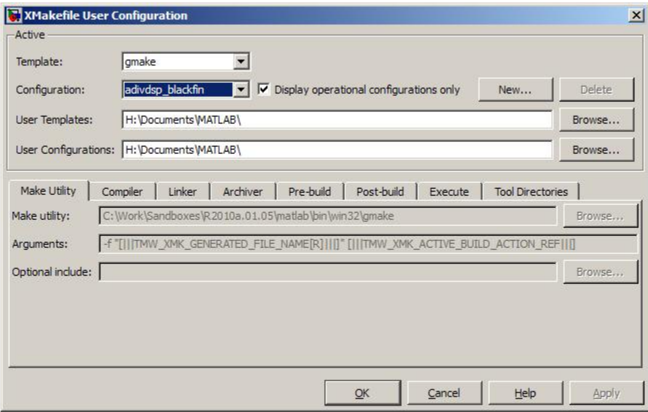
Enter `xmakefilesetup` at the MATLAB command line to configure how to generate makefiles.

Use this function:

- Before you build your software using makefiles for the first time.
- If you change the software build toolchain or processor family.

For more instructions and examples, see "XMakefiles for Software Build Tool Chains".

The `xmakefile` function displays the following dialog box, which prompts you for information about your make utility and software build toolchain.




## See Also

"Build format" on page 13-5 | "Build action" on page 13-6

**Introduced in R2011a**

# coder.mapping.create

Create C code mapping environment for model

## Syntax

```
coder.mapping.create(model)
coder.mapping.create(model,cs)
```

## Description

`coder.mapping.create(model)` creates an environment to configure code generation for data and functions of the specified model. Unless you previously opened your model in Code Perspective mode, before calling other default mapping functions, you must call this function.

`coder.mapping.create(model,cs)` creates a code mapping environment for the specified model that includes code customization settings stored in a configuration set object. The configuration set object can specify memory sections for data and functions and a naming rule for shared utilities. Specify a configuration set object to preserve memory section definitions or shared utility naming rules applied to a model in a version of Embedded Coder prior to R2018a.

## Examples

### Create Environment to Configure Code Mappings for Model

For model `rtwdemo_configdefaults`, create the environment for configuring data and functions for code generation.

```
coder.mapping.create('rtwdemo_configdefaults');
```

After calling this function, use calls to these functions to look up category names, property names, and values that you can use to configure aspects of code generation for model data and functions:

- `coder.mapping.defaults.dataCategories`
- `coder.mapping.defaults.functionCategories`
- `coder.mapping.defaults.allowedProperties`
- `coder.mapping.defaults.allowedValues`

Then, specify category, property, and value combinations in calls to `coder.mapping.defaults.set`.

## Input Arguments

**`model` — Name of model**
character vector

Model file, specified as a character vector. The model must be loaded (for example, by using `load_system`) or open. You can omit the `.slx` file extension.

Example: `'myLoadedModel'`

Data Types: `char`

**`cs` — Configuration set object**
object

Configuration set object from which to import code customization settings for data and functions.

Example: `'cs_basic'`

Data Types: `char`

## See Also

`coder.mapping.defaults.allowedProperties` | `coder.mapping.defaults.allowedValues` | `coder.mapping.defaults.dataCategories` | `coder.mapping.defaults.functionCategories` | `coder.mapping.defaults.get` | `coder.mapping.defaults.set`

### Topics

"Configure Default Code Generation for Categories of Model Data and Functions"

**Introduced in R2018a**

# coder.mapping.defaults.allowedProperties

Return properties for model default mapping category

## Syntax

`properties = coder.mapping.defaults.allowedProperties(model, category)`

## Description

`properties = coder.mapping.defaults.allowedProperties(model, category)` returns a cell array of names for properties that are relevant to `category` for the specified model. Use the property names that the `coder.mapping.defaults.allowedProperties` function returns in subsequent calls to `coder.mapping.defaults.allowedValues` and `coder.mapping.defaults.set`.

## Examples

### Get Properties for Model Default Data Categories

Get a list of the properties for the model default data categories `Inports`, `Outports`, `LocalParameters`, and `InternalData` by using calls to `coder.mapping.defaults.allowedProperties`.

```
coder.mapping.defaults.allowedProperties('rtwdemo_configdefaults', 'Inports')

ans =

  2×1 cell array

    {'StorageClass'}
    {'HeaderFile'  }

coder.mapping.defaults.allowedProperties('rtwdemo_configdefaults', 'Outports')

ans =
```

```
  4×1 cell array

    {'StorageClass'  }
    {'HeaderFile'    }
    {'DefinitionFile'}
    {'Owner'         }

coder.mapping.defaults.allowedProperties('rtwdemo_configdefaults', 'LocalParameters')

ans =

  1×1 cell array

    {'StorageClass'}

coder.mapping.defaults.allowedProperties('rtwdemo_configdefaults', 'InternalData')

ans =

  2×1 cell array

    {'StorageClass' }
    {'MemorySection'}
```

### Get Properties for Model Default Function Categories

Get a list of the properties for the model default function categories `InitializeTerminate` and `Execution` by using calls to `coder.mapping.defaults.allowedProperties`.

```
catData = coder.mapping.defaults.allowedProperties('rtwdemo_configdefaults',...
  'InitializeTerminate');
catFunctions = coder.mapping.defaults.allowedProperties('rtwdemo_configdefaults',...
'Execution');

catData =

  1×1 cell array

    {'FunctionCustomizationTemplate'}

catFunctions =

  1×1 cell array

    {'FunctionCustomizationTemplate'}
```

## Input Arguments

**`model` — Name of model**
character vector

Model file, specified as a character vector. The model must be loaded (for example, by using `load_system`) or open. You can omit the `.slx` file extension.

Example: `'myLoadedModel'`

Data Types: `char`

**`category` — Name of default mapping category**
character vector

Category name, specified as a character vector. To get valid data and function category names, call the functions `coder.mappings.defaults.dataCategories` and `coder.mappings.defaults.functionCategories`.

Example: `'Inports'`

Data Types: `char`

## Output Argument

**`properties` — Names of properties for category**
cell array

Cell array of names for properties of a default category for the specified model.

## See Also

`coder.mapping.defaults.allowedValues` | `coder.mapping.defaults.dataCategories` | `coder.mapping.defaults.functionCategories` | `coder.mapping.defaults.get` | `coder.mapping.defaults.set`

### Topics

"Configure Default Code Generation for Categories of Model Data and Functions"

**Introduced in R2018a**

# coder.mapping.defaults.allowedValues

Return value of property for model default mapping category

## Syntax

`values = coder.mapping.defaults.allowedValues(model,category, property)`

## Description

`values = coder.mapping.defaults.allowedValues(model,category, property)` returns a cell array of values that are relevant to the specified combination of `category` and `property` for the specified model. To set up category, property, and value combinations for a model, use the value names that the function returns in calls to `coder.mapping.defaults.set`.

## Examples

### Get Storage Class Values for Default Data Category Local Parameters

Get the list of values that you can specify for property `StorageClass` for model default data category `LocalParameters` by calling `coder.mapping.defaults.allowedValues`.

```
lclparam_scs = coder.mapping.defaults.allowedValues('rtwdemo_configdefaults', 'LocalParameters',...
  'StorageClass')
lclparam_scs

lclparam_scs =

  15×1 cell array

    {'Default'              }
    {'ExportedGlobal'       }
    {'ImportedExtern'       }
    {'ImportedExternPointer'}
    {'Const'                }
```

```
    {'Volatile'          }
    {'ConstVolatile'     }
    {'Define'            }
    {'ImportedDefine'    }
    {'ExportToFile'      }
    {'ImportFromFile'    }
    {'FileScope'         }
    {'Struct'            }
    {'GetSet'            }
    {'CompilerFlag'      }
```

## Input Arguments

**model — Name of model**
character vector

Model file, specified as a character vector. The model must be loaded (for example, by using `load_system`) or open. You can omit the `.slx` file extension.

Example: `'myLoadedModel'`

Data Types: `char`

**category — Name of default mapping category**
character vector

Category name, specified as a character vector. To get valid data and function category names, call the functions `coder.mappings.defaults.dataCategories` and `coder.mappings.defaults.functionCategories`.

Example: `'LocalParameters'`

Data Types: `char`

**property — Name of property for default mapping category**
character vector

Property name, specified as a character vector. To get valid property names for a default mapping category, call the function `coder.mappings.defaults.allowedProperties`.

Example: `'StorageClass'`

Data Types: `char`

## Output Argument

**`values` — Values for category and property combination**
cell array

Cell array of values that are for a default category and property combination for the specified model.

## See Also

`coder.mapping.defaults.allowedProperties` | `coder.mapping.defaults.dataCategories` | `coder.mapping.defaults.functionCategories` | `coder.mapping.defaults.get` | `coder.mapping.defaults.set`

### Topics

"Configure Default Code Generation for Categories of Model Data and Functions"

**Introduced in R2018a**

# coder.mapping.defaults.dataCategories

Return default mapping categories for model data

## Syntax

`categories = coder.mapping.defaults.dataCategories()`

## Description

`categories = coder.mapping.defaults.dataCategories()` returns a cell array of names for categories of model data elements that you can map to property settings, including a storage class and memory section. The storage class mapped to a category defines how the code generator produces code for that category of data. To set up data category, property, and value combinations for a model, use the category names that the function returns in calls to:

- `coder.mapping.defaults.allowedProperties`
- `coder.mapping.defaults.allowedValues`
- `coder.mapping.defaults.set`

## Example

### Get Model Data Element Categories

Get a list of the available data categories by calling `coder.mapping.defaults.dataCategories`.

```
catData = coder.mapping.defaults.dataCategories()
catData

ans =

  1×8 cell array

  Columns 1 through 4
```

```
    {'Inports'}    {'Outports'}    {'GlobalParameters'}    {'LocalParameters'}

  Columns 5 through 8

    {'SharedLocalData…'}    {'GlobalDataStores'}    {'InternalData'}    {'Constants'}
```

## Output Arguments

**categories — Names of data categoies**
cell array

Cell array of names for default mapping data categories.

## See Also

coder.mapping.defaults.allowedProperties | coder.mapping.defaults.allowedValues | coder.mapping.defaults.dataCategories | coder.mapping.defaults.functionCategories | coder.mapping.defaults.get | coder.mapping.defaults.set

### Topics

"Configure Default Code Generation for Categories of Model Data and Functions"

**Introduced in R2018a**

# coder.mapping.defaults.functionCategories

Return default mapping categories for model functions

## Syntax

`categories = coder.mapping.defaults.functionCategories()`

## Description

`categories = coder.mapping.defaults.functionCategories()` returns a cell array of names for categories of model functions that you can map to property settings, including a function customization template and memory section. The function customization template mapped to a category defines how the code generator produces code for that category of functions. To set up function category, property, and value combinations for a model, use the category names that the function returns in calls to:

- `coder.mapping.defaults.allowedProperties`
- `coder.mapping.defaults.allowedValues`
- `coder.mapping.defaults.set`

## Examples

### Get Model Function Categories

Get a list of the available function categories by calling `coder.mapping.defaults.functionCategories`.

```
catFunc = coder.mapping.defaults.functionCategories();
catFunc

catFunc =

  1×3 cell array

    {'InitializeTerminate'}    {'Execution'}    {'SharedUtility'}
```

## Output Arguments

**`categories` — Names of function categories**
cell array

Cell array of names for default mapping function categories.

## See Also

`coder.mapping.defaults.allowedProperties` | `coder.mapping.defaults.allowedValues` | `coder.mapping.defaults.dataCategories` | `coder.mapping.defaults.get` | `coder.mapping.defaults.set`

### Topics

"Configure Default Code Generation for Categories of Model Data and Functions"

**Introduced in R2018a**

# coder.mapping.defaults.get

Return value of property for model default mapping category

## Syntax

`value = coder.mapping.defaults.get(model,category,property)`

## Description

`value = coder.mapping.defaults.get(model,category,property)` returns the value of a property for a data or function default mapping category for a model. To determine valid category and property combinations, use calls to functions `coder.mapping.defaults.dataCategories`, `coder.mapping.defaults.functionCategories`, and `coder.mapping.defaults.allowedProperties`.

## Examples

### Return Storage Class Setting for Data Imported Into Model

For model `rtwdemo_configuredefaults`, return the storage class that the code generator uses for data imported into the model from external header and definitions files.

Determine the category name to specify for model input data by calling `coder.mapping.defaults.dataCategories`.

```
coder.mapping.defaults.dataCategories()

ans =

  1×8 cell array

  Columns 1 through 4

    {'Inports'}    {'Outports'}    {'GlobalParameters'}    {'LocalParameters'}
```

```
  Columns 5 through 8

    {'SharedLocalData…'}    {'GlobalDataStores'}    {'InternalData'}    {'Constants'}
```

Specify `Inports` as the category name.

Identify properties that you can configure for category **Outports** by calling `coder.mapping.defaults.allowedProperties`.

```
coder.mapping.defaults.allowedProperties('rtwdemo_configdefaults', 'Outports')

ans =

  4×1 cell array

    {'StorageClass'  }
    {'HeaderFile'    }
    {'DefinitionFile'}
    {'Owner'         }
```

Use a call to function `coder.mapping.defaults.get` to return the setting for category **Inports** and property `StorageClass`.

```
coder.mapping.defaults.get('rtwdemo_configdefaults', 'Inports', 'StorageClass')

ans =

    'ImportFromFile'
```

### Return Memory Section Setting for Model Execution Entry-Point Functions

For model `rtwdemo_configuredefaults`, return the memory section that the code generator uses for model execution entry-point functions, such as `step`.

Determine the category name to specify for execution functions by calling `coder.mapping.defaults.functionCategories`.

```
coder.mapping.defaults.functionCategories()

ans =

  1×3 cell array

    {'InitializeTerminate'}    {'Execution'}    {'SharedUtility'}
```

Specify `Execution` as the category name.

Identify properties that you can configure for category `Execution` by calling `coder.mapping.defaults.allowedProperties`.

```
coder.mapping.defaults.allowedProperties('rtwdemo_configdefaults', 'Execution')

ans =

  1×1 cell array

    {'FunctionCustomizationTemplate'}
```

Use a call to function `coder.mapping.defaults.get` to return the setting for category `Execution` and property `FunctionCustomizationTemplate`.

```
coder.mapping.defaults.get('rtwdemo_configdefaults', 'Execution', 'FunctionCustomizationTemplate')

ans =

    'exFastFunction'
```

## Input Arguments

### `model` — Name of model

character vector

Model file, specified as a character vector. The model must be loaded (for example, by using `load_system`) or open. You can omit the `.slx` file extension.

Example: `'myLoadedModel'`

Data Types: `char`

### `category` — Name of default mapping category

character vector

Category name, specified as a character vector. To get valid data and function category names, call the functions `coder.mappings.defaults.dataCategories` and `coder.mappings.defaults.functionCategories`.

Example: `'LocalParameters'`

Data Types: `char`

### `property` — Name of property for default mapping category

character vector

Property name, specified as a character vector. To get valid property names for a default mapping category, call the function `coder.mappings.defaults.allowedProperties`.

Example: `'StorageClass'`

Data Types: `char`

## Output Argument

**`value` — Value of property for default mapping category**
character vector

Character vector that is the setting of the specified default mapping category and property for the specified model.

## See Also

`coder.mapping.defaults.allowedProperties` | `coder.mapping.defaults.allowedValues` | `coder.mapping.defaults.dataCategories` | `coder.mapping.defaults.functionCategories` | `coder.mapping.defaults.set`

### Topics

"Configure Default Code Generation for Categories of Model Data and Functions"

**Introduced in R2018a**

# coder.mapping.defaults.set

Set value for property of model default mapping category

## Syntax

`coder.mapping.defaults.set(model,category,property,value,...)`

## Description

`coder.mapping.defaults.set(model,category,property,value,...)` sets property values for a data or function default mapping category for a model. To determine valid category, property, and value combinations for a model, use calls to:

- `coder.mapping.defaults.dataCategories`
- `coder.mapping.defaults.functionCategories`
- `coder.mapping.defaults.allowedProperties`
- `coder.mapping.defaults.allowedValues`

## Examples

### Configure Default Code Generation Settings for Model Output Data

For model `rtwdemo_configuredefaults`, configure how the code generator handles model output data by default.

Determine the category name to specify for model output data by calling `coder.mapping.defaults.dataCategories`.

```
coder.mapping.defaults.dataCategories()

ans =

  1×8 cell array
```

```
  Columns 1 through 4

    {'Inports'}    {'Outports'}    {'GlobalParameters'}    {'LocalParameters'}

  Columns 5 through 8

    {'SharedLocalData…'}    {'GlobalDataStores'}    {'InternalData'}    {'Constants'}
```

Specify `Outports` as the category name.

Identify properties that you can configure for category `Outports` by calling `coder.mapping.defaults.allowedProperties`.

```
coder.mapping.defaults.allowedProperties('rtwdemo_configdefaults', 'Outports')

ans =

  4×1 cell array

    {'StorageClass'  }
    {'HeaderFile'    }
    {'DefinitionFile'}
    {'Owner'         }
```

For this example, set values for properties `StorageClass`, `HeaderFile`, and `DefinitionFile`.

Look up the values that you can specify for properties `StorageClass`, `HeaderFile`, and `DefinitionFile`.

```
coder.mapping.defaults.allowedValues('rtwdemo_configdefaults', 'Outports', 'StorageClass')

ans =

  10×1 cell array

    {'Default'              }
    {'ExportedGlobal'       }
    {'ImportedExtern'       }
    {'ImportedExternPointer'}
    {'Volatile'             }
    {'ExportToFile'         }
    {'ImportFromFile'       }
    {'AutoScope'            }
    {'Struct'               }
    {'GetSet'               }

coder.mapping.defaults.allowedValues('rtwdemo_configdefaults', 'Outports', 'HeaderFile')

ans =

  0×1 empty cell array

coder.mapping.defaults.allowedValues('rtwdemo_configdefaults', 'Outports', 'DefinitionFile')

ans =
```

```
  0×1 empty cell array
```

Use a call to function `coder.mapping.defaults.set` to configure the default settings. For category `Outports`, set `StorageClass` to `ExportToFile`. Specify `exSysOut.h` and `exSysOut.c` for the header and definition files.

```
coder.mapping.defaults.set('rtwdemo_configdefaults', 'Outports',...
  'Storageclass', 'ExportToFile',...
  'HeaderFile', 'exSysOut.h',...
  'DefinitionFile', 'exSysOut.c')
```

### Configure Default Location in Memory for Storing Code Generated for Model Internal Data

For model `rtwdemo_configuredefaults`, configure the default location in memory for storing code generated for model data elements such as signals, states, and zero crossings.

Determine the category name to specify for model internal data by calling `coder.mapping.defaults.dataCategories`.

```
coder.mapping.defaults.dataCategories()

ans =

  1×8 cell array

  Columns 1 through 4

    {'Inports'}    {'Outports'}    {'GlobalParameters'}    {'LocalParameters'}

  Columns 5 through 8

    {'SharedLocalData…'}    {'GlobalDataStores'}    {'InternalData'}    {'Constants'}
```

Specify `InternalData` as the category name.

Identify properties that you can configure for category `InternalData` by calling `coder.mapping.defaults.allowedProperties`.

```
coder.mapping.defaults.allowedProperties('rtwdemo_configdefaults', 'InternalData')

ans =

  2×1 cell array
```

```
    {'StorageClass' }
    {'MemorySection'}
```

To configure the memory location, set the value for property `MemorySection`.

Look up the values that you can specify for property `MemorySection`.

```
coder.mapping.defaults.allowedValues('rtwdemo_configdefaults', 'InternalData', 'MemorySection')

ans =

  5×1 cell array

    {'None'           }
    {'MemVolatile'    }
    {'internalDataMem'}
    {'functionFastMem'}
    {'functionSlowMem'}
```

Use a call to function `coder.mapping.defaults.set` to configure the default setting. For category `InternalData`, set `MemorySection` to `internalDataMem`.

```
coder.mapping.defaults.set('rtwdemo_configdefaults', 'InternalData',...
  'MemorySection', 'internalDataMem')
```

## Input Arguments

### `model` — Name of model

character vector

Model file, specified as a character vector. The model must be loaded (for example, by using `load_system`) or open. You can omit the `.slx` file extension.

Example: `'myLoadedModel'`

Data Types: `char`

### `category` — Name of default mapping category

character vector

Category name, specified as a character vector. To get valid data and function category names, call the functions `coder.mappings.defaults.dataCategories` and `coder.mappings.defaults.functionCategories`.

Example: `'LocalParameters'`

Data Types: `char`

**`property` — Name of property for default mapping category**
character vector

Property name, specified as a character vector. To get valid property names for a default mapping category, call the function `coder.mappings.defaults.allowedProperties`.

Example: `'StorageClass'`

Data Types: `char`

**`value` — Value of property for default mapping category**
character vector

Property value, specified as a character vector. To get a list of values that you can specify for a category and property combination, call the function `coder.mappings.defaults.allowedValues`.

Example: `'ExportToFile'`

# See Also

`coder.mapping.defaults.allowedProperties` | `coder.mapping.defaults.allowedValues` | `coder.mapping.defaults.dataCategories` | `coder.mapping.defaults.functionCategories` | `coder.mapping.defaults.get`

## Topics

"Configure Default Code Generation for Categories of Model Data and Functions"

**Introduced in R2018a**

# Functions in Simulink Coder— Alphabetical List

# addCompileFlags

Add compiler options to model 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 `myModelBuildInfo` and place the option in the group `OPTS`.

```
myModelBuildInfo = RTW.BuildInfo;
addCompileFlags(myModelBuildInfo,'-O3','OPTS');
```

### Add Compiler Flags to OPT_OPTS Group

Add the compiler options `-Zi` and `-Wall` to the build information `myModelBuildInfo` and place the options in the group `OPT_OPTS`.

```
myModelBuildInfo = RTW.BuildInfo;
addCompileFlags(myModelBuildInfo,'-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 `myModelBuildInfo`. Place the options `-Zi` and `-Wall` in the group `Debug` and the option `-O3` in the group `MemOpt`.

```
myModelBuildInfo = RTW.BuildInfo;
addCompileFlags(myModelBuildInfo,{'-Zi -Wall' '-O3'}, ...
  {'Debug' 'MemOpt'});
```

## Input Arguments

**`buildinfo` — Name of build information object returned by `RTW.BuildInfo`**
object

**`options` — List of compiler options to add to build information**
character vector | array of character vectors

You can specify the *`options`* argument as a character vector or as an array of character vectors. 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

You can specify the *`groups`* argument as a character vector or as an array of character vectors. 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.

Example: `{'Debug' 'MemOpt'}`

## See Also

`addDefines` | `addLinkFlags` | `getCompileFlags`

### Topics

"Customize Post-Code-Generation Build Processing" (Simulink Coder)

**Introduced in R2006a**

# addDefines

Add preprocessor macro definitions to model 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 `myModelBuildInfo` and place the definition in the group `OPTS`.

```
myModelBuildInfo = RTW.BuildInfo;
addDefines(myModelBuildInfo,'-DPRODUCTION','OPTS');
```

### Add Macro Definitions to OPT_OPTS Group

Add the macro definitions `-DPROTO` and `-DDEBUG` to the build information `myModelBuildInfo` and place the definitions in the group `OPT_OPTS`.

```
myModelBuildInfo = RTW.BuildInfo;
addDefines(myModelBuildInfo, ...
  '-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 `myModelBuildInfo`. Place the definitions `-DPROTO` and `-DDEBUG` in the group `Debug` and the definition `-DPRODUCTION` in the group `Release`.

```
myModelBuildInfo = RTW.BuildInfo;
addDefines(myModelBuildInfo, ...
  {'-DPROTO -DDEBUG' '-DPRODUCTION'}, ...
  {'Debug' 'Release'});
```

## Input Arguments

**`buildinfo` — Name of build information object returned by `RTW.BuildInfo`**
object

**`macrodefs` — List of macro definitions to add to build information**
character vector | array of character vectors

You can specify the *`macrodefs`* argument as a character vector or as an array of character vectors. 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

You can specify the *`groups`* argument as a character vector or as an array of character vectors. 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.

Example: `{'Debug' 'Release'}`

## See Also

`addCompileFlags` | `addLinkFlags` | `getDefines`

### Topics

"Customize Post-Code-Generation Build Processing" (Simulink Coder)

**Introduced in R2006a**

# addIncludeFiles

Add include files to model 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.

## Examples

### Add Included File to SysFiles Group

Add the include file `mytypes.h` to the build information `myModelBuildInfo` and place the file in the group `SysFiles`.

```
myModelBuildInfo = RTW.BuildInfo;
addIncludeFiles(myModelBuildInfo, ...
   '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 `myModelBuildInfo`, and place the files in the group `AppFiles`.

```
myModelBuildInfo = RTW.BuildInfo;
addIncludeFiles(myModelBuildInfo, ...
   {'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 `myModelBuildInfo`. Group the files `etc.h` and `etc_private.h` with the character vector `AppFiles` and the file `mytypes.h` with the character vector `SysFiles`.

```
myModelBuildInfo = RTW.BuildInfo;
addIncludeFiles(myModelBuildInfo, ...
{'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 `myModelBuildInfo`, and place the files in the group `HFiles`.

```
myModelBuildInfo = RTW.BuildInfo;
addIncludeFiles(myModelBuildInfo, ...
   '*.h', '/proj/src', 'HFiles');
```

## Input Arguments

**`buildinfo` — Name of build information object returned by `RTW.BuildInfo`**
object

**`filenames` — List of included files to add to build information**
character vector | array of character vectors

You can specify the *filenames* argument as a character vector or as an array of character vectors. 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

You can specify the *paths* argument as a character vector or as an array of character vectors. 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

You can specify the *groups* argument as a character vector or as an array of character vectors. 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" (Simulink Coder)

**Introduced in R2006a**

# addIncludePaths

Add include paths to model 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 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 `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addIncludePaths(myModelBuildInfo,...
   '/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 `myModelBuildInfo` and place the files in the group `etc`.

```
myModelBuildInfo = RTW.BuildInfo;
addIncludePaths(myModelBuildInfo,...
   {'/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 `myModelBuildInfo`. 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`.

```
myModelBuildInfo = RTW.BuildInfo;
addIncludePaths(myModelBuildInfo,...
   {'/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

You can specify the *`paths`* argument as a character vector or as an array of character vectors. 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

You can specify the *`groups`* argument as a character vector or as an array of character vectors. 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" (Simulink Coder)

**Introduced in R2006a**

# addLinkFlags

Add link options to model 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 `myModelBuildInfo` and place the option in the group `OPTS`.

```
myModelBuildInfo = RTW.BuildInfo;
addLinkFlags(myModelBuildInfo,'-T','OPTS');
```

### Add Linker Flags to OPT_OPTS Group

Add the linker options `-MD` and `-Gy` to the build information `myModelBuildInfo` and place the options in the group `OPT_OPTS`.

```
myModelBuildInfo = RTW.BuildInfo;
addLinkFlags(myModelBuildInfo,'-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 `myModelBuildInfo`. Place the options `-MD` and`-Gy` in the group `Debug` and the option `-T` in the group`Temp`.

```
myModelBuildInfo = RTW.BuildInfo;
addLinkFlags(myModelBuildInfo, {'-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

You can specify the *`options`* argument as a character vector or as an array of character vectors. 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

You can specify the *`groups`* argument as a character vector or as an array of character vectors. 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" (Simulink Coder)

**Introduced in R2006a**

# addLinkObjects

Add link objects to model 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 `myModelBuildInfo`. 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.

```
myModelBuildInfo = RTW.BuildInfo;
addLinkObjects(myModelBuildInfo, {'libobj1' 'libobj2'}, ...
   {'/proj/lib/lib1' '/proj/lib/lib2'},[26 10]);
```

### Add Prioritized Link-Only Link Objects to Build Information

Add the linkable objects `libobj1` and `libobj2` to the build information `myModelBuildInfo`. Mark both objects as link-only. Since individual priorities are not specified, the function adds the objects to the vector in the order specified.

```
myModelBuildInfo = RTW.BuildInfo;
addLinkObjects(myModelBuildInfo,{'libobj1' 'libobj2'}, ...
   {'/proj/lib/lib1' '/proj/lib/lib2'},1000, ...
   false,true);
```

### Add Precompiled Link Objects to MyTest Group

Add the linkable objects `libobj1` and `libobj2` to the build information `myModelBuildInfo`. Set the priorities of the objects to 26 and 10, respectively. Mark both objects as precompiled. Group them under the name `MyTest`.

```
myModelBuildInfo = RTW.BuildInfo;
addLinkObjects(myModelBuildInfo,{'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

You can specify the *`linkobjs`* argument as a character vector or as an array of character vectors. 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

You can specify the *`paths`* argument as a character vector or as an array of character vectors. 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.

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 (no precompilation). 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

You can specify the *groups* argument as a character vector or as an array of character vectors. 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" (Simulink Coder)

**Introduced in R2006a**

# addNonBuildFiles

Add nonbuild-related files to model 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 `myModelBuildInfo`, and place the file in the group `DocFiles`.

```
myModelBuildInfo = RTW.BuildInfo;
addNonBuildFiles(myModelBuildInfo, ...
   '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 `myModelBuildInfo`, and place the files in the group `DLLFiles`.

```
myModelBuildInfo = RTW.BuildInfo;
addNonBuildFiles(myModelBuildInfo, ...
   {'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 `myModelBuildInfo`, and place the files in the group `DLLFiles`.

```
myModelBuildInfo = RTW.BuildInfo;
addNonBuildFiles(myModelBuildInfo, ...
   '*.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

You can specify the *`filenames`* argument as a character vector or as an array of character vectors. 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

You can specify the *paths* argument as a character vector or as an array of character vectors. 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

You can specify the *groups* argument as a character vector or as an array of character vectors. 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" (Simulink Coder)

**Introduced in R2008a**

# addSourceFiles

Add source files to model 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 `myModelBuildInfo` and place the file in the group `Drivers`.

```
myModelBuildInfo = RTW.BuildInfo;
addSourceFiles(myModelBuildInfo,'driver.c', ...
   '/proj/src', 'Drivers');
```

### Add Source Files to a Group

Add the source files `test1.c` and `test2.c` to the build information `myModelBuildInfo` and place the files in the group `Tests`.

```
myModelBuildInfo = RTW.BuildInfo;
addSourceFiles(myModelBuildInfo, ...
   {'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 `myModelBuildInfo`. Group the files `test1.c` and `test2.c` with the character vector `Tests`. Group the file `driver.c` with the character vector `Drivers`.

```
myModelBuildInfo = RTW.BuildInfo;
addSourceFiles(myModelBuildInfo, ...
   {'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 `myModelBuildInfo` and place the files in the group `CFiles`.

```
myModelBuildInfo = RTW.BuildInfo;
addIncludeFiles(myModelBuildInfo, ...
   '*.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

You can specify the *`filenames`* argument as a character vector or as an array of character vectors. 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

You can specify the *`paths`* argument as a character vector or as an array of character vectors. 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

You can specify the *`groups`* argument as a character vector or as an array of character vectors. 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” (Simulink Coder)

**Introduced in R2006a**

# addSourcePaths

Add source paths to model 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.

## Examples

### Add Source File Path to Build Information

Add the source path `/etcproj/etc/etc_build` to the build information `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addSourcePaths(myModelBuildInfo, ...
   '/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 `myModelBuildInfo` and place the files in the group `etc`.

```
myModelBuildInfo = RTW.BuildInfo;
addSourcePaths(myModelBuildInfo, ...
   {'/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 `myModelBuildInfo`. 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`.

```
myModelBuildInfo = RTW.BuildInfo;
addSourcePaths(myModelBuildInfo,...
   {'/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

You can specify the *`paths`* argument as a character vector or as an array of character vectors. 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

You can specify the *`groups`* argument as a character vector or as an array of character vectors. 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" (Simulink Coder)

**Introduced in R2006a**

# addTMFTokens

Add template makefile (TMF) tokens to model 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 model. For example, you can call `addTMFTokens` in a post-code-generation command to add a TMF token named `|>CUSTOM_OUTNAME<|` with a token value that specifies an output file name for the build. To achieve the result you want, the TMF must apply an action with the value of `|>CUSTOM_OUTNAME<|`. (See “Examples” on page 2-0 .)

The `addTMFTokens` function adds specified TMF token names and values to the model 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 `myModelBuildInfo`, and place the token in the group `LINK_INFO`.

```
myModelBuildInfo = RTW.BuildInfo;
addTMFTokens(myModelBuildInfo, ...
             '|>CUSTOM_OUTNAME<|', 'foo.exe', 'LINK_INFO');
```

### Apply Build Information as Tokens in TMF Build

In the TMF for the target selected for your model, 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

You can specify the *`tokennames`* argument as a character vector or as an array of character vectors. 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

You can specify the *`tokenvalues`* argument as a character vector or as an array of character vectors. 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

You can specify the *groups* argument as a character vector or as an array of character vectors. 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” (Simulink Coder)

**Introduced in R2009b**

# coder.buildstatus.close

Close build process status window

## Syntax

```
coder.buildstatus.close()
coder.buildstatus.close(model)
coder.buildstatus.close(subsystem)
```

## Description

`coder.buildstatus.close()` closes any open **Build Process Status** windows.

The **Build Process Status** window supports parallel builds of referenced model hierarchies. Do not use the **Build Process Status** window for sequential (non-parallel) builds.

`coder.buildstatus.close(model)` closes the **Build Process Status** window for the `model`.

`coder.buildstatus.close(subsystem)` closes the **Build Process Status** window for the `subsystem`.

## Examples

### Close Build Process Status Windows

Close any open **Build Process Status** windows.

```
coder.buildstatus.close()
```

### Close Build Process Status Window for a Model

After generating code for `rtwdemo_counter`, close the **Build Process Status** window for the model.

```
coder.buildstatus.close('rtwdemo_counter')
```

### Close Build Process Status Window for a Subsystem

Close the **Build Process 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` | `rtwbuild` | `slbuild`

### Topics

“View Build Process Status” (Simulink Coder)

**Introduced in R2018a**

# coder.buildstatus.open

Open build process status window

## Syntax

```
coder.buildstatus.open(model)
coder.buildstatus.open(subsystem)
```

## Description

`coder.buildstatus.open(model)` opens the **Build Process Status** window for the `model`.

The **Build Process Status** window supports parallel builds of referenced model hierarchies. Do not use the **Build Process Status** window for sequential (non-parallel) builds.

If the current working folder is the model build folder and the folder contains information from a previous parallel build, opening the **Build Process 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(subsystem)` opens the **Build Process Status** window for the `subsystem`.

## Examples

### Open Build Process Status Window for a Model

After generating code for `rtwdemo_counter`, open the **Build Process Status** window for the model.

```
coder.buildstatus.open('rtwdemo_counter')
```

### Open Build Process Status Window for a Subsystem

Open the **Build Process Status** window for the subsystem 'Amplifier' in model 'rtwdemo_counter'.

```
coder.buildstatus.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.buildstatus.close | coder.report.open | rtwbuild | slbuild

### Topics

"View Build Process Status" (Simulink Coder)

**Introduced in R2018a**

# coder.codedescriptor.CodeDescriptor class

**Package:** coder

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.

## Construction

`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

| | |
|---|---|
| getAllDataInterfaceTypes | Return all data interface types |
| getAllFunctionInterfaceTypes | Return all function interface types |
| getDataInterfaces | Return information of the specified data interface |
| getDataInterfaceTypes | Return all data interface types in the generated code |
| getFunctionInterfaces | Return information of the specified function interface |
| getFunctionInterfaceTypes | Return all function interface types in the generated code |
| getReferencedModelCodeDescriptor | Return `coder.codedescriptor.CodeDescriptor` object for the specified referenced model |
| getReferencedModelNames | Return names of the referenced models |

## Example

Create a `coder.codedescriptor.CodeDescriptor` object for the required model that is built.

1. Build the model.

```
rtwbuild('rtwdemo_comments')
```

2. Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

```
codeDescObj = coder.getCodeDescriptor('rtwdemo_comments')
```

   The `codeDescObj` with these properties is created:

```
    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)
```

   `allFunctionInterfaceTypes` has these values:

```
    {'Initialize'}
    {'Output'    }
    {'Update'    }
    {'Terminate' }
```

## See Also

getCodeDescriptor | coder.descriptor.DataInterface | coder.descriptor.FunctionInterface

### Topics

"Get Code Description of Generated Code" (Simulink Coder)

**Introduced in R2018a**

# getAllDataInterfaceTypes

**Class:** coder.codedescriptor.CodeDescriptor
**Package:** coder

Return all data interface types

## Syntax

```
allDataInterfaceTypes = getAllDataInterfaceTypes()
```

## Description

`allDataInterfaceTypes = getAllDataInterfaceTypes()` returns a list of all the data interface types available. This list is not specific to any model.

## Output Arguments

**`allDataInterfaceTypes` — All data interface types available**
cell array of character vectors

A list of all the available data interface types.

## Examples

Create a `coder.codedescriptor.CodeDescriptor` object for the required model that is built, then list all the available data interface types.

1. Build the model.

   ```
   rtwbuild('rtwdemo_comments')
   ```
2. Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

   ```
   codeDescObj = coder.getCodeDescriptor('rtwdemo_comments')
   ```

**3** Return a list of all available data interface types.

```
allDataInterfaceTypes = getAllDataInterfaceTypes(codeDescObj)
```

`allDataInterfaceTypes` has these values:

```
{'Inports'         }
{'Outports'        }
{'Parameters'      }
{'GlobalDataStores'}
{'GlobalParameters'}
{'LocalParameters' }
```

In a model, there can be `GlobalParameters` and/or `LocalParameters`. `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" (Simulink Coder)

**Introduced in R2018a**

# getAllFunctionInterfaceTypes

**Class:** coder.codedescriptor.CodeDescriptor
**Package:** coder

Return all function interface types

## Syntax

```
allFunctionInterfaceTypes = getAllFunctionInterfaceTypes()
```

## Description

`allFunctionInterfaceTypes = getAllFunctionInterfaceTypes()` returns a list of all the function interface types available. The returned list is not specific to any model.

## Output Arguments

**`allFunctionInterfaceTypes` — All function interface types available**
cell array of character vectors

A list of all the available function interface types.

## Examples

Create a `coder.codedescriptor.CodeDescriptor` object for the required model which is built, then list all the available function interface types.

1. Build the model.

   ```
   rtwbuild('rtwdemo_comments')
   ```
2. Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

   ```
   codeDescObj = coder.getCodeDescriptor('rtwdemo_comments')
   ```

**3** Return a list of all available function interface types.

```
allFunctionInterfaceTypes = getAllFunctionInterfaceTypes(codeDescObj)
```

`allFunctionInterfaceTypes` has these values:

```
{'Initialize'}
{'Output'    }
{'Update'    }
{'Terminate' }
```

# See Also

`coder.codedescriptor.CodeDescriptor` | `getFunctionInterfaceTypes` | `getFunctionInterfaces` | `getCodeDescriptor` | `coder.descriptor.FunctionInterface`

## Topics

"Get Code Description of Generated Code" (Simulink Coder)
"Configure Code Generation for Model Entry-Point Functions" (Simulink Coder)

**Introduced in R2018a**

# getDataInterfaces

**Class:** coder.codedescriptor.CodeDescriptor
**Package:** coder

Return information of the specified data interface

## Syntax

`dataInterface = getDataInterfaces(dataInterfaceName)`

## Description

`dataInterface = getDataInterfaces(dataInterfaceName)` returns the type of data, SID, graphical name, timing, implementation, and variant information on the data interface that `dataInterfaceName` specifies.

## Input Arguments

**`dataInterfaceName` — Name of data interface**
`Inports` | `Outports` | `Parameters` | `GlobalDataStores` | `GlobalParameters` | `LocalParameters`

`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.DataInterface` object with properties of specified data interface type**
`coder.DataInterface` object | array of `coder.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.

   ```
   rtwbuild('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'      }
       {'GlobalParameters'}
   ```
4. Return properties of Inport blocks in the generated code.

   ```
   dataInterface = getDataInterfaces(codeDescObj, 'Inports')
   ```

   `dataInterface` is an array of `coder.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.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" (Simulink Coder)

**Introduced in R2018a**

# getDataInterfaceTypes

**Class:** coder.codedescriptor.CodeDescriptor
**Package:** coder

Return all data interface types in the generated code

## Syntax

```
dataInterfaceTypes = getDataInterfaceTypes()
```

## Description

`dataInterfaceTypes = getDataInterfaceTypes()` returns a list of all the data interface types in the generated code. To get a list of all the available data interfaces, call `getAllDataInterfaceTypes()`.

## Output Arguments

**`dataInterfaceTypes` — All data interface types in the generated code**
cell array of character vectors

A list of all the data interface types in the generated code.

## Examples

1. Build the model.

   ```
   rtwbuild('rtwdemo_counter')
   ```
2. Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

   ```
   codeDescObj = coder.getCodeDescriptor('rtwdemo_counter')
   ```
3. Return a list of all data interface types in the generated code.

   ```
   dataInterfaceTypes = getDataInterfaceTypes(codeDescObj)
   ```

`dataInterfaceTypes` has these values for model `rtwdemo_counter`:

```
{'Inports'         }
{'Outports'        }
```

# See Also

`coder.codedescriptor.CodeDescriptor` | `getAllDataInterfaceTypes` | `getDataInterfaces` | `getCodeDescriptor`

## Topics

“Get Code Description of Generated Code” (Simulink Coder)

**Introduced in R2018a**

# getFunctionInterfaces

**Class:** coder.codedescriptor.CodeDescriptor
**Package:** coder

Return information of the specified function interface

## Syntax

```
functionInterface = getFunctionInterfaces(functionInterfaceName)
```

## Description

`functionInterface = getFunctionInterfaces(functionInterfaceName)` returns the function prototype, input arguments, return arguments, variant conditions, and timing information of the function interface that `functionInterfaceName` specifies.

## Input Arguments

**`functionInterfaceName` — Name of function interface**
`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.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.

```
rtwbuild('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.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: [1×0 coder.descriptor.DataInterface List]
```

## See Also

coder.codedescriptor.CodeDescriptor | getAllFunctionInterfaceTypes | getFunctionInterfaceTypes | coder.descriptor.FunctionInterface

### Topics

"Get Code Description of Generated Code" (Simulink Coder)

**Introduced in R2018a**

# getFunctionInterfaceTypes

**Class:** coder.codedescriptor.CodeDescriptor
**Package:** coder

Return all function interface types in the generated code

## Syntax

```
functionInterfaceTypes = getFunctionInterfaceTypes()
```

## Description

`functionInterfaceTypes = getFunctionInterfaceTypes()` returns a list of all the function interface types in the generated code. To get a list of all the available function interfaces, call `getAllFunctionInterfaceTypes()`.

## Output Arguments

**`functionInterfaceTypes` — All function interface types in the generated code**
cell array of character vectors

A list of all the data interface types in the generated code.

## Examples

1. Build the model.

   ```
   rtwbuild('rtwdemo_counter')
   ```
2. Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

   ```
   codeDescObj = coder.getCodeDescriptor('rtwdemo_counter')
   ```
3. Return a list of all function interface types in the generated code.

   ```
   functionInterfaceTypes = getFunctionInterfaceTypes(codeDescObj)
   ```

`functionInterfaceTypes` has these values for model `rtwdemo_counter`:

```
{'Output'    }
```

## See Also

`coder.codedescriptor.CodeDescriptor` | `getAllFunctionInterfaceTypes` | `getFunctionInterfaces` | `getCodeDescriptor`

### Topics

"Get Code Description of Generated Code" (Simulink Coder)

**Introduced in R2018a**

# getReferencedModelCodeDescriptor

**Class:** coder.codedescriptor.CodeDescriptor
**Package:** coder

Return `coder.codedescriptor.CodeDescriptor` object for the specified referenced model

## Syntax

```
refCodeDescriptor = getReferencedModelCodeDescriptor(refModelName)
```

## Description

`refCodeDescriptor = getReferencedModelCodeDescriptor(refModelName)` returns the `coder.codedescriptor.CodeDescriptor` object for the referenced model specified in `refModelName`.

## Input Arguments

**`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.

```
rtwbuild('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" (Simulink Coder)

**Introduced in R2018a**

# getReferencedModelNames

**Class:** coder.codedescriptor.CodeDescriptor
**Package:** coder

Return names of the referenced models

## Syntax

`refModels = getReferencedModelNames()`

## Description

`refModels = getReferencedModelNames()` returns a list of referenced models for a `coder.codedescriptor.CodeDescriptor` object.

## Output Arguments

**`refModels` — Names of referenced models**
cell array of character vectors

A list of referenced models.

## Examples

1. Build the model.

   ```
   rtwbuild('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” (Simulink Coder)

**Introduced in R2018a**

# coder.descriptor.DataInterface class

**Package:** coder

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. These are the different types of data interfaces:

- Root-level inports and outports: An interface between the model and external models or systems, for exchanging data.
- Block-specific parameters: Local and Global parameters that describes the data for the block.
- Global Data Store: A repository to store global data that can be written and read.

## Construction

`dataInterface = getDataInterfaces(codeDescObj, dataInterfaceName)` creates a `coder.descriptor.DataInterface` object. `codeDescObj` 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` | `GlobalParameters` | `LocalParameters`

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.

## Example

1. Build the model.

```
rtwbuild('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'      }
    {'GlobalParameters'}
```

4. Return properties of a specified data interface in the generated code.

```
dataInterface = getDataInterfaces(codeDescObj, 'Inports')
```

`dataInterface` is an array of `coder.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.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" (Simulink Coder)

**Introduced in R2018a**

# coder.descriptor.FunctionInterface class

**Package:** coder

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:

- 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`.

## Construction

`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**
`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. If 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. If there is no data passed as an argument to the function, this field is empty.

## Example

1. Build the model.

   ```
   rtwbuild('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)
   ```

`functionInterfaceTypes` consists this:

```
    {'Initialize'}
    {'Output'    }
```

4 Return properties of a specified function interface in the generated code.

```
functionInterface = getFunctionInterfaces(codeDescObj, 'Output')
```

`functionInterface` is a `coder.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` | `getFunctionInterfaces`

### Topics

"Get Code Description of Generated Code" (Simulink Coder)

**Introduced in R2018a**

# 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" (Simulink Coder)

**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.

```
rtwbuild('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.

```
rtwbuild('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.

```
rtwbuild('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 single 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” (Simulink 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” (Simulink 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” (Simulink 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" (Simulink 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" (Simulink 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" (Simulink 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” (Simulink 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 “Static code metrics” (Simulink Coder).

Example: ‘'GenerateCodeMetricsReport','on'’

Data Types: char

## See Also

coder.report.close | coder.report.open

### Topics

“Reports for Code Generation” (Simulink Coder)
“Generate a Code Generation Report” (Simulink Coder)
“Generate Code Generation Report After Build Process” (Simulink Coder)

**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" (Simulink Coder)
"Open Code Generation Report" (Simulink Coder)

**Introduced in R2012a**

# 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" (Simulink Coder).

## 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 with XCP Communication" (Simulink Coder)
"Customize XCP Slave Software" (Simulink Coder)

**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, etc.

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. As the function is thread-safe, you can invoke the function with different sample time IDs in separate threads.

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" (Simulink Coder).

# 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, etc.

**`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 with XCP Communication" (Simulink Coder)
"Customize XCP Slave Software" (Simulink Coder)

**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 with XCP Communication" (Simulink Coder)
"Customize XCP Slave Software" (Simulink Coder)

**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” (Simulink Coder).

## 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, for example, through the `'-tf '` option detected by `extmodeParseArgs()` or `extmodeSetFinalSimulationTime()`, then `finalTime` is an output and the pointer location is updated with the initialized value.

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 with XCP Communication” (Simulink Coder)
“Customize XCP Slave Software” (Simulink Coder)

**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” (Simulink Coder).

## 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” (Simulink Coder).

## 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 with XCP Communication” (Simulink Coder)
“Customize XCP Slave Software” (Simulink Coder)

**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” (Simulink Coder).

## 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 with XCP Communication” (Simulink Coder)
“Customize XCP Slave Software” (Simulink Coder)

**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 with XCP Communication" (Simulink Coder)
"Customize XCP Slave Software" (Simulink Coder)

**Introduced in R2018a**

# extmodeSimulationComplete

Check that external mode simulation is complete

## Syntax

```
simComplete = extmodeSimulationComplete();
```

## Description

`simComplete = extmodeSimulationComplete();` checks, during an external mode simulation, 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” (Simulink Coder).

## 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 with XCP Communication" (Simulink Coder)
"Customize XCP Slave Software" (Simulink Coder)

**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” (Simulink Coder).

## Output Arguments

**`stopRequest` — Stop request**
true | false

`true` if request to stop extenal mode simulation is received. Otherwise, returns `false`.

## See Also

`extmodeBackgroundRun` | `extmodeEvent` | `extmodeGetFinalSimulationTime` | `extmodeInit` | `extmodeParseArgs` | `extmodeReset` | `extmodeSetFinalSimulationTime` | `extmodeSimulationComplete` | `extmodeWaitForHostRequest`

### Topics

"External Mode Simulation with XCP Communication" (Simulink Coder)
"Customize XCP Slave Software" (Simulink Coder)

**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. As the function is a blocking function, use the function during initialization.

## 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” (Simulink Coder).

## Input Arguments

**`timeoutInMicroseconds` — Timeout**
uint32_T

Specifies the timeout value. If the value is set to `EXTMODE_WAIT_FOREVER`, the function waits forever. 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 with XCP Communication" (Simulink Coder)
"Customize XCP Slave Software" (Simulink Coder)

**Introduced in R2018a**

# findBuildArg

Find a specific build argument in model 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 model 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 `myModelBuildInfo`. Then, view the argument identifier and value.

```
load buildInfo.mat
myModelBuildInfo = 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

To get the build argument identifiers from the build information, use the `getBuildArgs` function.

## Output Arguments

**`identifier` — Name of the build argument**
character vector

**`value` — Value of the build argument**
character vector

## See Also

`getBuildArgs`

### Topics

"Customize Post-Code-Generation Build Processing" (Simulink Coder)

**Introduced in R2014a**

# findIncludeFiles

Find and add include (header) files to model 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.

## Examples

### Find and Add Include Files to Build Information

Find include files with file name extension `.h` that are in the build information, `myModelBuildInfo`. Add the full paths for these files to the build information. View the include files from the build information.

```
myModelBuildInfo = RTW.BuildInfo;
addSourcePaths(myModelBuildInfo,{fullfile(pwd,...
   'mycustomheaders')},'myheaders');
findIncludeFiles(myModelBuildInfo);
headerfiles = getIncludeFiles(myModelBuildInfo,true,false);
```

```
>> headerfiles

headerfiles =

    'W:\work\mycustomheaders\myheader.h'
```

## Input Arguments

**`buildinfo` — Name of build information object returned by `RTW.BuildInfo`**
object

**`extPatterns` — Patterns of file name extensions that specify files for the search**
`'*.h'` (default) | cell array

To specify files for the search, the character vectors 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" (Simulink Coder)

**Introduced in R2006b**

# getBuildArgs

Get build arguments from model build information

## Syntax

`[identifiers,values] = getBuildArgs(buildinfo,includeGroupIDs, excludeGroupIDs)`

## Description

`[identifiers,values] = getBuildArgs(buildinfo,includeGroupIDs, excludeGroupIDs)` returns build argument identifiers and values from the 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 model, the build information is available in the `buildInfo.mat` file. This example shows how to get the build arguments from the build information object, `myModelBuildInfo`.

```
load buildInfo.mat
myModelBuildInfo = buildInfo;
[buildArgIds,buildArgValues] = getBuildArgs(myModelBuildInfo);
```

To get the value of a single build argument from the build information, use the `findBuildArg` function.

### View Build Argument Identifiers

To view the build argument identifiers, display `buildArgIds`.

```
>> buildArgIds

buildArgIds =

    'GENERATE_ERT_S_FUNCTION'
    'INCLUDE_MDL_TERMINATE_FCN'
    'COMBINE_OUTPUT_UPDATE_FCNS'
    'MAT_FILE'
    'MULTI_INSTANCE_CODE'
    'INTEGER_CODE'
    'GENERATE_ASAP2'
    'EXT_MODE'
    'EXTMODE_STATIC_ALLOC'
    'EXTMODE_STATIC_ALLOC_SIZE'
    'EXTMODE_TRANSPORT'
    'TMW_EXTMODE_TESTING'
    'MODELLIB'
    'SHARED_SRC'
    'SHARED_SRC_DIR'
    'SHARED_BIN_DIR'
    'SHARED_LIB'
    'MODELREF_LINK_LIBS'
    'RELATIVE_PATH_TO_ANCHOR'
    'MODELREF_TARGET_TYPE'
    'ISPROTECTINGMODEL'
```

### View Build Argument Values

To view the build argument values, display `buildArgValues`.

```
>> buildArgValues

buildArgValues =
```

```
'0'
'1'
'1'
'0'
'0'
'0'
'0'
'0'
'0'
'1000000'
'0'
'0'
'iirlib.lib'
''
''
''
''
''
'..'
'NONE'
'NOTPROTECTING'
```

## 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

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

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

**`values` — values of the build arguments**
cell array

## See Also
`findBuildArg`

### Topics
"Customize Post-Code-Generation Build Processing" (Simulink Coder)

**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" (Simulink Coder)

**Introduced in R2018a**

# getCompileFlags

Get compiler options from model 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 `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addCompileFlags(myModelBuildInfo,{'-Zi -Wall' '-O3'}, ...
   'OPTS');
compflags = getCompileFlags(myModelBuildInfo);
```

```
>> 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 `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addCompileFlags(myModelBuildInfo,{'-Zi -Wall' '-O3'}, ...
   {'Debug' 'MemOpt'});
compflags = getCompileFlags(myModelBuildInfo,'Debug');
```

```
>> compflags

compflags = 

    '-Zi -Wall'
```

### Get Compiler Options with Exclude Group Argument

Get the compiler options stored in the build information `myModelBuildInfo`, except those options with the group name `Debug`.

```
myModelBuildInfo = RTW.BuildInfo;
addCompileFlags(myModelBuildInfo,{'-Zi -Wall' '-O3'}, ...
   {'Debug' 'MemOpt'});
compflags = getCompileFlags(myModelBuildInfo,'','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

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

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

## See Also

`addCompileFlags` | `getDefines` | `getLinkFlags`

### Topics

"Customize Post-Code-Generation Build Processing" (Simulink Coder)

**Introduced in R2006a**

# getDefines

Get preprocessor macro definitions from model 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 `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addDefines(myModelBuildInfo, ...
   {'PROTO=first' '-DDEBUG' 'test' '-dPRODUCTION'},'OPTS');
[defs,names,values] = getDefines(myModelBuildInfo);
```

```
>> 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 `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addDefines(myModelBuildInfo, ...
   {'PROTO=first' '-DDEBUG' 'test' '-dPRODUCTION'}, ...
   {'Debug' 'Debug' 'Debug' 'Release'});
[defs,names,values] = getDefines(myModelBuildInfo,'Debug');
```

```
>> defs

defs =

    '-DPROTO=first'    '-DDEBUG'     '-Dtest'
```

### Get Macro Definitions with Exclude Group Argument

Get the preprocessor macro definitions stored in the build information `myModelBuildInfo`, except those definitions with the group name `Debug`.

```
myModelBuildInfo = RTW.BuildInfo;
addDefines(myModelBuildInfo, ...
   {'PROTO=first' '-DDEBUG' 'test' '-dPRODUCTION'}, ...
   {'Debug' 'Debug' 'Debug' 'Release'});
[defs,names,values] = getDefines(myModelBuildInfo,'','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

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

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

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

**`values` — Values assigned to the macros from the build information**
cell array

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" (Simulink Coder)

**Introduced in R2006a**

# getFullFileList

Get list of files from model build information

## Syntax

```
[fPathNames,names] = getFullFileList(buildinfo,fcase)
```

## Description

`[fPathNames,names] = getFullFileList(buildinfo,fcase)` returns the fully qualified paths and names of all 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.

The `packNGo` function calls `getFullFileList` to return a list of files in the build information before processing files for packaging.

The makefile for the model 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 File List of All Files

After building a model and loading the generated `buildInfo.mat` file, you can list the files stored in a build information variable, `myModelBuildInfo`. This example returns information for the current model and descendants (submodels).

```
myModelBuildInfo = RTW.BuildInfo;
[fPathNames,names] = getFullFileList(myModelBuildInfo);
```

### Get Full File List of Source Files

If you use any of the *`fcase`* options, you limit the listing to the files stored in the `myModelBuildInfo` variable for the current model. This example returns information for the current model only (no descendants or submodels).

```
[fPathNames,names] = getFullFileList(myModelBuildInfo,'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 file list for all 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 all 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

**`names` — File names from the build information**
cell array

## See Also

### Topics
"Customize Post-Code-Generation Build Processing" (Simulink Coder)

**Introduced in R2008a**

# getIncludeFiles

Get include files from model 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 model 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, `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addIncludeFiles(myModelBuildInfo,{'etc.h' 'etc_private.h' ...
   'mytypes.h'},{'/etc/proj/etclib' '/etcproj/etc/etc_build' ...
   '/common/lib'},{'etc' 'etc' 'shared'});
incfiles=getIncludeFiles(myModelBuildInfo,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, `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addIncludeFiles(myModelBuildInfo,{'etc.h' 'etc_private.h' ...
   'mytypes.h'},{'/etc/proj/etclib' '/etcproj/etc/etc_build' ...
   '/common/lib'},{'etc' 'etc' 'shared'});
incfiles = getIncludeFiles(myModelBuildInfo,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

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

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

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 model code.

## See Also

`addIncludeFiles` | `findIncludeFiles` | `getIncludePaths` | `getSourceFiles` | `getSourcePaths` | `updateFilePathsAndExtensions`

### Topics

"Customize Post-Code-Generation Build Processing" (Simulink Coder)

**Introduced in R2006a**

# getIncludePaths

Get include paths from model 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, `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addIncludePaths(myModelBuildInfo,{'/etc/proj/etclib' ...
   '/etcproj/etc/etc_build' '/common/lib'}, ...
   {'etc' 'etc' 'shared'});
incpaths = getIncludePaths(myModelBuildInfo,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, `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addIncludePaths(myModelBuildInfo,{'/etc/proj/etclib' ...
   '/etcproj/etc/etc_build' '/common/lib'}, ...
   {'etc' 'etc' 'shared'});
incpaths = getIncludePaths(myModelBuildInfo,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

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

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

## See Also

`addIncludePaths` | `getIncludeFiles` | `getSourceFiles` | `getSourcePaths`

### Topics

"Customize Post-Code-Generation Build Processing" (Simulink Coder)

**Introduced in R2006a**

# getLinkFlags

Get link options from model 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, `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addLinkFlags(myModelBuildInfo,{'-MD -Gy' '-T'},'OPTS');
linkflags = getLinkFlags(myModelBuildInfo);
```

```
>> 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, `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addLinkFlags(myModelBuildInfo,{'-MD -Gy' '-T'}, ...
   {'Debug' 'MemOpt'});
linkflags = getLinkFlags(myModelBuildInfo,{'Debug'});
```

```
>> linkflags

linkflags = 

    '-MD -Gy'
```

### Get Linker Options with Exclude Group Argument

Get the linker options from the build information `myModelBuildInfo`, except those options with the group name `Debug`.

```
myModelBuildInfo = RTW.BuildInfo;
addLinkFlags(myModelBuildInfo,{'-MD -Gy' '-T'}, ...
   {'Debug' 'MemOpt'});
linkflags = getLinkFlags(myModelBuildInfo,'',{'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

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

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

## See Also

`addLinkFlags` | `getCompileFlags` | `getDefines`

### Topics

"Customize Post-Code-Generation Build Processing" (Simulink Coder)

**Introduced in R2006a**

# getNonBuildFiles

Get nonbuild-related files from model 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 model 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, `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addNonBuildFiles(myModelBuildInfo,{'readme.txt' 'myutility1.dll' ...
   'myutility2.dll'});
nonbuildfiles = getNonBuildFiles(myModelBuildInfo,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

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

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

# See Also

`addNonBuildFiles`

## Topics

"Customize Post-Code-Generation Build Processing" (Simulink Coder)

**Introduced in R2008a**

# getSourceFiles

Get source files from model 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 model 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, `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addSourceFiles(myModelBuildInfo, ...
   {'test1.c' 'test2.c' 'driver.c'},'', ...
   {'Tests' 'Tests' 'Drivers'});
srcfiles = getSourceFiles(myModelBuildInfo,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, `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addSourceFiles(myModelBuildInfo,{'test1.c' 'test2.c'...
   'driver.c'}, {'/proj/test1' '/proj/test2'...
   '/drivers/src'}, {'tests', 'tests', 'drivers'});
incfiles = getSourceFiles(myModelBuildInfo,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

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

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

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 model code.

## See Also

`addSourceFiles` | `getIncludeFiles` | `getIncludePaths` | `getSourcePaths` | `updateFilePathsAndExtensions`

### Topics

"Customize Post-Code-Generation Build Processing" (Simulink Coder)

**Introduced in R2006a**

# getSourcePaths

Get source paths from model 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 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, `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addSourcePaths(myModelBuildInfo,{'/proj/test1' ...
   '/proj/test2' '/drivers/src'}, {'tests' 'tests' ...
   'drivers'});
srcpaths = getSourcePaths(myModelBuildInfo,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, `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addSourcePaths(myModelBuildInfo,{'/proj/test1' ...
   '/proj/test2' '/drivers/src'}, {'tests' 'tests' ...
   'drivers'});
srcpaths = getSourcePaths(myModelBuildInfo,true,'tests');
```

```
>> srcpaths

srcpaths =

     '\proj\test1'    '\proj\test2'
```

### Get Source Paths from Build Information

Get a source path from the build information, `myModelBuildInfo`. 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`.

```
myModelBuildInfo = RTW.BuildInfo;
addSourcePaths(myModelBuildInfo, fullfile(matlabroot, ...
   'rtw', 'c', 'src'));
srcpaths = getSourcePaths(myModelBuildInfo,false);
```

```
>> srcpaths{:}

ans =

$(MATLAB_ROOT)\rtw\c\src

>> srcpaths = getSourcePaths(myModelBuildInfo,true);
```

```
>> srcpaths{:}

ans =

\\myserver\myworkspace\matlab\rtw\c\src
```

## 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**
`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 paths to include in the return from the function**
cell array

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

To use the *`excludeGroups`* argument, view available groups with `myGroups = getGroups(buildInfo)`.

Example: `''`

## Output Arguments

**`srcpaths` — Paths of source files from the build information**
cell array

## See Also

`addSourcePaths` | `getIncludeFiles` | `getIncludePaths` | `getSourceFiles`

### Topics

"Customize Post-Code-Generation Build Processing" (Simulink Coder)

**Introduced in R2006a**

# model_initialize

Initialization entry-point function in generated code for Simulink model

## Syntax

```
void model_initialize(void)
```

## Calling Interfaces

The calling interface generated for this function differs depending on the value of the model parameter **Code interface packaging** (Simulink Coder):

- `C++ class` (default for C++ language) — Generated function is encapsulated into a C ++ class method. Required model data is encapsulated into C++ class attributes.
- `Nonreusable function` (default for C language) — Generated function passes `(void)`. Model data structures are statically allocated, global, and accessed directly in the model code.
- `Reusable function` — Generated function passes the real-time model data structure, by reference, as an input argument. The real-time model data structure is exported with the *`model`*`.h` header file.

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

  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 the **Use dynamic memory allocation for model initialization** parameter to control whether an allocation function is generated.

  - When set, you can restart code generated from the model from a single execution instance. The sequence of function calls from the `main.c` is `allocfcn`, `model_init`, `model_term`, `allocfcn`, `model_init`, `model_term`.

- When cleared,

---

**Note** If you have an Embedded Coder license, for `Nonreusable function` code interface packaging, you can use the **Configure Model Functions** button on the **Interface** pane of the Configuration Parameters dialog box. For more information, see “Customize Generated C Function Interfaces”.

---

## Description

The generated *`model`*`_initialize` function contains initialization code for a Simulink model and should be called once at the start of your application code.

Do not use the *`model`*`_initialize` function to reset the real-time model data structure (rtM).

## See Also

`model_step` | `model_terminate`

### Topics

“Configure Code Generation for Model Entry-Point Functions” (Simulink Coder)
“Generate Code That Responds to Initialize, Reset, and Terminate Events” (Simulink Coder)

**Introduced before R2006a**

# model_step

Step routine entry point in generated code for Simulink model

## Syntax

```
void model_step(void)
```

```
void model_stepN(void)
```

## Calling Interfaces

The `model_step` default function prototype varies depending on the **Treat each discrete rate as a separate task** (Simulink) (`EnableMultiTasking`) parameter specified for the model:

| Parameter Value | Function Prototype |
|---|---|
| `Off`<br>(single rate or multirate) | `void model_step(void);` |
| `On`<br>(multirate) | `void model_stepN (void);`<br>(*N* is a task identifier) |

The calling interface generated for this function also differs depending on the value of the model parameter **Code interface packaging** (Simulink Coder):

- `C++ class` (default for C++ language) — Generated function is encapsulated into a C ++ class method. Required model data is encapsulated into C++ class attributes.
- `Nonreusable function` (default for C language) — Generated function passes `(void)`. Model data structures are statically allocated, global, and accessed directly in the model code.
- `Reusable function` — Generated function passes the real-time model data structure, by reference, as an input argument. The real-time model data structure is exported with the `model.h` header file.

  For an ERT-based model, you can use the **Pass root-level I/O as** parameter to control how root-level input and output arguments are passed to the function. They can be

included in the real-time model data structure, passed as individual arguments, or passed as references to an input structure and an output structure.

---

**Note** If you have an Embedded Coder license:

- For `Nonreusable function` code interface packaging, you can use the **Configure Model Functions** button on the **Interface** pane of the Configuration Parameters dialog box. For more information, see "Customize Generated C Function Interfaces".
- For `C++ class` code interface packaging, you can use the **Configure C++ Class Interface** button and related controls on the **Interface** pane of the Configuration Parameters dialog box. For more information, see "Customize Generated C++ Class Interfaces".

---

## Description

The generated *model*`_step` function contains the output and update code for the blocks in a Simulink model. The *model*`_step` function computes the current value of the blocks. If logging is enabled, *model*`_step` updates logging variables. If the model's stop time is finite, *model*`_step` signals the end of execution when the current time equals the stop time.

Under the following conditions, *model*`_step` does not check the current time against the stop time:

- The model's stop time is set to `inf`.
- Logging is disabled.
- The **Terminate function required** option is not selected.

Therefore, if one or more of these conditions are true, the program runs indefinitely.

For a GRT or ERT-based model, the software generates a *model*`_step` function when the **Single output/update function** configuration option is selected (the default) in the Configuration Parameters dialog box.

*model*`_step` is designed to be called at interrupt level from `rt_OneStep`, which is assumed to be invoked as a timer ISR. `rt_OneStep` calls *model*`_step` to execute processing for one clock period of the model. For a description of how calls to

*model*_step are generated and scheduled, see “rt_OneStep and Scheduling Considerations”.

---

**Note** If the **Single output/update function** configuration option is not selected, the software generates the following model entry point functions in place of *model*_step:

- *model*_output: Contains the output code for the blocks in the model
- *model*_update: Contains the update code for the blocks in the model

---

## See Also

model_initialize | model_terminate

### Topics

“Configure Code Generation for Model Entry-Point Functions” (Simulink Coder)

**Introduced before R2006a**

# model_terminate

Termination entry point in generated code for Simulink model

## Syntax

```
void model_terminate(void)
```

## Calling Interfaces

The calling interface generated for this function also differs depending on the value of the model parameter **Code interface packaging** (Simulink Coder):

- `C++ class` (default for C++ language) — Generated function is encapsulated into a C ++ class method. Required model data is encapsulated into C++ class attributes.
- `Nonreusable function` (default for C language) — Generated function passes `(void)`. Model data structures are statically allocated, global, and accessed directly in the model code.
- `Reusable function` — Generated function passes the real-time model data structure, by reference, as an input argument. The real-time model data structure is exported with the *`model`*`.h` header file.

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

## Description

The generated *`model`*`_terminate` function contains the termination code for a Simulink model and should be called as part of system shutdown.

When *`model`*`_terminate` is called, blocks that have a terminate function execute their terminate code. If logging is enabled, *`model`*`_terminate` ends data logging.

The *model*_terminate function should be called only once.

For an ERT-based model, the code generator produces the *model*_terminate function for a model when the **Terminate function required** configuration option is selected (the default) in the Configuration Parameters dialog box. If your application runs indefinitely, you do not need the *model*_terminate function. To suppress the function, clear the **Terminate function required** configuration option in the Configuration Parameters dialog box.

# See Also

model_initialize | model_step

## Topics

"Configure Code Generation for Model Entry-Point Functions" (Simulink Coder)
"Generate Code That Responds to Initialize, Reset, and Terminate Events" (Simulink Coder)

**Introduced before R2006a**

# packNGo

Package model code in zip file for relocation

## Syntax

`packNGo(buildInfo,{propVals})`

## Description

`packNGo(buildInfo,{propVals})` packages the code files in a compressed zip file so you can relocate, unpack, and rebuild them in another development environment. The list of `{propVals}` name-value pairs is optional.

The types of code files in the zip file include:

- Source files (for example, `.c` and `.cpp` files)
- Header files (for example, `.h` and `.hpp` files)
- MAT-file that contains the model 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.

You can 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 named *`model`*`.zip`. You can tailor the output by specifying property name and value pairs as described.

After relocating the zip file, use a standard zip utility to unpack the compressed file.

The `packNGo` function potentially can modify the build information passed in the first `packNGo` argument. As part of packaging model code, `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.

## Examples

### Configure packNGo from Code Generation UI

If you configure zip file packaging from the code generation UI, 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. The code generator outputs the zip file at the end of the build process.

### Configure packNGo from set_param

If you configure zip file packaging with `set_param`, the code generator uses `packNGo` to output a zip file during the build process.

This example shows how to apply `set_param` to configure zip file packaging for model `zingbit` in the file `zingbit.zip` as a flat folder structure.

```
set_param('zingbit','PostCodeGenCommand', ...
   'packNGo(buildInfo);');
```

### Run packNGo from Command Window

After a model build, you can run packNGo from the Command Window. This example shows how to apply `packNGo` for zip file packaging of the code files for model `zingbit` in the file `portzingbit.zip` and maintain the relative file hierarchy.

1. Change folders to the model build folder.

2 Load the `buildInfo` object file that describes the model build.
3 Run `packNGo` with property settings for `packType` and `fileName`.

```
cd zingbit_grt_rtw;
load buildInfo.mat
packNGo(buildInfo,{'packType', 'hierarchical', ...
   'fileName','portzingbit'});
```

# Input Arguments

**`buildInfo` — Provides model build information for the zip file**
buildInfo

During model builds, the code generator places the `buildInfo.mat` file in the build folder. This file provides build information that `packNGo` uses to produce the zip file.

**`propVals` — Property values as name-value pairs select options for producing the zip file**
name-value pairs

See “Name-Value Pair Arguments” (Simulink Coder).

## 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 single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `{'packType','flat','nestedZipFiles',true}`

**`packType` — Determines whether the primary zip file contains secondary zip files or folders**
`'flat'` (default) | `'hierarchical'`

If `'flat'`, package the model code files in a zip file as a single, flat folder.

If `'hierarchical'`, package the model code files hierarchically in a primary zip file.

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 build 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 build folder and *`matlabroot`*.

Example: `{'nestedZipFiles',true}`

**`fileName` — Specifies a file name for the primary zip file**
`'`*`model`*`.zip'` (default) | `'`*`zipName`*`'`

By default, the function packages the files in a zip file named *`model`*`.zip` and places the zip file in the build folder.

Example: `{'fileName','`*`model`*`.zip'}`

**`minimalHeaders` — Selects whether only to include 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`, error out 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}`

# See Also

## Topics
"Customize Post-Code-Generation Build Processing" (Simulink Coder)
"Relocate Code to Another Development Environment" (Simulink Coder)
"packNGo Function Limitations" (Simulink Coder)

**Introduced in R2006b**

# 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` |

| Field | Description |
|---|---|
| `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 |
| `BlockPath` | Block path for a block parameter. This field is empty for model parameters. |
| `CAPIIdx` | Internal use only |

It is recommended that you 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" (Simulink Coder)
"Update Diagram and Run Simulation" (Simulink)
"Default parameter behavior" (Simulink Coder)
"Block Creation" (Simulink)
"Tune Parameters" (Simulink Coder)

**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" (Simulink Coder)
"Update Diagram and Run Simulation" (Simulink)
"Default parameter behavior" (Simulink Coder)
"Block Creation" (Simulink)
"Tune Parameters" (Simulink Coder)

**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” (Simulink Coder)
“Recompile Precompiled Libraries” (Simulink Coder)
“Choose Build Approach and Configure Build Process” (Simulink Coder)
“Use rtwmakecfg.m API to Customize Generated Makefiles” (Simulink Coder)

**Introduced in R2009b**

# rtwbuild

Build generated code from a model

## Syntax

```
rtwbuild(model)
rtwbuild(model,name,value)

rtwbuild(subsystem)

rtwbuild(subsystem,'Mode','ExportFunctionCalls')
blockHandle = rtwbuild(subsystem,'Mode','ExportFunctionCalls')
rtwbuild(subsystem,'Mode','ExportFunctionCalls,
'ExportFunctionInitializeFunctionName', fcnname)
```

## 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.

Do not use `rtwbuild`, `rtwrebuild`, or `slbuild` commands with parallel language features (Parallel Computing Toolbox) (for example, within a `parfor` or `spmd` loop). For information about parallel builds of referenced models, see “Reduce Build Time for Referenced Models” (Simulink Coder).

`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 **Configuration Parameters** > **Code Generation** > **Verification** > **Advanced parameters** > **Create block** is set to `SIL` and if you have Embedded Coder. You can then use the SIL block for SIL verification testing.

`rtwbuild(subsystem,'Mode','ExportFunctionCalls, 'ExportFunctionInitializeFunctionName', fcnname)` names the exported initialization function, specified as a character vector, for the specified subsystem.

# 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 | Other Files |
|---|---|---|---|
| rtwdemo_rtwintro.c<br>rtwdemo_rtwintro.h<br>rtwdemo_rtwintro_private.h<br>rtwdemo_rtwintrotypes.h | rtGetInf.c<br>rtGetInf.h<br>rtGetNaN.c<br>rtGetNaN.h<br>rt_nonfinite.c<br>rt_nonfinite.h<br>rtwtypes.h<br>multiword_types.h<br>builtin_typeid_types.h | rtmodel.h | rt_logging.c |

If the following model configuration parameters settings apply, the code generator produces additional results.

| Parameter Setting | Results |
|---|---|
| **Code Generation** > **Generate code only** pane 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)
```

### Display Error Messages in Diagnostic Viewer

Introduce an error to model rtwdemo_mdlreftop and save the model as rtwdemo_mdlreftop_witherr. Display build error messages in the Diagnostic Viewer and in the Command Window while generating code and building an executable image for model rtwdemo_mdlreftop_witherr.

```
rtwbuild('rtwdemo_mdlreftop_witherr', ...
    'OkayToPushNags',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 | Other Files |
|---|---|---|---|
| Amplifier.c<br>Amplifier.h<br>Amplifier_private.h<br>Amplifier_types.h | rtGetInf.c<br>rtGetInf.h<br>rtGetNaN.c<br>rtGetNaN.h<br>rt_nonfinite.c<br>rt_nonfinite.h<br>rtwtypes.h<br>multiword_types.h<br>builtin_typeid_types.h | rtmodel.h | rt_logging.c |

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** pane 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 **Configuration Parameters** > **Code Generation** > **Verification** > **Advanced parameters** > **Create block** 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 Process Status Window

Display build information in the Build Process 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` — Name-value pairs select options for the build process

name-value pairs

## 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 single 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 (Simulink), such as `slprj` or the generated model code folder.

### `OkayToPushNags` — Display build error messages in Diagnostic Viewer

`false` (default) | `true`

Display error messages from the build in Diagnostic Viewer, specified as `true` or `false`.

| Action | Specify |
| --- | --- |
| Display build error messages in the Diagnostic Viewer and in the Command Window | `true` |

| Action | Specify |
| --- | --- |
| Display build error messages in the Command Window only | `false` |

**`generateCodeOnly` — Specify code generation versus an executable build**
`false` (default) | `true`

Specify code generation versus an executable build, specified as `true` or `false`.

| Action | Specify |
| --- | --- |
| Specify code generation (same operation as value `'on'` for `GenCodeOnly` parameter) | `true` |
| Specify executable build (same operation as value `'off'` for `GenCodeOnly` parameter) | `false` |

**`Mode` — (for subsystem builds only) Direct code generator to export function calls**
`'ExportFunctionCalls'` (default)

If you have Embedded Coder, generates code from `subsystem` that includes function calls that you can export to external application code.

**`OpenBuildStatusAutomatically` — Display build information in the Build Process Status Window**
`false` (default) | `true`

Display build information in the **Build Process Status** window, specified as `true` or `false`. For more information about using the status window, see “View Build Process Status” (Simulink Coder).

The **Build Process Status** window support parallel builds of referenced model hierarchies. Do not use the **Build Process Status** window for sequential (non-parallel) builds.

| Action | Specify |
| --- | --- |
| Display build information in the Build Process 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` |

## 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.
- **Configuration Parameters** > **Code Generation** > **Verification** > **Advanced parameters** > **Create block** 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.

## See Also

`coder.buildstatus.close` | `coder.buildstatus.open` | `rtwrebuild` | `slbuild`

### Topics

"Build and Run a Program" (Simulink Coder)
"Choose Build Approach and Configure Build Process" (Simulink Coder)
"Control Regeneration of Top Model Code" (Simulink Coder)

“Generate Component Source Code for Export to External Code Base”
“Software-in-the-Loop Simulation”

**Introduced in R2009a**

# 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

`rtwbuild`

## Topics

“Working Folder” (Simulink Coder)
“Manage Build Process Folders” (Simulink Coder)

**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 invokes the makefile in the build folder. If the current working folder is not the build folder, the function exits with an error.

`rtwrebuild` invokes the makefile generated during the previous build 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, `rtwrebuild` invokes the makefile for referenced model code recursively before recompiling the top model.

Do not use `rtwbuild`, `rtwrebuild`, or `slbuild` commands with parallel language features (Parallel Computing Toolbox) (for example, within a `parfor` or `spmd` loop). For information about parallel builds of referenced models, see “Reduce Build Time for Referenced Models” (Simulink Coder).

`rtwrebuild(model)` assumes that the current working folder is one level above the build folder and invokes the makefile in the build folder. 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 and invokes the makefile in the build folder. 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

Invoke the makefile 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

Invoke the previously generated makefile in the current working folder (build folder) `C:\work\mymodel_grt_rtw`.

```
rtwrebuild()
```

#### Rebuild Code from Parent Folder of Build Folder

When the current working folder is one level above the build folder, invoke the makefile and recompile code.

```
rtwrebuild('mymodel')
```

#### Rebuild Code from Any Folder

Invoke the makefile and recompile code from any 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

`rtwbuild` | `slbuild`

### Topics

"Rebuild a Model" (Simulink Coder)

**Introduced in R2009a**

# rtwreport

Create generated code report for model with Simulink Report Generator

## 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 (Simulink). If the software cannot find the `folder`, an error occurs and code is not generated.

## Examples

### Create Report Documenting Generated Code

Create a report for model `rtwdemo_secondOrderSystem`:

```
rtwreport('rtwdemo_secondOrderSystem');
```

### 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`

## See Also

### Topics

"Document Generated Code with Simulink Report Generator" (Simulink Coder)
`Import Generated Code`
"Working with the Report Explorer" (Simulink Report Generator)
`Code Generation Summary`

**Introduced in R2007a**

# rtwtrace

Trace a block to generated code in HTML code generation report

## Syntax

```
rtwtrace('blockpath')
rtwtrace('Simulink_identifier')
rtwtrace('blockpath', 'hdl')
rtwtrace('blockpath', 'plc')
```

## Description

`rtwtrace('blockpath')` opens an HTML 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** (Simulink 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 an HTML 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 “Locate Diagram Components Using Simulink Identifiers” (Simulink).

`rtwtrace('blockpath', 'hdl')` opens an HTML 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 an HTML 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.

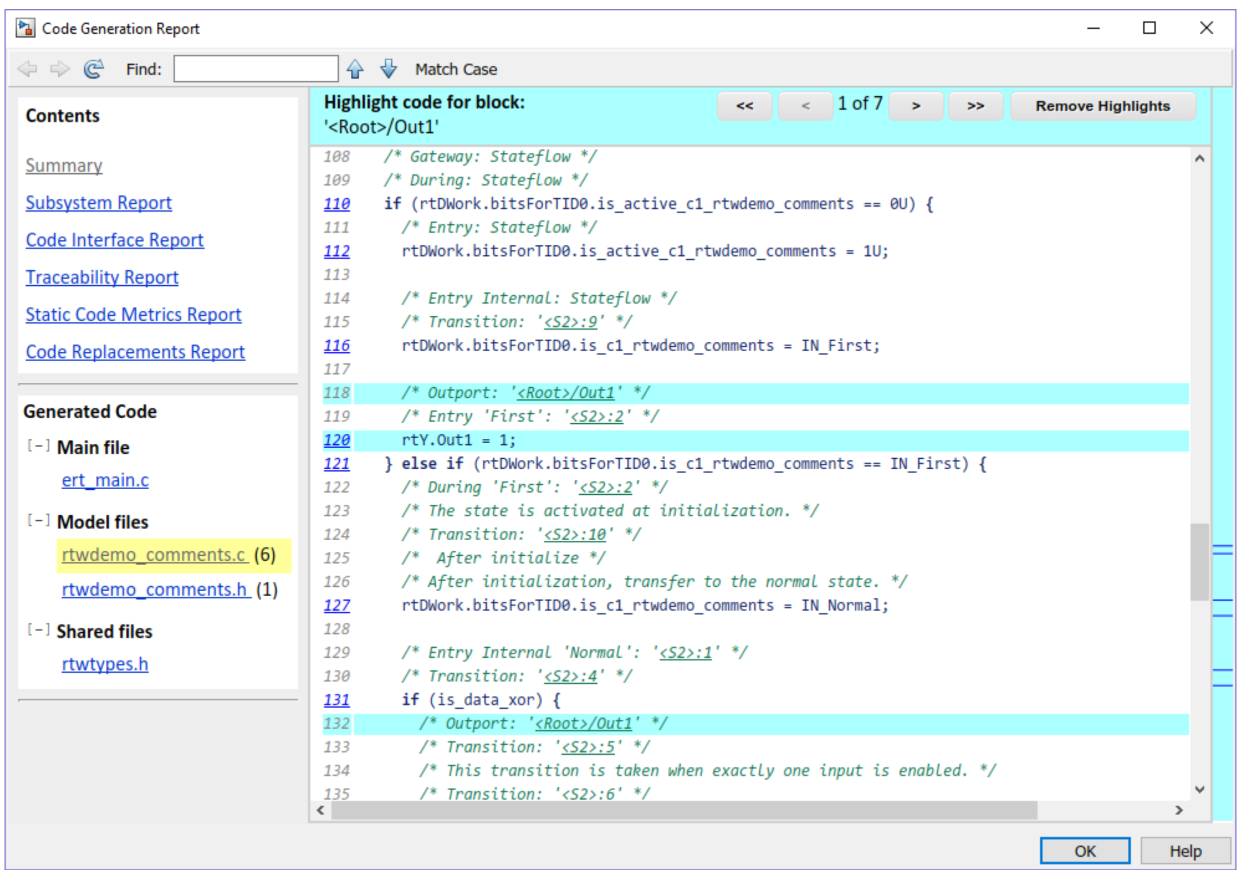
## Examples

### Display Generated Code for a Block

Display the generated code for block `Out1` in the model `rtwdemo_comments` in HTML 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

"Model-to-Code Traceability"
"Model-to-code" (Simulink Coder)

**Introduced in R2009b**

# setTargetProvidesMain

Disable inclusion of code generator provided (generated or static) `main.c` source file during model 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,modelName,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, modelName, 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 model. When you indicate that the target provides `main.c`, the model 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" (Simulink Coder)

**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 (Simulink) – `CacheFolder`
- Code generation folder (Simulink) – `CodeGenFolder`
- Code generation folder structure (Simulink) – `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 single 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

Specifiy 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. The call produces an error.

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?” (MATLAB) and “Files and Folders that MATLAB Accesses” (MATLAB).

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” (Simulink) | “Code generation folder” (Simulink) | Code generation folder structure (Simulink)

### Topics

“Manage Build Process Folders” (Simulink Coder)
“Reuse Simulation Builds for Faster Simulations” (Simulink)

**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 single 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`

### `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'` | `'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.
- `'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'`: 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'`: All code in the build folder is visible. All headers 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.

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`

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.

### 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`.

- Protected Models for Model Reference
- “Test the Protected Model” (Simulink Coder)
- “Package a Protected Model” (Simulink Coder)
- “Specify Custom Obfuscator for Protected Model” (Simulink Coder)
- “Configure and Run SIL Simulation”
- “Define Callbacks for Protected Models” (Simulink Coder)

## 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 model to be protected.

## 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 single 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.

**`Harness` — Option to create a harness model**
false (default) | true

Option to create a harness model, specified as a Boolean value.

Example: `'Harness',true`

**`Mode` — Model protection mode**
`'Normal'` (default) | `'Accelerator'` | `'CodeGeneration'` | `'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.
- `'ViewOnly'`: Turns off Simulate and Generate code functionality modes. 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'`

Applies only if the system target file (`SystemTargetFile`) is set to an ERT based system target file (for example, `ert.tlc`). Requires Embedded Coder license.

Specify 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'`

**`ObfuscateCode` — Option to obfuscate generated code**
true (default) | false

Option to obfuscate generated code, specified as a Boolean value. Applicable only when code generation during protection is enabled.

Example: `'ObfuscateCode',true`

**`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`

**`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'`: 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'`: All code in the build folder is visible. All headers referenced by the code are also visible.

Example: `'OutputFormat','AllReferencedHeaders'`

#### `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`

#### `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`

Example: `'Encrypt',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. 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 a modifiable protected model**
false (default) | true

Option to create a modifiable protected model, specified as a Boolean value. To use this option:

- Add a password for modification 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`

**`Callbacks` — Option to specify protected model callbacks**
cell array

Option to specify callbacks for a protected model, specified as a cell array of `Simulink.ProtectedModel.Callback` objects.

Example: `'Callbacks',{pmcallback_sim, pmcallback_cg}`

## 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 model being protected, returned as a cell array.

The cell array can also include variables that the protected model does not use.

## Alternatives

"Create a Protected Model" (Simulink Coder)

## See Also

`Simulink.ModelReference.ProtectedModel.clearPasswords` |
`Simulink.ModelReference.ProtectedModel.clearPasswordsForModel` |
`Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration` |
`Simulink.ModelReference.ProtectedModel.setPasswordForModify` |

`Simulink.ModelReference.ProtectedModel.setPasswordForSimulation` | `Simulink.ModelReference.ProtectedModel.setPasswordForView` | `Simulink.ModelReference.modifyProtectedModel`

## Topics

Protected Models for Model Reference
"Test the Protected Model" (Simulink Coder)
"Package a Protected Model" (Simulink Coder)
"Specify Custom Obfuscator for Protected Model" (Simulink Coder)
"Configure and Run SIL Simulation"
"Define Callbacks for Protected Models" (Simulink Coder)
"Simulate Protected Models from Third Parties" (Simulink)
"Protect Models for Third-Party Use" (Simulink Coder)
"Protected Model File" (Simulink Coder)
"Harness Model" (Simulink Coder)
"Protected Model Report" (Simulink Coder)
"Code Generation Support in Protected Models" (Simulink Coder)
"Code Interfaces for SIL and PIL"

**Introduced in R2012b**

# Simulink.ModelReference.ProtectedModel.clearPasswords

Clear all cached passwords for protected models

## Syntax

`Simulink.ModelReference.ProtectedModel.clearPasswords()`

## Description

`Simulink.ModelReference.ProtectedModel.clearPasswords()` clears all 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 all 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 for Third-Party Use" (Simulink Coder)

**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 all 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 all 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 for Third-Party Use" (Simulink Coder)

**Introduced in R2014b**

# Simulink.ModelReference.ProtectedModel.HookInfo class

**Package:** Simulink.ModelReference.ProtectedModel

Represent files and exported symbols generated by creation of protected model

## Description

Specifies information about files and symbols generated when creating a protected model. The creator of a protected model can use this information for postprocessing of the generated files prior to packaging. Information includes:

- List of source code files (`*.c`, `*.h`, `*.cpp`,`*.hpp`).
- List of other related files (`*.mat`, `*.rsp`, `*.prj`, etc.).
- List of exported symbols that you must not modify.

## Construction

To access the properties of this class, use the '`CustomPostProcessingHook`' option of the `Simulink.ModelReference.protect` function. The value for the option is a handle to a postprocessing function accepting a `Simulink.ModelReference.ProtectedModel.HookInfo` object as input.

## Properties

**`ExportedSymbols` — Exported Symbols**
cell array of character vectors

A list of exported symbols generated by protected model that you must not modify. Default value is empty.

For a protected model with a top model interface, the `HookInfo` object cannot provide information on exported symbols.

**`NonSourceFiles` — Non source code files**
cell array of character vectors

A list of non-source files generated by protected model creation. Examples are `*.mat`, `*.rsp`, and `*.prj`. Default value is empty.

**`SourceFiles` — Source code files**
cell array of character vectors

A list of source code files generated by protected model creation. Examples are `*.c`, `*.h`, `*.cpp`, and `*.hpp`. Default value is empty.

# Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

# See Also

`Simulink.ModelReference.protect`

## Topics

"Specify Custom Obfuscator for Protected Model" (Simulink Coder)

# 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.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.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.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')
```

- “Create a Protected Model with Multiple Targets” (Simulink Coder)

## 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 a Protected Model with Multiple Targets" (Simulink Coder)

**Introduced in R2015a**

# Simulink.ProtectedModel.Callback class

**Package:** Simulink.ProtectedModel

Represents callback code that executes in response to protected model events

## Description

For a 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. The object includes:

- The code to execute for the callback.
- The event that triggers the callback.
- The protected model functionality that the event applies to.
- The option to override the protected model build.

When you create a protected model, to specify callbacks, call the Simulink.ModelReference.protect function with the `'Callbacks'` option. The value of this option is a cell array of `Simulink.ProtectedModel.Callback` objects.

## Construction

`pmCallback = Simulink.ProtectedModel.Callback(event,functionality, callbackText)` creates a callback object for a specific protected model functionality and event. The `callbackText` specifies MATLAB commands to execute for the callback.

`pmCallback = Simulink.ProtectedModel.Callback(event,functionality, callbackFile)` creates a callback object for a specific protected model functionality and event. The `callbackFile` specifies a MATLAB script to execute for the callback. The script must be on the MATLAB path.

### Input Arguments

**`event` — Event that triggers callback**
`'PreAccess'` | `'Build'`

Callback trigger event. Specify one of the following values:

- `'PreAccess'`: Callback code is executed before simulation, build, or read-only viewing.
- `'Build'`: Callback code is executed before build. Valid only for `'CODEGEN'` functionality.

#### `functionality` — Protected model functionality

`'CODEGEN'` | `'SIM'` | `'VIEW'` | `'AUTO'`

Protected model functionality that the event applies to. Specify one of the following 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 only for `'CODEGEN'` functionality.

If you do not specify a functionality, the default behavior is `'AUTO'`.

#### `callbackText` — Callback code to execute

string or character vector

MATLAB commands to execute in response to an event, specified as a string or character vector.

#### `callbackFile` — Callback script to execute

string or character vector

MATLAB script to execute in response to an event, specified as a string or character vector. Script must be on the MATLAB path.

## Properties

#### `AppliesTo` — 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 you do not specify a functionality, the default behavior is `'AUTO'`.

**`CallbackFileName` — Callback script to execute**
string or character vector

MATLAB script to execute in response to an event, specified as a string or character vector. Script must be on the MATLAB path.

Example: `'pmCallback.m'`

**`CallbackText` — Callback code to execute**
string or character vector

MATLAB commands to execute in response to an event, specified as a string or character vector.

Example: `'A = [15 150];disp(A)'`

**`Event` — Event that triggers callback**
`'PreAccess'` | `'Build'`

Callback trigger event. Value is one of the following:

- `'PreAccess'`: Callback code is executed before simulation, build, or read-only viewing.
- `'Build'`: Callback code is executed before build. Valid only for `'CODEGEN'` functionality.

**`OverrideBuild` — Option to override protected model build**
`false` (default) | `true`

Option to override the protected model build process, specified as a Boolean value. Applies only to a callback object that you define for a `'Build'` event for `'CODEGEN'` functionality. You set this option using the `setOverrideBuild` method.

## Methods

| | |
|---|---|
| setOverrideBuild | Specify option to override protected model build |

## Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

## Examples

### Create Protected Model Using a Callback

Create a callback object with a character vector of MATLAB commands for the callback code. Specify the object when you create a protected model.

```
pmCallback = Simulink.ProtectedModel.Callback('PreAccess',...
'SIM','disp(''Hello world!'')')
Simulink.ModelReference.protect('sldemo_mdlref_counter',...
'Callbacks',{pmCallback})
sim('sldemo_mdlref_basic')
```

For each instance of the protected model reference in the top model, the output is listed.

```
Hello world!
Hello world!
Hello world!
```

### Create Protected Model With a Callback Script

Create a callback object with a MATLAB script for the callback code. Specify the object when you create a protected model.

```
pmCallback = Simulink.ProtectedModel.Callback('Build',...
'CODEGEN','pm_callback.m')
Simulink.ModelReference.protect('sldemo_mdlref_counter',...
'Mode', 'CodeGeneration','Callbacks',{pmCallback})
rtwbuild('sldemo_mdlref_basic')
```

Before the protected model build process begins, code in `pm_callback.m` executes.

## See Also

`Simulink.ModelReference.protect` |
`Simulink.ProtectedModel.getCallbackInfo`

### Topics

"Define Callbacks for Protected Models" (Simulink Coder)
"Protect Models for Third-Party Use" (Simulink Coder)
"Code Generation Support in Protected Models" (Simulink Coder)

**Introduced in R2016a**

# setOverrideBuild

**Class:** Simulink.ProtectedModel.Callback
**Package:** Simulink.ProtectedModel

Specify option to override protected model build

## Syntax

`setOverrideBuild(override)`

## Description

`setOverrideBuild(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.

## Input Arguments

**`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 only to a callback object defined for a `'Build'` event for `'CODEGEN'` functionality.

Example: `pmcallback.setOverrideBuild(true)`

# Examples

## Create Code Generation Callback to Override Build Process

Create a callback object with a character vector of MATLAB commands for the callback code. Specify that the callback override the build process.

```
pmCallback = Simulink.ProtectedModel.Callback('Build',...
'CODEGEN','disp(''Hello world!'')')
pmCallback.setOverrideBuild(true);
Simulink.ModelReference.protect('sldemo_mdlref_counter',...
'Mode', 'CodeGeneration','Callbacks',{pmCallback})
rtwbuild('sldemo_mdlref_basic')
```

# See Also

`Simulink.ModelReference.protect` | `Simulink.ProtectedModel.Callback`

### Topics

"Define Callbacks for Protected Models" (Simulink Coder)
"Protect Models for Third-Party Use" (Simulink Coder)
"Code Generation Support in Protected Models" (Simulink Coder)

**Introduced in R2016a**

# Simulink.ProtectedModel.CallbackInfo class

**Package:** Simulink.ProtectedModel

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. The object provides:

- 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.

## Construction

`cbinfobj = Simulink.ProtectedModel.getCallbackInfo(modelName,event,functionality)` creates a `Simulink.ProtectedModel.CallbackInfo` object.

## Properties

**`CodeInterface` — Code interface generated by protected model**
`'Top model'` | `'Model reference'`

Code interface that the protected model generates.

**`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'`.

**`ModelName` — Protected model name**
character vector

Protected model name, specified as a character vector.

**`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 only for code generation callbacks.

# Methods

| | |
|---|---|
| getBuildInfoForModel | Get build information object for specified model |

## Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

## Examples

### Use Protected Model Information in Simulation Callback

Create a protected model callback that uses information from the `Simulink.ProtectedModel.Callback` object.

First, on the MATLAB path, create a callback script , `pm_callback.m`, containing:

```
s1 = 'Simulating protected model: ';
cbinfobj = Simulink.ProtectedModel.getCallbackInfo(...
'sldemo_mdlref_counter','PreAccess','SIM');
disp([s1 cbinfobj.ModelName])
```

When you create a protected model with a simulation callback, use the script.

```
pmCallback = Simulink.ProtectedModel.Callback('PreAccess'...
,'SIM', 'pm_callback.m')
Simulink.ModelReference.protect('sldemo_mdlref_counter',...
'Callbacks',{pmCallback})
```

Simulate the protected model. For each instance of the protected model reference in the top model, 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” (Simulink Coder)
“Protect Models for Third-Party Use” (Simulink Coder)
“Code Generation Support in Protected Models” (Simulink Coder)

**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.

```
rtwbuild('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" (Simulink Coder)
"Protect Models for Third-Party Use" (Simulink Coder)
"Code Generation Support in Protected Models" (Simulink Coder)

**Introduced in R2016a**

# getBuildInfoForModel

**Class:** Simulink.ProtectedModel.CallbackInfo
**Package:** Simulink.ProtectedModel

Get build information object for specified model

## Syntax

```
bldobj = getBuildInfoForModel(model)
```

## Description

`bldobj = getBuildInfoForModel(model)` returns a handle to an `RTW.BuildInfo` object. This object specifies the build toolchain and arguments. 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 only for code generation callbacks in response to a `'Build'` event.

## Input Arguments

**`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. You can call this method only for code generation callbacks in response to a `'Build'` event.

## Output Arguments

**`bldobj` — Object for build toolchain and arguments**
`RTW.BuildInfo`

Build toolchain and arguments, specified as a `RTW.BuildInfo` object. If you do not call the method for a code generation callback and `'Build'` event, the return value is an empty array.

# Examples

## Get Build Information from a Code Generation Callback

On the MATLAB path, create a callback script, `pm_callback.m`, containing:

```
cbinfobj = Simulink.ProtectedModel.getCallbackInfo(...
'sldemo_mdlref_counter','Build','CODEGEN');
bldinfo = cbinfobj.getBuildInfoForModel(cbinfobj.ModelName);
buildargs = getBuildArgs(bldinfo)
```

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, the build arguments are displayed.

```
rtwbuild('sldemo_mdlref_basic')
```

# See Also

`Simulink.ModelReference.protect` | `Simulink.ProtectedModel.CallbackInfo`

### Topics

“Define Callbacks for Protected Models” (Simulink Coder)
“Protect Models for Third-Party Use” (Simulink Coder)
“Code Generation Support in Protected Models” (Simulink Coder)

**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')
```

- "Create a Protected Model with Multiple Targets" (Simulink Coder)
- "Use a Protected Model with Multiple Targets" (Simulink Coder)

## 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 a Protected Model with Multiple Targets" (Simulink Coder)

“Use a Protected Model with Multiple Targets” (Simulink Coder)

**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')
```

- “Create a Protected Model with Multiple Targets” (Simulink Coder)
- “Use a Protected Model with Multiple Targets” (Simulink Coder)

## 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 a Protected Model with Multiple Targets" (Simulink Coder)
"Use a Protected Model with Multiple Targets" (Simulink Coder)

**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')
```

- “Create a Protected Model with Multiple Targets” (Simulink Coder)
- “Use a Protected Model with Multiple Targets” (Simulink Coder)

## 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 a Protected Model with Multiple Targets" (Simulink Coder)
"Use a Protected Model with Multiple Targets" (Simulink Coder)

**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 with no 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')
```

- "Create a Protected Model with Multiple Targets" (Simulink Coder)

## 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 a Protected Model with Multiple Targets" (Simulink Coder)

**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 correct.

```
ct = Simulink.ProtectedModel.getCurrentTarget('mdlref_counter')
```

- “Create a Protected Model with Multiple Targets” (Simulink Coder)
- “Use a Protected Model with Multiple Targets” (Simulink Coder)

## 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 a Protected Model with Multiple Targets" (Simulink Coder)
"Use a Protected Model with Multiple Targets" (Simulink Coder)

**Introduced in R2015a**

# slConfigUIGetVal

Return current value for custom target configuration option

## Syntax

`value = slConfigUIGetVal(hDlg,hSrc,'OptionName')`

## Input Arguments

`hDlg`

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.

`hSrc`

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.

`'OptionName'`

Quoted name of the TLC variable defined for a custom target configuration option.

## Output Arguments

Current value of the specified option. The data type of the return value depends on the data type of the option.

## Description

The `slConfigUIGetVal` function 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 `slConfigUIGetVal` to read the current value of a specified target option.

## Examples

In the following example, 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);
```

## See Also

`slConfigUISetEnabled` | `slConfigUISetVal`

### Topics

“Define and Display Custom Target Options” (Simulink Coder)
“Custom Target Optional Features” (Simulink Coder)

**Introduced in R2006b**

# slConfigUISetEnabled

Enable or disable custom target configuration option

## Syntax

```
slConfigUISetEnabled(hDlg,hSrc,'OptionName',true)
slConfigUISetEnabled(hDlg,hSrc,'OptionName',false)
```

## Arguments

`hDlg`

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.

`hSrc`

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.

'*OptionName*'

Quoted name of the TLC variable defined for a custom target configuration option.

`true`

Specifies that the option should be enabled.

`false`

Specifies that the option should be disabled.

## Description

The `slConfigUISetEnabled` function 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 target option.

If you use this function to disable a parameter that is represented in the Configuration Parameters dialog box, the parameter appears greyed out in the dialog context.

## Examples

In the following example, 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);
```

## See Also

`slConfigUIGetVal` | `slConfigUISetVal`

### Topics

“Define and Display Custom Target Options” (Simulink Coder)
“Custom Target Optional Features” (Simulink Coder)

**Introduced in R2006b**

# slConfigUISetVal

Set value for custom target configuration option

## Syntax

```
slConfigUISetVal(hDlg,hSrc,'OptionName',OptionValue)
```

## Arguments

`hDlg`

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.

`hSrc`

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.

*'OptionName'*

Quoted name of the TLC variable defined for a custom target configuration option.

*OptionValue*

Value to be set for the specified option.

## Description

The `slConfigUISetVal` function 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

In the following example, 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);
```

## See Also

`slConfigUIGetVal` | `slConfigUISetEnabled`

### Topics

“Define and Display Custom Target Options” (Simulink Coder)
“Custom Target Optional Features” (Simulink Coder)

**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 `Simulink.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`. For no options, specify 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

`Simulink.ConfigSet` | `Simulink.ConfigSetRef` | `getActiveConfigSet`

## Topics

"Select a System Target File Programmatically" (Simulink Coder)

“Configure a System Target File” (Simulink Coder)
“Set Target Language Compiler Options” (Simulink Coder)

**Introduced in R2009b**

# tlc

Invoke Target Language Compiler to convert model description file to generated code

## Syntax

`tlc [-options] [file]`

## Description

`tlc [-options] [file]` invokes the Target Language Compiler (TLC) from the command prompt. The TLC converts the model description file, `model.rtw` (or similar files), into target-specific code or text. Typically, you do not call this command because the build process automatically invokes the Target Language Compiler when generating code. For more information, see “Introduction to the Target Language Compiler” (Simulink Coder).

---

**Note** This command is used only when invoking the TLC separately from the build process. You cannot use this command to initiate code generation for a model.

---

You can change the default behavior by specifying one or more compilation *options* as described in “Options” on page 2-277

## Options

You can specify one or more compilation options with each `tlc` command. Use spaces to separate options and arguments. TLC resolves options from left to right. If you use conflicting options, the right-most option prevails. The `tlc` options are:


- “-r Specify model.rtw file name” on page 2-278
- “-v Specify verbose level” on page 2-278
- “-l Specify path to local include files” on page 2-278

- "-m Specify maximum number of errors" on page 2-278
- "-O Specify the output file path" on page 2-279
- "-d[a|c|n|o] Invoke debug mode" on page 2-279
- "-a Specify parameters" on page 2-279
- "-p Print progress" on page 2-279
- "-lint Performance checks and runtime statistics" on page 2-279
- "-xO Parse only" on page 2-280

## -r Specify *`model`*`.rtw` file name

`-r` *file_name*

Specify the file name that you want to translate.

## -v Specify verbose level

`-v` *number*

Specify a `number` indicating the verbose level. If you omit this option, the default value is one.

## -l Specify path to local include files

`-l` *path*

Specify a folder `path` to local include files. The TLC searches this path in the order specified.

## -m Specify maximum number of errors

`-m` *number*

Specify the maximum `number` of errors reported by the TLC prior to terminating the translation of the `.tlc` file.

If you omit this option, the default value is five.

## -O Specify the output file path

`-O` *path*

Specify the folder `path` to place output files.

If you omit this option, TLC places output files in the current folder.

## -d[a|c|n|o] Invoke debug mode

`-da` execute any %assert directives

`-dc` invoke the TLC command line debugger

`-dn` produce log files, which indicate those lines hit and those lines missed during compilation.

`-do` disable debugging behavior

## -a Specify parameters

`-a` *identifier* = *expression*

Specify parameters to change the behavior of your TLC program. For example, this option is used by the code generator to set inlining of parameters or file size limits.

## -p Print progress

`-p` *number*

Print a '.' indicating progress for every `number` of TLC primitive operations executed.

## -lint Performance checks and runtime statistics

`-lint`

Perform simple performance checks and collect runtime statistics.

# -xO Parse only

```
-x0
```

Parse only a TLC file; do not execute it.

**Introduced in R2009a**

# updateFilePathsAndExtensions

Update files in model 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 model.

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 `myModelBuildInfo` with missing paths or file extensions.

```
myModelBuildInfo = RTW.BuildInfo;
addSourcePaths(myModelBuildInfo,fullfile(pwd, ...
```

```
   'etcproj','/etc'),'test');
addSourceFiles(myModelBuildInfo,{'etc' 'test1' ...
   'test2'},'','test');
before = getSourceFiles(myModelBuildInfo,true,true);
```

```
>> before

before = 

    '\etc'    '\test1'    '\test2'
```

```
updateFilePathsAndExtensions(myModelBuildInfo);
after = getSourceFiles(myModelBuildInfo,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

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" (Simulink Coder)

**Introduced in R2006a**

# updateFileSeparator

Update file separator character for file lists in model build information

## Syntax

`updateFileSeparator(buildinfo,separator)`

## Description

`updateFileSeparator(buildinfo,separator)` changes instances of the current file separator (/ or \) in the model 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 (see “Cross-Compile Code Generated on Microsoft Windows” (Simulink Coder). If the `GenerateMakefile` parameter is set, the code generator overrides the default separator and updates the model build information after evaluating the `PostCodeGenCommand` configuration parameter.

## Examples

### Update File Separator in Build Information

Update object `myModelBuildInfo` to apply the Windows file separator.

```
myModelBuildInfo = RTW.BuildInfo;
updateFileSeparator(myModelBuildInfo,'\');
```

## Input Arguments

**`buildinfo` — Name of build information object returned by `RTW.BuildInfo`**
object

**`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" (Simulink Coder)
"Cross-Compile Code Generated on Microsoft Windows" (Simulink Coder)

**Introduced in R2006a**

# Blocks in Embedded Coder— Alphabetical List

# Byte Pack

Convert input signals to `uint8` vector

## Library

Embedded Coder/Embedded Targets/Host Communication

## Description

Using the input port, the block converts data of one or more data types into a single `uint8` vector for output. With the options available, you specify the input data types and the alignment of the data in the output vector. Because UDP messages are in `uint8` data format, use this block before a UDP Send block to format the data for transmission using the UDP protocol.

## Parameters

**Input port data types (cell array)**

Specify the data types for the different signals as part of the parameters. The block supports allSimulink data types except characters. Enter the data types as Simulink types in the cell array, such as `'double'` or `'int32'`. The order of the data type entries in the cell array must match the order in which the data arrives at the block input. This block determines the signal sizes automatically. The block has at least one input port and only one output port.

**Byte alignment**

This option specifies how to align the data types to form the `uint8` output vector. Select one of the values in bytes from the list.

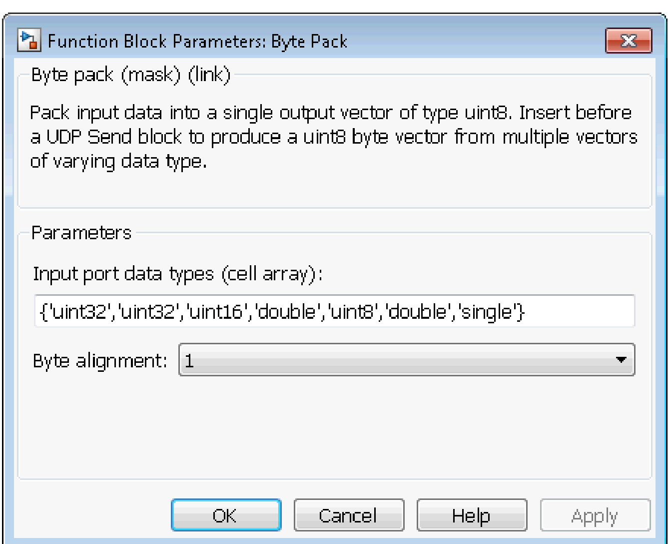
Alignment can occur on 1, 2, 4, or 8-byte boundaries depending on the value you choose. The value defaults to 1. Given the alignment value, each signal data value begins on multiples of the alignment value. The alignment algorithm s that each element in the output vector begins on a byte boundary specified by the alignment value. Byte alignment sets the boundaries relative to the starting point of the vector.

Selecting 1 for **Byte alignment** provides the tightest packing, without holes between data types in the various combinations of data types and signals.

Sometimes, you can have multiple data types of varying lengths. In such cases, specifying a 2-byte alignment can produce 1–byte gaps between uint8 or int8 values and another data type. In the pack implementation, the block copies data to the output data buffer 1 byte at a time. You can specify data alignment options with data types.

## Example

Use a cell array to enter input data types in the **Input port data types** parameter. The order of the data types you enter must match the order of the data types at the block input.



In the cell array, you provide the order in which the block expects to receive data—`uint32`, `uint32`, `uint16`, `double`, `uint8`, `double`, and `single`. With this information, the block automatically provides the number of block inputs.

Byte alignment equal to `2` specifies that each new value begins 2 bytes from the previous data boundary.

The example shows the following data types:

```
{'uint32','uint32','uint16','double','uint8','double','single'}
```

When the signals are scalar values (not matrices or vectors in this example), the first signal value in the vector starts at 0 bytes. Then, the second signal value starts at 2 bytes, and the third at 4 bytes. Next, the fourth signal value follows at 6 bytes, the fifth at 8 bytes, the sixth at 10 bytes, and the seventh at 12 bytes. As the example shows, the packing algorithm leaves a 1-byte gap between the `uint8` data value and the double value.

## See Also

Byte Reversal, Byte Unpack

**Introduced in R2011a**

# Byte Reversal

Reverse order of bytes in input word

## Library

Embedded Coder/Embedded Targets/Host Communication

## Description

Byte reversal changes the order of the bytes in data you input to the block. Use this block when your process communicates between targets that use different endianness, such as between Intel processors that are little endian and others that are big endian. Texas Instruments processors are little-endian by default.

To exchange data with a processor that has different endianness, place a Byte Reversal block just before the send block and immediately after the receive block.

## Parameters

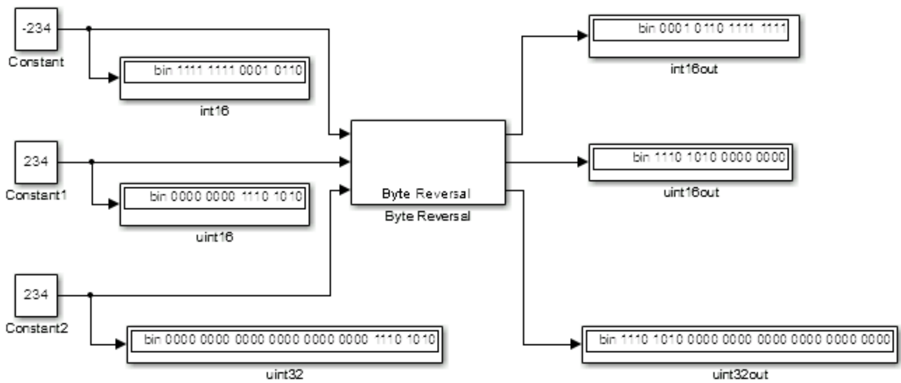
**Number of inputs**

Specify the number of block inputs. The number of block inputs adjusts automatically to match value so the number of outputs equals the number of inputs.

When you use more than one input port, each input port maps to the matching output port. Data entering input port 1 leaves through output port 1, and so on.

Reversing the bytes does not change the data type. Input and output retain matching data type.

The following model shows byte reversal in use. In this figure, the input and output ports match for each path.




## See Also

Byte Pack, Byte Unpack

**Introduced in R2011a**

# Byte Unpack

Unpack UDP `uint8` input vector into Simulink data type values

## Library

Embedded Coder/Embedded Targets//Host Communication

## Description

Byte Unpack is the inverse of the Byte Pack block. It takes a UDP message from a UDP receive block as a `uint8` vector, and outputs Simulink data types in various sizes depending on the input vector.

The block supports all Simulink data types.

## Parameters

**Output port dimensions (cell array)**

Containing a cell array, each element in the array specifies the dimension that the MATLAB `size` function returns for the corresponding signal. Usually you use the same dimensions as you set for the corresponding Byte Pack block in the model. Entering one value means that the block applies that dimension to all data types.

**Output port data types (cell array)**

Specify the data types for the different input signals to the Pack block. The block supports all Simulink data types—`single`, `double`, `int8`, `uint8`, `int16`, `uint16`, `int32`, and `uint32`, and `Boolean`. The entry here is the same as the Input port data types parameter in the Byte Pack block in the model. You can enter one data type and the block applies that type to all output ports.

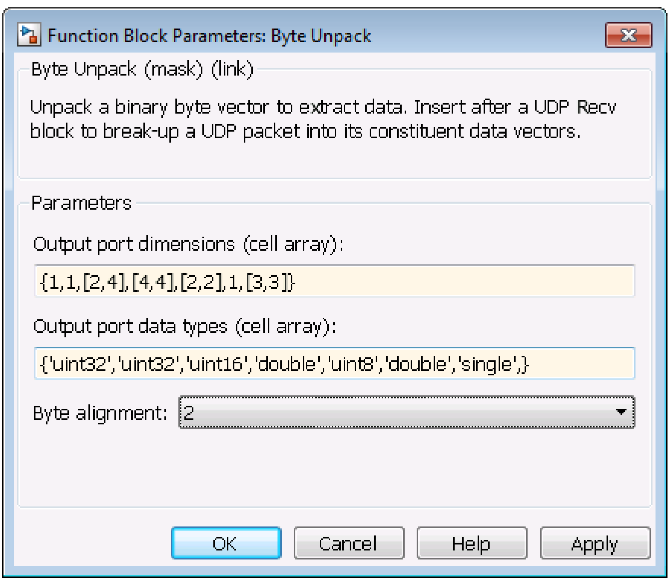
**Byte Alignment**

This option specifies how to align the data types to form the input `uint8` vector. Match this setting with the corresponding Byte Pack block alignment value of 1, 2, 4, or 8 bytes.

# Example

This figure shows the Byte Unpack block that corresponds to the example in the Byte Pack example. The **Output port data types (cell array)** entry shown is the same as the **Input port data types (cell array)** entry in the Byte Pack block

`{'uint32','uint32','uint16','double','uint8','double','single'}.`




In addition, the **Byte alignment** setting matches as well. **Output port dimensions (cell array)** now includes scalar values and matrices to show how to enter nonscalar values. The example for the Byte Pack block assumed only scalar inputs.

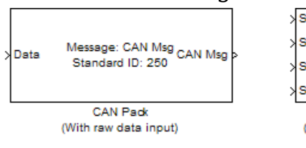
# See Also

Byte Pack, Byte Reversal

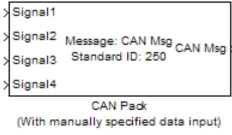
**Introduced in R2011a**

# CAN Pack

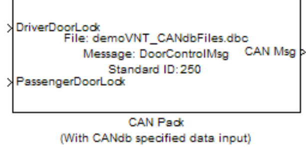
Pack individual signals into CAN message



CAN Pack
(With raw data input)



CAN Pack
(With manually specified data input)



CAN Pack
(With CANdb specified data input)

## Library

CAN Communication

Embedded Coder/ Embedded Targets/ Host Communication

## Description

The CAN Pack block loads signal data into a message at specified intervals during the simulation.

**Note** To use this block, you also need a license for Simulink software.

CAN Pack block has one input port by default. The number of block inputs is dynamic and depends on the number of signals you specify for the block. For example, if your block has four signals, it has four block inputs.

This block has one output port, CAN Msg. The CAN Pack block takes the specified input parameters and packs the signals into a message.

### Other Supported Features

The CAN Pack block supports:

- The use of Simulink Accelerator™ Rapid Accelerator mode. Using this feature, you can speed up the execution of Simulink models.

- The use of model referencing. Using this feature, your model can include other Simulink models as modular components.
- Code generation to deploy models to targets.

**Note** Code generation is not supported if your signal information consists of signed or unsigned integers greater than 32 bits long.

For more information on these features, see the Simulink documentation.

# Dialog Box

Use the Function Block Parameters dialog box to select your CAN Pack block parameters.

## Parameters

**Data is input as**

Select your data signal:

- **raw data**: Input data as a uint8 vector array. If you select this option, you only specify the message fields. all other signal parameter fields are unavailable. This option opens only one input port on your block.
- **manually specified signals**: Allows you to specify data signal definitions. If you select this option, use the **Signals** table to create your signals. The number of block inputs depends on the number of signals you specify.
- **CANdb specified signals**: Allows you to specify a CAN database file that contains message and signal definitions. If you select this option, select a CANdb file. The number of block inputs depends on the number of signals specified in the CANdb file for the selected message.

**Note** The block supports the following input signals data types: single, double, int8, int16, int32, int64, uint8, uint16, uint32, uint64, and boolean. The block does not support fixed-point data types.

**CANdb file**

This option is available if you specify that your data is input via a CANdb file in the **Data is input as** list. Click **Browse** to find the CANdb file on your system. The

message list specified in the CANdb file populates the **Message** section of the dialog box. The CANdb file also populates the **Signals** table for the selected message.

---

**Note** File names that contain non-alphanumeric characters such as equal signs, ampersands, and so forth are not valid CAN database file names. You can use periods in your database name. Rename CAN database files with non-alphanumeric characters before you use them.

---

**Message list**

This option is available if you specify that your data is input via a CANdb file in the **Data is input as** field and you select a CANdb file in the **CANdb file** field. Select the message to display signal details in the **Signals** table.

## Message

**Name**

Specify a name for your CAN message. The default is `CAN Msg`. This option is available if you choose to input raw data or manually specify signals. This option in unavailable if you choose to use signals from a CANdb file.

**Identifier type**

Specify whether your CAN message identifier is a `Standard` or an `Extended` type. The default is `Standard`. A standard identifier is an 11-bit identifier and an extended identifier is a 29-bit identifier. This option is available if you choose to input raw data or manually specify signals. For `CANdb specified signals`, the **Identifier type** inherits the type from the database.

**Identifier**

Specify your CAN message ID. This number must be a positive integer from 0 through 2047 for a standard identifier and from 0 through 536870911 for an extended identifier. You can also specify hexadecimal values using the `hex2dec` function. This option is available if you choose to input raw data or manually specify signals.

**Length (bytes)**

Specify the length of your CAN message from 0 to 8 bytes. If you are using `CANdb specified signals` for your data input, the CANdb file defines the length of your message. If not, this field defaults to `8`. This option is available if you choose to input raw data or manually specify signals.

**Remote frame**

Specify the CAN message as a remote frame.

Output as bus

Select this option for the block to output CAN messages as a Simulink bus signal. For more information on Simulink bus objects, see “Composite Signals” (Simulink).

## Signals Table

This table appears if you choose to specify signals manually or define signals using a CANdb file.

If you are using a CANdb file, the data in the file populates this table automatically and you cannot edit the fields. To edit signal information, switch to manually specified signals.

If you have selected to specify signals manually, create your signals manually in this table. Each signal you create has the following values:

**Name**

Specify a descriptive name for your signal. The Simulink block in your model displays this name. The default is `Signal [row number]`.

**Start bit**

Specify the start bit of the data. The start bit is the least significant bit counted from the start of the message data. The start bit must be an integer from 0 through 63.

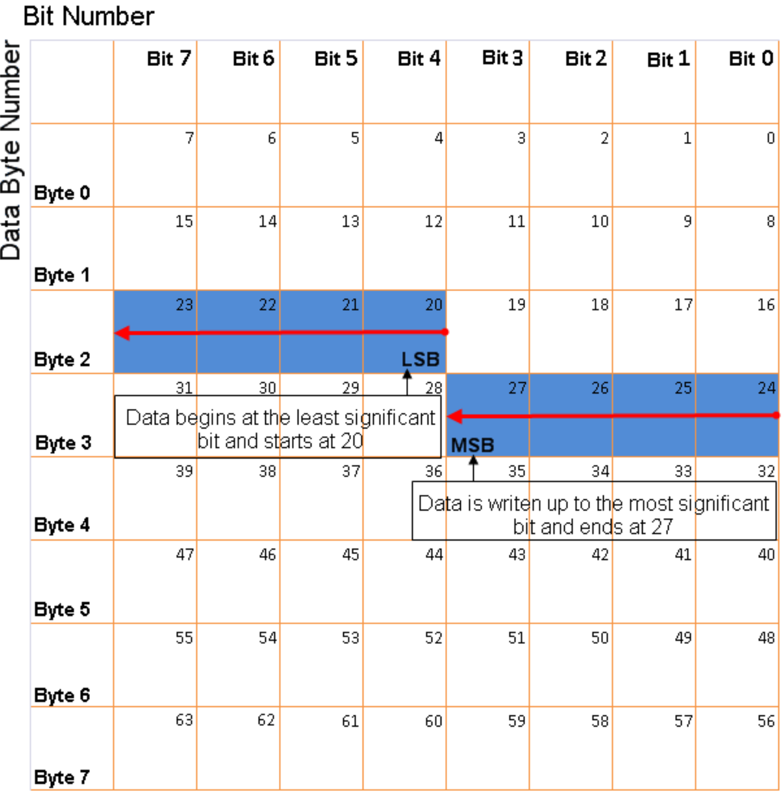
**Length (bits)**

Specify the number of bits the signal occupies in the message. The length must be an integer from 1 through 64.

**Byte order**

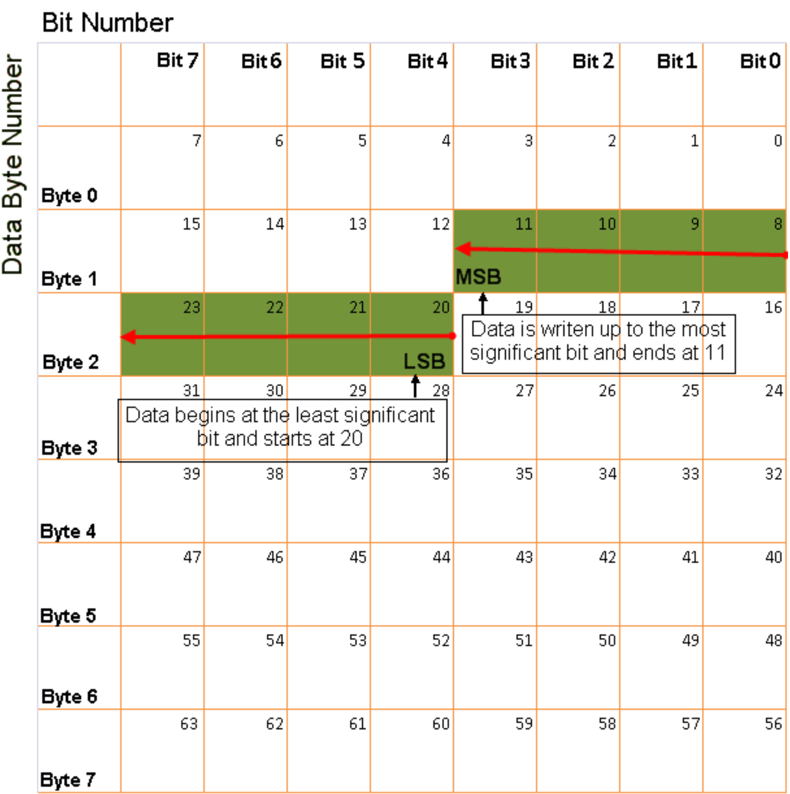
Select either of the following options:

- `LE`: Where the byte order is in little-endian format (Intel). In this format you count bits from the start, which is the least significant bit, to the most significant bit, which has the highest bit index. For example, if you pack one byte of data in little-endian format, with the start bit at 20, the data bit table resembles this figure.




**Little-Endian Byte Order Counted from the Least Significant Bit to the Highest Address**

- BE: Where byte order is in big-endian format (Motorola®). In this format you count bits from the start, which is the least significant bit, to the most significant bit. For example, if you pack one byte of data in big-endian format, with the start bit at 20, the data bit table resembles this figure.




**Big-Endian Byte Order Counted from the Least Significant Bit to the Lowest Address**

**Data type**

Specify how the signal interprets the data in the allocated bits. Choose from:

- `signed` (default)
- `unsigned`
- `single`
- `double`

**Multiplex type**

Specify how the block packs the signals into the CAN message at each timestep:

- `Standard`: The signal is packed at each timestep.
- `Multiplexor`: The `Multiplexor` signal, or the mode signal is packed. You can specify only one `Multiplexor` signal per message.
- `Multiplexed`: The signal is packed if the value of the `Multiplexor` signal (mode signal) at run time matches the configured **Multiplex value** of this signal.

For example, a message has four signals with the following types and values.

| Signal Name | Multiplex Type | Multiplex Value |
| --- | --- | --- |
| Signal-A | Standard | N/A |
| Signal-B | Multiplexed | 1 |
| Signal-C | Multiplexed | 0 |
| Signal-D | Multiplexor | N/A |

In this example:

- The block packs Signal-A (Standard signal) and Signal-D (Multiplexor signal) in every timestep.
- If the value of Signal-D is 1 at a particular timestep, then the block packs Signal-B along with Signal-A and Signal-D in that timestep.
- If the value of Signal-D is 0 at a particular timestep, then the block packs Signal-C along with Signal-A and Signal-D in that timestep.
- If the value of Signal-D is not 1 or 0, the block does not pack either of the Multiplexed signals in that timestep.

**Multiplex value**

This option is available only if you have selected the **Multiplex type** to be `Multiplexed`. The value you provide here must match the `Multiplexor` signal value at run time for the block to pack the `Multiplexed` signal. The **Multiplex value** must be a positive integer or zero.

**Factor**

Specify the **Factor** value to apply to convert the physical value (signal value) to the raw value packed in the message. See “Conversion Formula” on page 3-17 to understand how physical values are converted to raw values packed into a message.

**Offset**

Specify the **Offset** value to apply to convert the physical value (signal value) to the raw value packed in the message. See "Conversion Formula" on page 3-17 to understand how physical values are converted to raw values packed into a message.

**Min**

Specify the minimum physical value of the signal. The default value is `-inf` (negative infinity). You can specify a number for the minimum value. See "Conversion Formula" on page 3-17 to understand how physical values are converted to raw values packed into a message.

**Max**

Specify the maximum physical value of the signal. The default value is `inf`. You can specify a number for the maximum value. See "Conversion Formula" on page 3-17 to understand how physical values are converted to raw values packed into a message.

## Conversion Formula

The conversion formula is

```
raw_value = (physical_value - Offset) / Factor
```

where `physical_value` is the value of the signal after it is saturated using the specified **Min** and **Max** values. `raw_value` is the packed signal value.

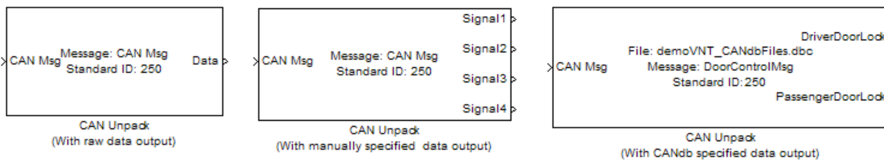
# See Also

**Blocks**
CAN Unpack

**Introduced in R2009a**

# CAN Unpack

Unpack individual signals from CAN messages



CAN Unpack (With raw data output) CAN Unpack (With manually specified data output) CAN Unpack (With CANdb specified data output)

## Library

CAN Communication

Embedded Coder/ Embedded Targets/ Host Communication

## Description

The CAN Unpack block unpacks a CAN message into signal data using the specified output parameters at every timestep. Data is output as individual signals.

---

**Note** To use this block, you also need a license for Simulink software.

---

The CAN Unpack block has one output port by default. The number of output ports is dynamic and depends on the number of signals you specify for the block to output. For example, if your block has four signals, it has four output ports.

### Other Supported Features

The CAN Unpack block supports:

- The use of Simulink Accelerator Rapid Accelerator mode. Using this feature, you can speed up the execution of Simulink models.

- The use of model referencing. Using this feature, your model can include other Simulink models as modular components.
- Code generation to deploy models to targets.

**Note** Code generation is not supported if your signal information consists of signed or unsigned integers greater than 32 bits long.

For more information on these features, see the Simulink documentation.

# Dialog Box

Use the Function Block Parameters dialog box to select your CAN message unpacking parameters.

## Parameters

**Data to be output as**

Select your data signal:

- **raw data**: Output data as a uint8 vector array. If you select this option, you only specify the message fields. The other signal parameter fields are unavailable. This option opens only one output port on your block.
- **manually specified signals**: Allows you to specify data signals. If you select this option, use the `Signals` table to create your signals message manually.

  The number of output ports on your block depends on the number of signals you specify. For example, if you specify four signals, your block has four output ports.
- **CANdb specified signals**: Allows you to specify a CAN database file that contains data signals. If you select this option, select a CANdb file.

  The number of output ports on your block depends on the number of signals specified in the CANdb file. For example, if the selected message in the CANdb file has four signals, your block has four output ports.

**Note** For manually or CANdb specified signals, the default output signal data type is double. To specify other types, use a Signal Specification block. This allows the block to

support the following output signal data types: single, double, int8, int16, int32, int64, uint8, uint16, uint32, uint64, and boolean. The block does not support fixed-point types.

**CANdb file**

This option is available if you specify that your data is input via a CANdb file in the **Data to be output as** list. Click **Browse** to find the CANdb file on your system. The messages and signal definitions specified in the CANdb file populate the **Message** section of the dialog box. The signals specified in the CANdb file populate **Signals** table.

**Note** File names that contain non-alphanumeric characters such as equal signs, ampersands, and so forth are not valid CAN database file names. You can use periods in your database name. Rename CAN database files with non-alphanumeric characters before you use them.

**Message list**

This option is available if you specify that your data is to be output as a CANdb file in the **Data to be output as** list and you select a CANdb file in the **CANdb file** field. You can select the message that you want to view. The **Signals** table then displays the details of the selected message.

## Message

**Name**

Specify a name for your CAN message. The default is `CAN Msg`. This option is available if you choose to output raw data or manually specify signals.

**Identifier type**

Specify whether your CAN message identifier is a `Standard` or an `Extended` type. The default is `Standard`. A standard identifier is an 11-bit identifier and an extended identifier is a 29-bit identifier. This option is available if you choose to output raw data or manually specify signals. For CANdb-specified signals, the **Identifier type** inherits the type from the database.

**Identifier**

Specify your CAN message ID. This number must be a integer from 0 through 2047 for a standard identifier and from 0 through 536870911 for an extended identifier. If

you specify `–1`, the block unpacks the messages that match the length specified for the message. You can also specify hexadecimal values using the `hex2dec` function. This option is available if you choose to output raw data or manually specify signals.

**Length (bytes)**

Specify the length of your CAN message from 0 to 8 bytes. If you are using `CANdb specified signals` for your output data, the CANdb file defines the length of your message. If not, this field defaults to `8`. This option is available if you choose to output raw data or manually specify signals.

## Signals Table

This table appears if you choose to specify signals manually or define signals using a CANdb file.

If you are using a CANdb file, the data in the file populates this table automatically and you cannot edit the fields. To edit signal information, switch to manually specified signals.

If you have selected to specify signals manually, create your signals manually in this table. Each signal you create has the following values:

**Name**

Specify a descriptive name for your signal. The Simulink block in your model displays this name. The default is `Signal [row number]`.

**Start bit**

Specify the start bit of the data. The start bit is the least significant bit counted from the start of the message. The start bit must be an integer from 0 through 63.

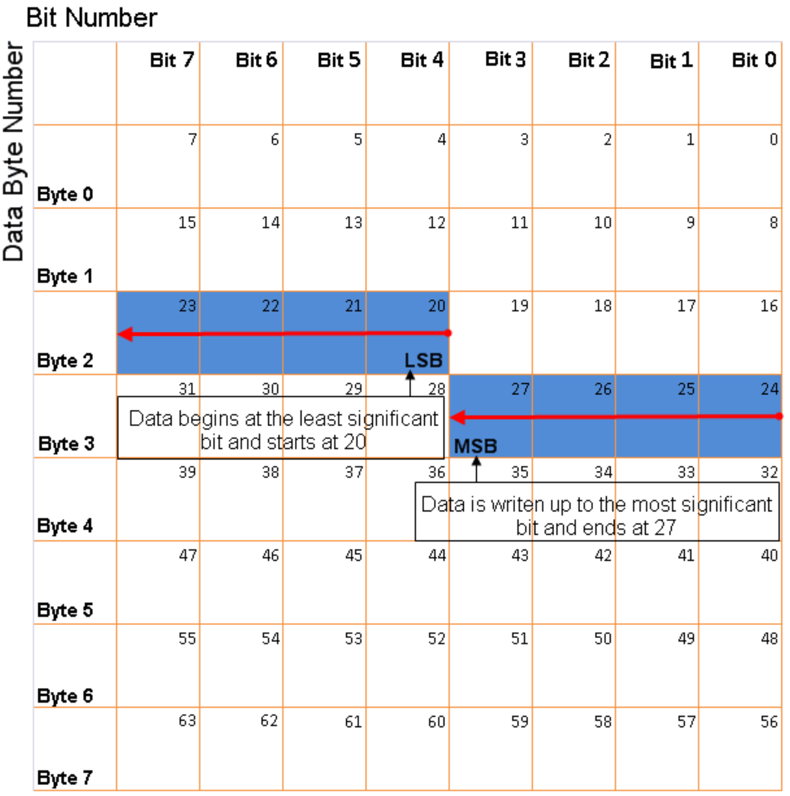
**Length (bits)**

Specify the number of bits the signal occupies in the message. The length must be an integer from 1 through 64.

**Byte order**

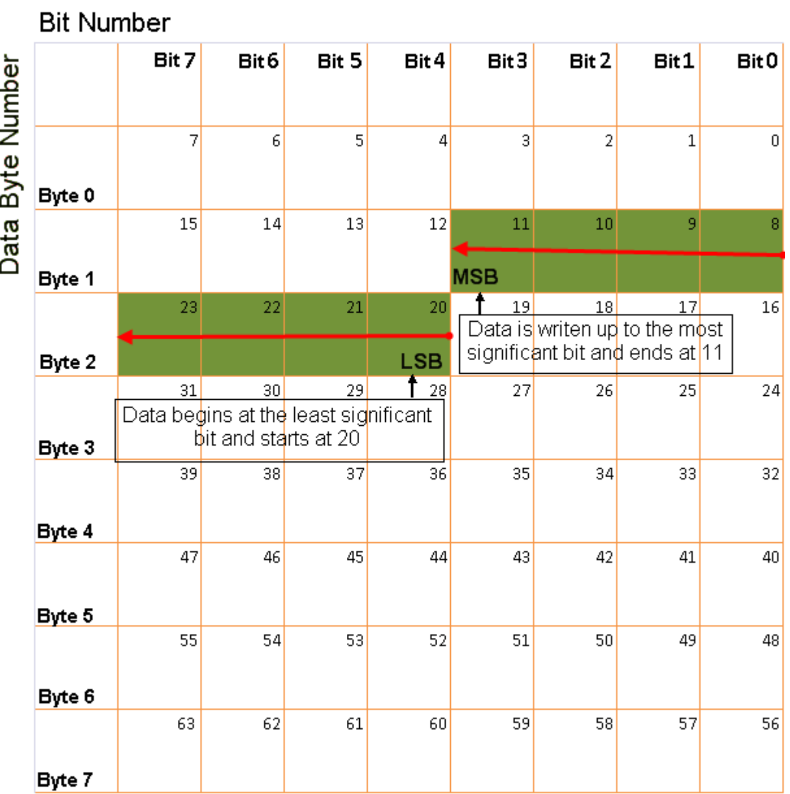
Select either of the following options:

- `LE`: Where the byte order is in little-endian format (Intel). In this format you count bits from the start, which is the least significant bit, to the most significant bit, which has the highest bit index. For example, if you pack one byte of data in little-endian format, with the start bit at 20, the data bit table resembles this figure.




**Little-Endian Byte Order Counted from the Least Significant Bit to the Highest Address**

- BE: Where the byte order is in big-endian format (Motorola). In this format you count bits from the start, which is the least significant bit, to the most significant bit. For example, if you pack one byte of data in big-endian format, with the start bit at 20, the data bit table resembles this figure.




**Big-Endian Byte Order Counted from the Least Significant Bit to the Lowest Address**

**Data type**

Specify how the signal interprets the data in the allocated bits. Choose from:

- `signed` (default)
- `unsigned`
- `single`
- `double`

**Multiplex type**

Specify how the block unpacks the signals from the CAN message at each timestep:

- `Standard`: The signal is unpacked at each timestep.
- `Multiplexor`: The `Multiplexor` signal, or the mode signal is unpacked. You can specify only one `Multiplexor` signal per message.
- `Multiplexed`: The signal is unpacked if the value of the `Multiplexor` signal (mode signal) at run time matches the configured **Multiplex value** of this signal.

For example, a message has four signals with the following values.

| Signal Name | Multiplex Type | Multiplex Value |
|---|---|---|
| Signal-A | Standard | N/A |
| Signal-B | Multiplexed | 1 |
| Signal-C | Multiplexed | 0 |
| Signal-D | Multiplexor | N/A |

In this example:

- The block unpacks Signal-A (Standard signal) and Signal-D (Multiplexor signal) in every timestep.
- If the value of Signal-D is 1 at a particular timestep, then the block unpacks Signal-B along with Signal-A and Signal-D in that timestep.
- If the value of Signal-D is 0 at a particular timestep, then the block unpacks Signal-C along with Signal-A and Signal-D in that timestep.
- If the value of Signal-D is not 1 or 0, the block does not unpack either of the Multiplexed signals in that timestep.

**Multiplex value**

This option is available only if you have selected the **Multiplex type** to be `Multiplexed`. The value you provide here must match the `Multiplexor` signal value at run time for the block to unpack the `Multiplexed` signal. The **Multiplex value** must be a positive integer or zero.

**Factor**

Specify the **Factor** value applied to convert the unpacked raw value to the physical value (signal value). See “Conversion Formula” on page 3-26 to understand how unpacked raw values are converted to physical values.

**Offset**

Specify the **Offset** value applied to convert the physical value (signal value) to the unpacked raw value. See “Conversion Formula” on page 3-26 to understand how unpacked raw values are converted to physical values.

**Min**

Specify the minimum raw value of the signal. The default value is `-inf` (negative infinity). You can specify a number for the minimum value. See “Conversion Formula” on page 3-26 to understand how unpacked raw values are converted to physical values.

**Max**

Specify the maximum raw value of the signal. The default value is `inf`. You can specify a number for the maximum value. See “Conversion Formula” on page 3-26 to understand how unpacked raw values are converted to physical values.

## Output Ports

Selecting an **Output ports** option adds an output port to your block.

**Output identifier**

Select this option to output a CAN message identifier. The data type of this port is **uint32**.

**Output remote**

Select this option to output the message remote frame status. This option adds a new output port to the block. The data type of this port is **uint8**.

**Output timestamp**

Select this option to output the message time stamp. This option adds a new output port to the block. The data type of this port is **double**.

**Output length**

Select this option to output the length of the message in bytes. This option adds a new output port to the block. The data type of this port is **uint8**.

**Output error**

Select this option to output the message error status. This option adds a new output port to the block. The data type of this port is **uint8**.

**Output status**

Select this option to output the message received status. The status is `1` if the block receives new message and `0` if it does not. This option adds a new output port to the block. The data type of this port is **uint8**.

If you do not select an **Output ports** option, the number of output ports on your block depends on the number of signals you specify.

## Conversion Formula

The conversion formula is

```
physical_value = raw_value * Factor + Offset
```

where `raw_value` is the unpacked signal value. `physical_value` is the scaled signal value which is saturated using the specified **Min** and **Max** values.

# See Also

**Blocks**
CAN Pack

**Introduced in R2009a**

# Custom MATLAB file

Update active configuration parameters of parent model by using file containing custom MATLAB code

**Library:** Embedded Coder / Configuration Wizards

Configure Model
(double-click to activate)
Custom MATLAB file

## Description

When you add a Custom MATLAB file block to your Simulink model and double-click it, a custom MATLAB script executes and configures model parameters that are relevant to code generation. Set a block parameter to invoke the build process after configuring the model.

After you double-click the block, open the Configuration Parameters dialog box to see that the model parameter values have changed.

Use the example MATLAB script, `*matlabroot*/toolbox/rtw/rtw/rtwsampleconfig.m` with the Custom MATLAB file block and adapt to your model requirements. The block and the script provide a starting point for customization. For more information, see "Create a Custom Configuration Wizard Block".

---

**Note** To provide a quick way to switch between configurations, you can include more than one Configuration Wizard block in your model.

---

## Parameters

**`Configure the model for` — Configuration objective**

`Custom` (default) | `ERT (optimized for fixed-point)` | `ERT (optimized for floating-point)` | `GRT (optimized for fixed/floating-point)` | `GRT (debug for fixed/floating-point)`

Objective for updating the active configuration parameters of a parent model by using a predefined MATLAB script.

**`Configuration function` — Custom MATLAB script**
`rtwsampleconfig` (default)

Predefined or custom MATLAB script that updates the active configuration parameters of the parent model.

**Dependencies**

This parameter is only used with this Configuration Wizards block. To enable this parameter, set **Configure the model for** to `Custom`.

**`Invoke build process after configuration` — Initiate build process**
`off` (default) | `on`

If selected, the predefined script initiates the code generation and build process after updating the configuration parameters.

## See Also

ERT (optimized for fixed-point) | ERT (optimized for floating-point) | GRT (debug for fixed/floating-point) | GRT (optimized for fixed/floating-point)

### Topics

"Configure and Optimize Model with Configuration Wizard Blocks"

**Introduced in R2011a**

# ERT (optimized for fixed-point)

Update active configuration parameters of parent model for ERT fixed-point code generation

**Library:** Embedded Coder / Configuration Wizards

## Description

When you add an ERT (optimized for fixed-point) block to your Simulink model and double-click it, a predefined MATLAB script executes and optimally configures the model parameters for fixed-point code generation with the ERT target. Set a block parameter to invoke the build process after configuring the model.

After you double-click the block, open the Configuration Parameters dialog box to see that the model parameter values have changed.

---

**Note** To provide a quick way to switch between configurations, you can include more than one Configuration Wizard block in your model.

---

## Parameters

### `Configure the model for` — Configuration objective

`ERT (optimized for fixed-point)` (default) | `ERT (optimized for floating-point)` | `GRT (optimized for fixed/floating-point)` | `GRT (debug for fixed/floating-point)` | `Custom`

Objective for updating the active configuration parameters of a parent model by using a predefined MATLAB script.

### `Configuration function` — Custom MATLAB script

`rtwsampleconfig` (default)

Predefined or custom MATLAB script that updates the active configuration parameters of the parent model.

#### Dependencies

This parameter is only used with the Custom MATLAB file block. To enable this parameter, set **Configure the model for** to `Custom`.

### `Invoke build process after configuration` — Initiate build process

`off` (default) | `on`

If selected, the predefined script initiates the code generation and build process after updating the configuration parameters.

# See Also

Custom MATLAB file | ERT (optimized for floating-point) | GRT (debug for fixed/floating-point) | GRT (optimized for fixed/floating-point)

## Topics

“Configure and Optimize Model with Configuration Wizard Blocks”

**Introduced in R2006b**

# ERT (optimized for floating-point)

Update active configuration parameters of parent model for ERT floating-point code generation

**Library:** Embedded Coder / Configuration Wizards

## Description

When you add an ERT (optimized for floating-point) block to your Simulink model and double-click it, a predefined MATLAB script executes and optimally configures the model parameters for floating-point code generation with the ERT target. Set a block parameter to invoke the build process after configuring the model.

After you double-click the block, open the Configuration Parameters dialog box to see that the model parameter values have changed.

**Note** To provide a quick way to switch between configurations, you can include more than one Configuration Wizard block in your model.

## Parameters

### `Configure the model for` — Configuration objective

`ERT (optimized for floating-point)` (default) | `ERT (optimized for fixed-point)` | `GRT (optimized for fixed/floating-point)` | `GRT (debug for fixed/floating-point)` | `Custom`

Objective for updating the active configuration parameters of a parent model by using a predefined MATLAB script.

### `Configuration function` — Custom MATLAB script

`rtwsampleconfig` (default)

Predefined or custom MATLAB script that updates the active configuration parameters of the parent model.

#### Dependencies

This parameter is only used with the Custom MATLAB file block. To enable this parameter, set **Configure the model for** to `Custom`.

#### `Invoke build process after configuration` — Initiate build process

`off` (default) | `on`

If selected, the predefined script initiates the code generation and build process after updating the configuration parameters.

## See Also

Custom MATLAB file | ERT (optimized for fixed-point) | GRT (debug for fixed/floating-point) | GRT (optimized for fixed/floating-point)

### Topics

"Configure and Optimize Model with Configuration Wizard Blocks"

**Introduced in R2006b**

# GRT (debug for fixed/floating-point)

Update active configuration parameters of parent model for GRT fixed-point or floating-point code generation

**Library:** Embedded Coder / Configuration Wizards

Configure Model
(double-click to activate)
GRT (debug for fixed/floating-point)

## Description

When you add a GRT (debug for fixed/floating-point) block to your Simulink model and double-click it, a predefined MATLAB script executes and optimally configures the model parameters for fixed-point or floating-point code generation, with TLC debugging options enabled, with the GRT target. Set a block parameter to invoke the build process after configuring the model.

After you double-click the block, open the Configuration Parameters dialog box to see that the model parameter values have changed.

---

**Note** To provide a quick way to switch between configurations, you can include more than one Configuration Wizard block in your model.

---

## Parameters

### `Configure the model for` — Configuration objective

`GRT (debug for fixed/floating-point)` (default) | `ERT (optimized for fixed-point)` | `ERT (optimized for floating-point)` | `GRT (optimized for fixed/floating-point)` | `Custom`

Objective for updating the active configuration parameters of a parent model by using a predefined MATLAB script.

### `Configuration function` — Custom MATLAB script

`rtwsampleconfig` (default)

Predefined or custom MATLAB script that updates the active configuration parameters of the parent model.

#### Dependencies

This parameter is only used with the Custom MATLAB file block. To enable this parameter, set **Configure the model for** to `Custom`.

#### `Invoke build process after configuration` — Initiate build process

`off` (default) | `on`

If selected, the predefined script initiates the code generation and build process after updating the configuration parameters.

## See Also

Custom MATLAB file | ERT (optimized for fixed-point) | ERT (optimized for floating-point) | GRT (optimized for fixed/floating-point)

### Topics

"Configure and Optimize Model with Configuration Wizard Blocks"

**Introduced in R2006b**

# GRT (optimized for fixed/floating-point)

Update active configuration parameters of parent model for GRT fixed-point or floating-point code generation

**Library:** Embedded Coder / Configuration Wizards

## Description

When you add a GRT(optimized for fixed-point) block to your Simulink model and double-click it, a predefined MATLAB script executes and optimally configures the model parameters for fixed-point or floating-point code generation with the GRT target. Set a block parameter to invoke the build process after configuring the model.

After you double-click the block, open the Configuration Parameters dialog box to see that the model parameter values have changed.

**Note** To provide a quick way to switch between configurations, you can include more than one Configuration Wizard block in your model.

## Parameters

### `Configure the model for` — Configuration objective

`GRT (optimized for fixed/floating-point)` (default) | `ERT (optimized for fixed-point)` | `ERT (optimized for floating-point)` | `GRT (debug for fixed/floating-point)` | `Custom`

Objective for updating the active configuration parameters of a parent model by using a predefined MATLAB script.

### `Configuration function` — Custom MATLAB script

`rtwsampleconfig` (default)

Predefined or custom MATLAB script that updates the active configuration parameters of the parent model.

#### Dependencies

This parameter is only used with the Custom MATLAB file block. To enable this parameter, set **Configure the model for** to `Custom`.

#### `Invoke build process after configuration` — Initiate build process

`off` (default) | `on`

If selected, the predefined script initiates the code generation and build process after updating the configuration parameters.

## See Also

Custom MATLAB file | ERT (optimized for fixed-point) | ERT (optimized for floating-point) | GRT (debug for fixed/floating-point)

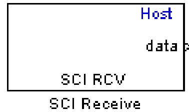
### Topics

"Configure and Optimize Model with Configuration Wizard Blocks"

**Introduced in R2006b**

# Host SCI Receive

Configure host-side serial communications interface to receive data from serial port




## Library

Embedded Coder/ Embedded Targets/ Host Communication

## Description

Specify the configuration of data being received from the target by this block.

The data package being received is limited to 16 bytes of ASCII characters, including package headers and terminators. Calculate the size of a package by including the package header, or terminator, or both, and the data size.

Acceptable data types are single, `int8`, `uint8`, `int16`, `uint16`, `int32`, or `uint32`. The number of bytes in each data type is listed in the following table:

| Data Type | Byte Count |
|---|---|
| `single` | 4 bytes |
| `int8` and `uint8` | 1 byte |
| `int16` and `uint16` | 2 bytes |
| `int32` and`uint32` | 4 bytes |

For example, if your data package has package header 'S' (1 byte) and package terminator 'E' (1 byte), that leaves 14 bytes for the actual data. If your data is of type `int8`, there is room in the data package for 14 `int8`s. If your data is of type `uint16`, there is room in the data package for 7 `uint16`s. If your data is of type `int32`, there is room in the data package for only 3 `int32`s, with 2 bytes left over. Even though you could fit two `int8`s or one `uint16` in the remaining space, you may not, because you cannot mix data types in the same package.

The number of data types that can fit into a data package determine the data length (see **Data length** in the Dialog Box description). In the example just given, the 14 for data type `int8` and the 7 for data type `uint16` are the data lengths for each data package, respectively. When the data length exceeds 16 bytes, unexpected behavior, including run time errors, may result.

## Parameters

**Port name**

You may configure up to four COM ports (COM1 through COM4) for up to four host-side SCI Receive blocks.

**Additional package header**

This field specifies the data located at the front of the received data package, which is not part of the data being received, and generally indicates start of data. The additional package header must be an ASCII value. You can use text or a number (0–255). You must put single quotes around text entered in this field, but the quotes are not received nor are they included in the total byte count.

**Note** Match additional package headers or terminators with those specified in the target SCI transmit block.

**Additional package terminator**

This field specifies the data located at the end of the received data package, which is not part of the data being received, and generally indicates end of data. The additional package terminator must be an ASCII value. You can use text or a number (0–255). You must put single quotes around text entered in this field, but the quotes are not received nor are they included in the total byte count.

**Data type**

Choice of single, int8, uint8, int16, uint16, int32, or uint32.

The input port of the SCI Transmit block accepts only one of these values. Which value it accepts is inherited from the data type from the input (the data length is also inherited from the input). Data must consist of only one data type; you cannot mix types.

**Data length**

How many of **Data type** the block receives (not bytes). Anything more than 1 is a vector. The data length is inherited from the input (the data length input to the SCI Transmit block).

**Initial output**

Default value from the SCI Receive block. This value is used, for example, if a connection time-out occurs and the **Action taken when connection timeout** field is set to “Output the last received value”, but nothing yet has been received.

**Action Taken when connection times out**

Specify what to output if a connection time-out occurs. If “Output the last received value” is selected, the block outputs the last received value. If a value has not been received, the block outputs the **Initial output**.

If you select `Output custom value`, use the **Output value when connection times out** field to set the custom value.

**Sample time**

Determines how often the SCI Receive block is called (in seconds). When you set this value to `-1`, the model inherits the sample time value of the model. To execute this block asynchronously, set **Sample Time** to `-1`.

**Output receiving status**

Selecting this check box creates a **Status** block output that provides the status of the transaction.

The error status may be one of the following values:

- `0`: No errors
- `1`: A time-out occurred while the block was waiting to receive data
- `2`: There is an error in the received data (checksum error)
- `3`: SCI parity error flag — Occurs when a character is received with a mismatch
- `4`: SCI framing error flag — Occurs when an expected stop bit is not found

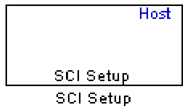
# See Also

Host SCI Transmit

**Introduced in R2011a**

# Host SCI Setup

Configure COM ports for host-side SCI Transmit and Receive blocks




## Library

Embedded Coder/ Embedded Targets/ Host Communication

## Description

Standardize COM port settings for use by the host-side SCI Transmit and Receive blocks. Setting COM port configurations globally with the SCI Setup block avoids conflicts (e.g., the host-side SCI Transmit block cannot use COM1 with settings different than those the COM1 used by the host-side SCI Receive block) and requires that you set configurations only once for each COM port. The SCI Setup block is a stand alone block.

## Parameters

**Communication Mode**

Raw data or protocol. Raw data is unformatted and sent whenever the transmitting side is ready to send, whether the receiving side is ready or not. Without a wait state, deadlocks do not occur. Data transmission is asynchronous. With this mode, it is possible the receiving side could miss data, but if the data is noncritical, using raw data mode can avoid blocking processes.

If you specify protocol mode, some handshaking between host and target occurs. The transmitting side sends $SND indicating that it is ready to transmit. The receiving side sends back $RDY indicating that it is ready to receive. The transmitting side then sends data and, when the transmission is completed, it sends a checksum.

Advantages to using protocol mode include

- Data is received as expected (checksum)
- Data is received by target
- Time consistency; each side waits for its turn to send or receive

---

**Note** Deadlocks can occur if one SCI Transmit block is trying to communicate with more than one SCI Receive block on different COM ports when both are blocking (using protocol mode). Deadlocks cannot occur on the same COM port.

---

**Baud rate**

Choose from 110, 300, 1200, 2400, 4800, 9600, 19200, 38400, 57600, or 115200.

**Number of stop bits**

Select 1 or 2.

**Parity mode**

Select none, odd, or even.

**Timeout**

Enter values greater than or equal to 0, in seconds. When the COM port involved is using protocol mode, this value indicates how long the transmitting side waits for an acknowledgement from the receiving side or how long the receiving side waits for data. The system displays a warning message if the time-out is exceeded, every *n* number of seconds, *n* being the value in **Timeout**.

---

**Note** Simulink suspends processing for the length of the time-out. During that time you cannot perform actions in Simulink. If the time-out is set for a long period of time, it may appear that Simulink has frozen.

---

## See Also

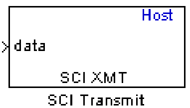
Host SCI Receive

Host SCI Transmit

**Introduced in R2011a**

# Host SCI Transmit

Configure host-side serial communications interface to transmit data to serial port




## Library

Embedded Coder/ Embedded Targets/ Host Communication

## Description

Specify the configuration of data being transmitted to the target from this block.

The data package being sent is limited to 16 bytes of ASCII characters, including package headers and terminators. Calculate the size of a package by figuring in package header, or terminator, or both, and the data size.

Acceptable data types are `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, or `uint32`. The byte size of each data type is as follows:

| Data Type | Byte Count |
|---|---|
| `single` | 4 bytes |
| `int8` & `uint8` | 1 byte |
| `int16` & `uint16` | 2 bytes |
| `int32` & `uint32` | 4 bytes |

For example, if your data package has package header “S” (1 byte) and package terminator “E” (1 byte), that leaves 14 bytes for the actual data. If your data is of type `int8`, there is room in the data package for 14 `int8`s. If your data is of type `uint16`, there is room in the data package for only 7 `uint16`s. If your data is of type `int32`, there is room in the data package for only 3 `int32`s, with 2 bytes left over. Even though you could fit two `int8`s or one `uint16` in the remaining space, you may not, because you cannot mix data types in the same package.

The number of data types that can fit into a data package determine the data length (see **Data length** in the Dialog Box description). In the example just given, the 14 for data type int8 and the 7 for data type uint16 are the data lengths for each data package, respectively. When the data length exceeds 16 bytes, unexpected behavior, including run time errors, may result.

## Parameters

**Port name**

You may configure up to four COM ports (COM1 through COM4) for up to four host-side SCI Transmit blocks.

**Additional package header**

This field specifies the data located at the front of the transmitted data package, which is not part of the data being transmitted, and generally indicates start of data. The additional package header must be an ASCII value. You can use text or a number (0–255). You must put single quotes around text entered in this field, but the quotes are not sent nor are they included in the total byte count.

---

**Note** Match additional package headers or terminators with those specified in the target SCI receive block.

---

**Additional package terminator**

This field specifies the data located at the end of the transmitted data package, which is not part of the data being sent, and generally indicates end of data. The additional package terminator must be an ASCII value. You can use text or a number (0–255). You must put single quotes around text entered in this field, but the quotes are not transmitted nor are they included in the total byte count.

## See Also

Host SCI Receive

**Introduced in R2011a**

# Idle Task

Create free-running task

## Description

The Idle Task block, and the subsystem connected to it, specify one or more functions to execute as background tasks. The tasks executed through the Idle Task block are of the lowest priority, lower than that of the base rate task.

This block is not supported on targets running an operating system or RTOS.

### Vectorized Output

The block output comprises a set of vectors—the task numbers vector and the preemption flag or flags vector. A preemption-flag vector must be the same length as the number of tasks vector unless the preemption flag vector has only one element. The value of the preemption flag determines whether a given interrupt (and task) is preemptible. Preemption overrides prioritization. A lower-priority nonpreemptible task can preempt a higher-priority preemptible task.

When the preemption flag vector has one element, that element value applies to the functions in the downstream subsystem as defined by the task numbers in the task number vector. If the preemption flag vector has the same number of elements as the task number vector, each task defined in the task number vector has a preemption status defined by the value of the corresponding element in the preemption flag vector.

# Parameters

**Task numbers**

Identifies the created tasks by number. Enter as many tasks as you need by entering a vector of integers. The default values are `[1,2]` to indicate that the downstream subsystem has two functions.

The values you enter determine the execution order of the functions in the downstream subsystem, while the number of values you enter corresponds to the number of functions in the downstream subsystem.

Enter a vector containing the same number of elements as the number of functions in the downstream subsystem. This vector can contain up to 16 elements, and the values must be from 0 to 15 inclusive.

The value of the first element in the vector determines the order in which the first function in the subsystem is executed, the value of the second element determines the order in which the second function in the subsystem is executed, and so on.

For example, entering [2,3,1] in this field indicates that there are three functions to be executed, and that the third function is executed first, the first function is executed second, and the second function is executed third. After the functions are executed, the Idle Task block cycles back and repeats the execution of the functions in the same order.

**Preemption flags**

Higher-priority interrupts can preempt interrupts that have lower priority. To allow you to control preemption, use the preemption flags to specify whether an interrupt can be preempted.

Entering `1` indicates that the interrupt can be preempted. Entering `0` indicates the interrupt cannot be preempted. When **Task numbers** contains more than one task, you can assign different preemption flags to each task by entering a vector of flag values, corresponding to the order of the tasks in **Task numbers**. If **Task numbers** contains more than one task, and you enter only one flag value here, that status applies to the tasks.

In the default settings `[0 1]`, the task with priority `1` in **Task numbers** is not preemptible, and the priority `2` task can be preempted.

**Enable simulation input**

When you select this option, Simulink software adds an input port to the Idle Task block. This port is used in simulation only. Connect one or more simulated interrupt sources to the simulation input.

---

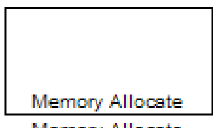
**Note** Select this check box to test asynchronous interrupt processing behavior in Simulink software.

---

**Introduced in R2011a**

# Memory Allocate

Allocate memory section




## Description

On C2xxx, C5xxx, or C6xxx processors, this block directs the TI compiler to allocate memory for a new variable you specify. Parameters in the block dialog box let you specify the variable name, the alignment of the variable in memory, the data type of the variable, and other features that fully define the memory required.

The block does not verify whether the entries for your variable are valid, such as checking the variable name, data type, or section. You must check that all variable names are valid, that they use valid data types, and that all section names you specify are valid as well.

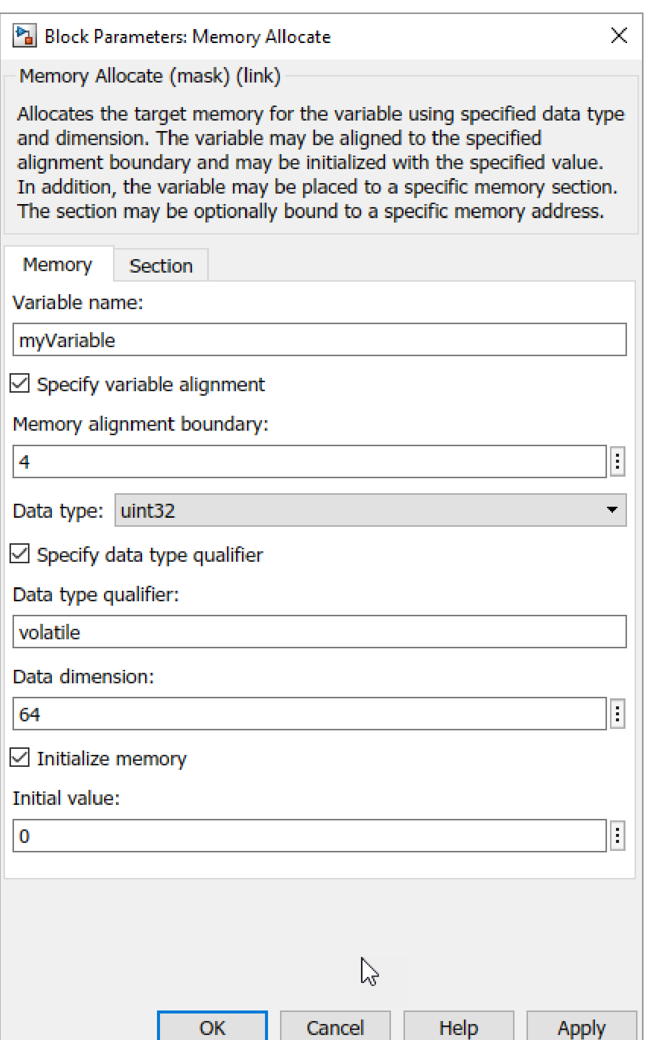
The block does not have input or output ports. It only allocates a memory location. You do not connect it to other blocks in your model.

## Dialog Box

The block dialog box comprises multiple tabs:

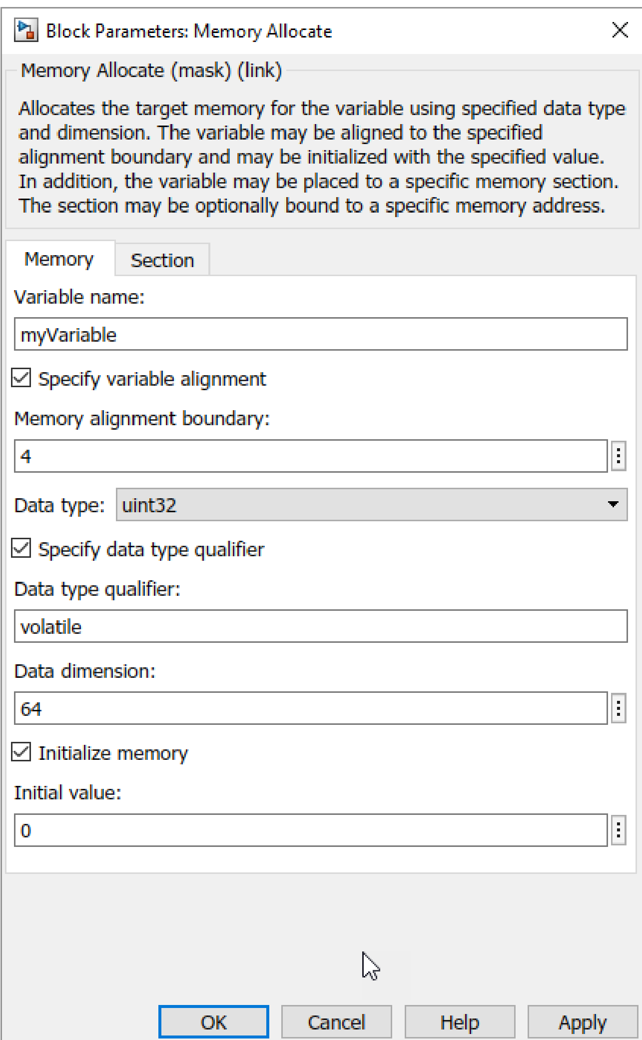
- **Memory** — Allocate the memory for storing variables. Specify the data type and size.
- **Section** — Specify the memory section in which to allocate the variable.

The dialog box images show all of the available parameters enabled. Some of the parameters shown do not appear until you select one or more other parameters.




The following sections describe the contents of each pane in the dialog box.

## Memory Parameters




You find the following memory parameters on this tab.

**Variable name**

Specify the name of the variable to allocate. The variable is allocated in the generated code.

**Specify variable alignment**

Select this option to direct the compiler to align the variable in **Variable name** to an alignment boundary. When you select this option, the **Memory alignment boundary** parameter appears so you can specify the alignment. Use this parameter and **Memory alignment boundary** when your processor requires this feature.

**Memory alignment boundary**

After you select **Specify variable alignment**, this option enables you to specify the alignment boundary in bytes. If your variable contains more than one value, such as a vector or an array, the elements are aligned according to rules applied by the compiler.

**Data type**

Defines the data type for the variable. Select from the list of types available.

**Specify data type qualifier**

Selecting this enables **Data type qualifier** so you can specify the qualifier to apply to your variable.

**Data type qualifier**

After you select **Specify data type qualifier**, you enter the desired qualifier here. `Volatile` is the default qualifier. Enter the qualifier you need as text. Common qualifiers are `static` and `register`. The block does not check for valid qualifiers.

**Data dimension**

Specifies the number of elements of the type you specify in **Data type**. Enter an integer here for the number of elements.

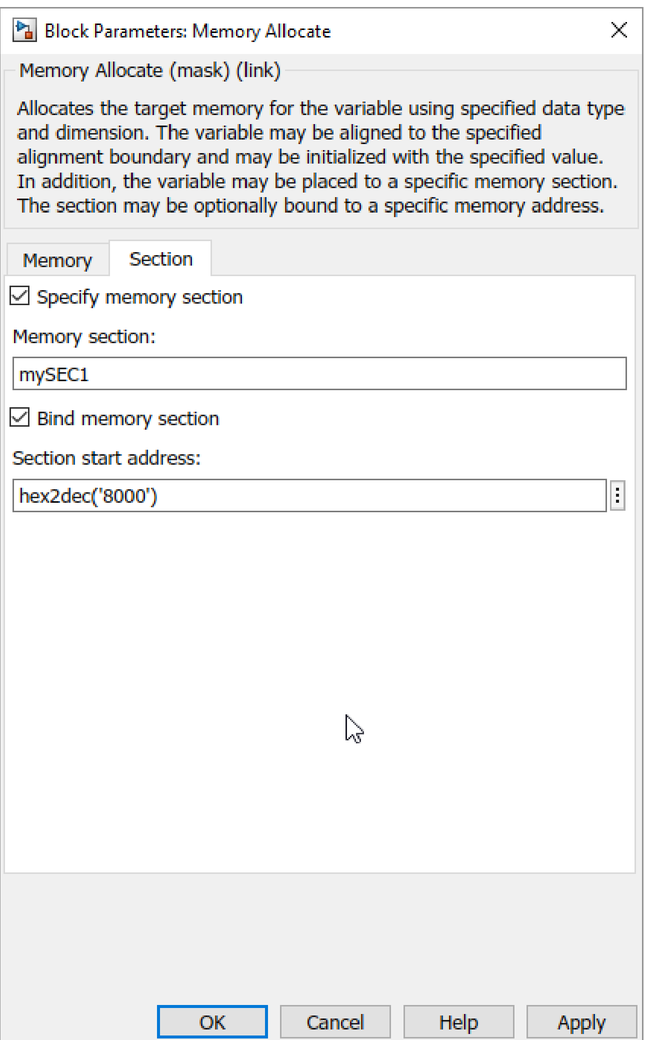
**Initialize memory**

Directs the block to initialize the memory location to a fixed value before processing.

**Initial value**

Specifies the initialization value for the variable. At run time, the block sets the memory location to this value.

## Section Parameters




Parameters on this pane specify the section in memory to store the variable.

**Specify memory section**

Selecting this parameter enables you to specify the memory section to allocate space for the variable. Enter either one of the standard memory sections or a custom section that you declare elsewhere in your code.

**Memory section**

Identify a specific memory section to allocate the variable in **Variable name**. Verify that the section has enough space to store your variable. After you specify a memory section by selecting **Specify memory section** and entering the section name in **Memory section**, use **Bind memory section** to bind the memory section to a location.

**Bind memory section**

After you specify a memory section by selecting **Specify memory section** and entering the section name in **Memory section**, use this parameter to bind the memory section to the location in memory specified in **Section start address**. When you select this, you enable the **Section start address** parameter.

The new memory section specified in **Memory section** is defined when you check this parameter.

---

**Note** Do not use **Bind memory section** for existing memory sections.

---

**Section start address**

Specify the address to which to bind the memory section. Enter the address in decimal form or in hexadecimal with a conversion to decimal as shown by the default value `hex2dec('8000')`. The block does not verify the address—verify that the address exists and can contain the memory section you entered in **Memory section**.

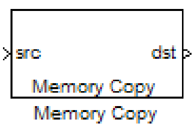
# See Also

Memory Copy

**Introduced in R2011a**

# Memory Copy

Copy to and from memory section




## Description

In generated code, this block copies variables or data from and to processor memory as configured by the block parameters. Your model can contain as many of these blocks as you require to manipulate memory on your processor.

Each block works with one variable, address, or set of addresses provided to the block. Parameters for the block let you specify both the source and destination for the memory copy, as well as options for initializing the memory locations.

Using parameters provided by the block, you can change options like the memory stride and offset at run time. In addition, by selecting various parameters in the block, you can write to memory at program initialization, at program termination, and at every sample time. The initialization process occurs once, rather than occurring for every read and write operation.

With the custom source code options, the block enables you to add custom ANSI C source code before and after each memory read and write (copy) operation. You can use the custom code capability to lock and unlock registers before and after accessing them. For example, some processors have registers that you may need to unlock and lock with EALLOW and EDIS macros before and after your program accesses them.

If your processor or board supports quick direct memory access (QDMA) the block provides a parameter to check that implements the QDMA copy operation, and enables you to specify a function call that can indicate that the QDMA copy is finished. Only the C621x, C64xx, and C671x processor families support QDMA copy.

---

**Note** Replace Read from Memory and Write To Memory blocks, which were removed in a previous release, with the Memory Copy block.

---

## Block Operations

This block performs operations at three periods during program execution—initialization, real-time operations, and termination. With the options for setting memory initialization and termination, you control when and how the block initializes memory, copies to and from memory, and terminates memory operations. The parameters enable you to turn on and off memory operations in the three periods independently.

Used in combination with the Memory Allocate block, this block supports building custom device drivers, such as PCI bus drivers or codec-style drivers, by letting you manipulate and allocate memory. This block does not require the Memory Allocate block to be in the model.

In a simulation, this block does not perform an operation. The block output is not defined.

## Copy Memory

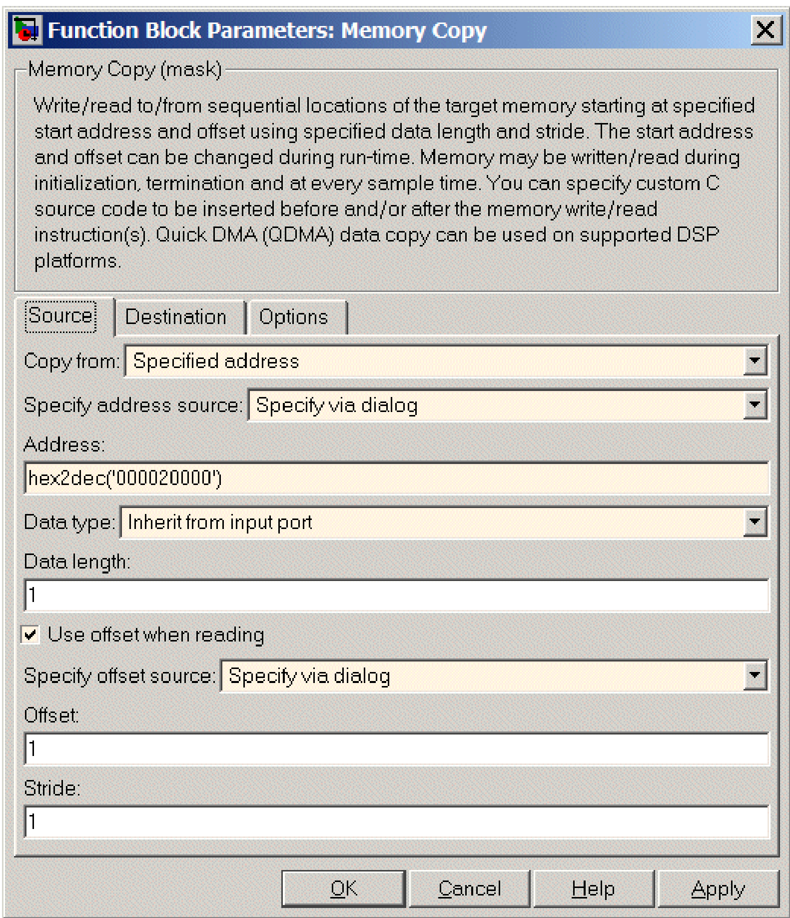
When you employ this block to copy an individual data element from the source to the destination, the block copies the element from the source in the source data type, and then casts the data element to the destination data type as provided in the block parameters.

# Dialog Box

The block dialog box contains multiple tabs:

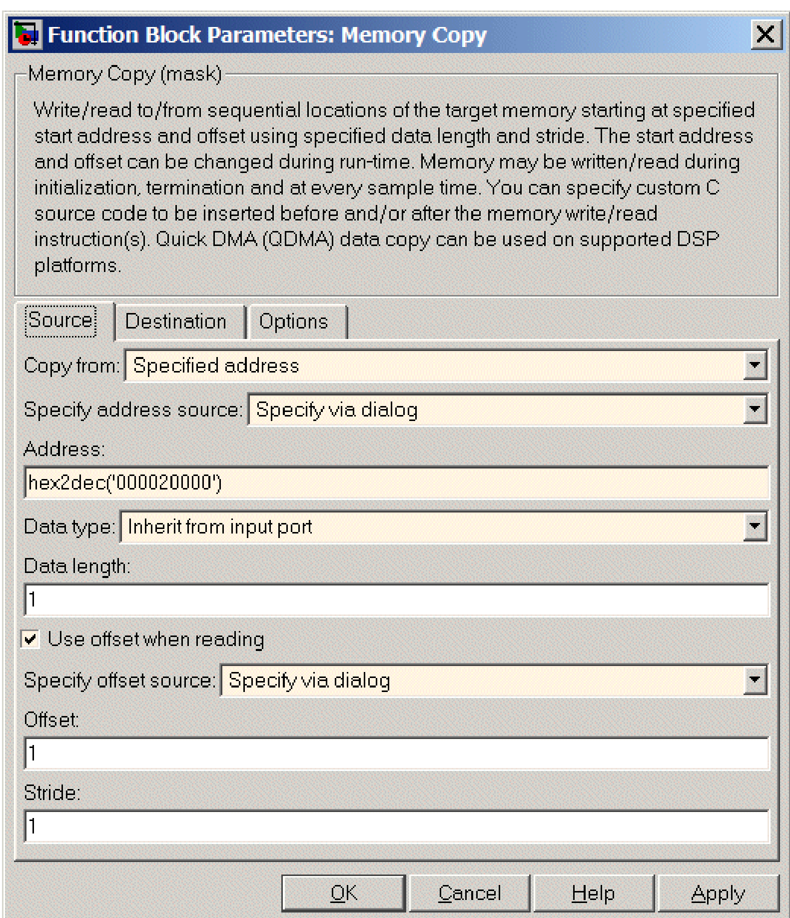
- **Source** — Identifies the sequential memory location to copy from. Specify the data type, size, and other attributes of the source variable.
- **Destination** — Specify the memory location to copy the source to. Here you also specify the attributes of the destination.
- **Options** — Select various parameters to control the copy process.

The dialog box images show many of the available parameters enabled. Some parameters shown do not appear until you select one or more other parameters. Some parameters are not shown in the figures, but the text describes them and how to make them available.




Sections that follow describe the parameters on each tab in the dialog box.

## Source Parameters




### Copy from

Select the source of the data to copy. Choose one of the entries on the list:

- `Input port` — This source reads the data from the block input port.
- `Specified address` — This source reads the data at the specified location in **Specify address source** and **Address**.

- `Specified source code symbol` — This source tells the block to read the symbol (variable) you enter in **Source code symbol**. When you select this copy from option, you enable the **Source code symbol** parameter.

  ---

  **Note** If you do not select `Input port` for **Copy from**, change **Data type** from the default `Inherit from source` to one of the data types on the **Data type** list. If you do not make the change, you receive an error message that the data type cannot be inherited because the input port does not exist.

  ---

Depending on the choice you make for **Copy from**, you see other parameters that let you configure the source of the data to copy.

**Specify address source**

This parameter directs the block to get the address for the variable either from an entry in **Address** or from the input port to the block. Select either `Specify via dialog` or `Input port` from the list. Selecting `Specify via dialog` activates the **Address** parameter for you to enter the address for the variable.

When you select `Input port`, the port label on the block changes to `&src`, indicating that the block expects the address to come from the input port. Being able to change the address dynamically lets you use the block to copy different variables by providing the variable address from an upstream block in your model.

**Source code symbol**

Specify the symbol (variable) in the source code symbol table to copy. The symbol table for your program must include this symbol. The block does not verify that the symbol exists and uses valid syntax. Enter text to specify the symbol exactly as you use it in your code.

**Address**

When you select Specify via dialog for the address source, you enter the variable address here. Addresses should be in decimal form. Enter either the decimal address or the address as a hexadecimal character vector with single quotations marks and use `hex2dec` to convert the address to the expected format. The following example converts `0x1000` to decimal form.

```
4096 = hex2dec('1000');
```

For this example, you could enter either `4096` or `hex2dec('1000')` as the address.

**Data type**

Use this parameter to specify the type of data that your source uses. The list includes the supported data types, such as `int8`, `uint32`, and `Boolean`, and the option `Inherit from source` for inheriting the data type from the block input port.

**Data length**

Specifies the number of elements to copy from the source location. Each element has the data type specified in **Data type**.

**Use offset when reading**

When you are reading the input, use this parameter to specify an offset for the input read. The offset value is in elements with the assigned data type. The **Specify offset source** parameter becomes available when you check this option.

**Specify offset source**

The block provides two sources for the offset — `Input port` and `Specify via dialog`. Selecting `Input port` configures the block input to read the offset value by adding an input port labeled `src ofs`. This port enables your program to change the offset dynamically during execution by providing the offset value as an input to the block. If you select `Specify via dialog`, you enable the **Offset** parameter in this dialog box so you can enter the offset to use when reading the input data.

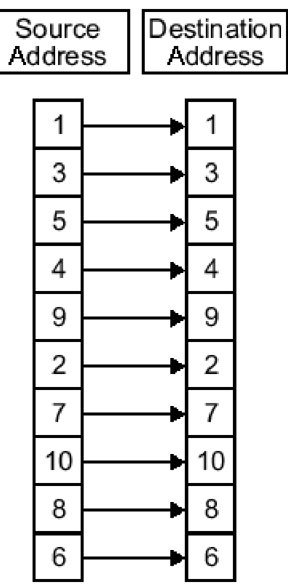
**Offset**

**Offset** tells the block whether to copy the first element of the data at the input address or value, or skip one or more values before starting to copy the input to the destination. **Offset** defines how many values to skip before copying the first value to the destination. Offset equal to one is the default value and **Offset** accepts only positive integers of one or greater.

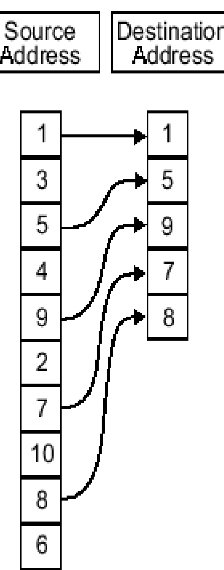
**Stride**

Stride lets you specify the spacing for reading the input. By default, the stride value is one, meaning the generated code reads the input data sequentially. When you add a stride value that is not equal to one, the block reads the input data elements not sequentially, but by skipping spaces in the source address equal to the stride. **Stride** must be a positive integer.

The next two figures help explain the stride concept. In the first figure you see data copied without a stride. Following that figure, the second figure shows a stride value of two applied to reading the input when the block is copying the input to an output location. You can specify a stride value for the output with parameter **Stride** on the **Destination** pane. Compare stride with offset to see the differences.

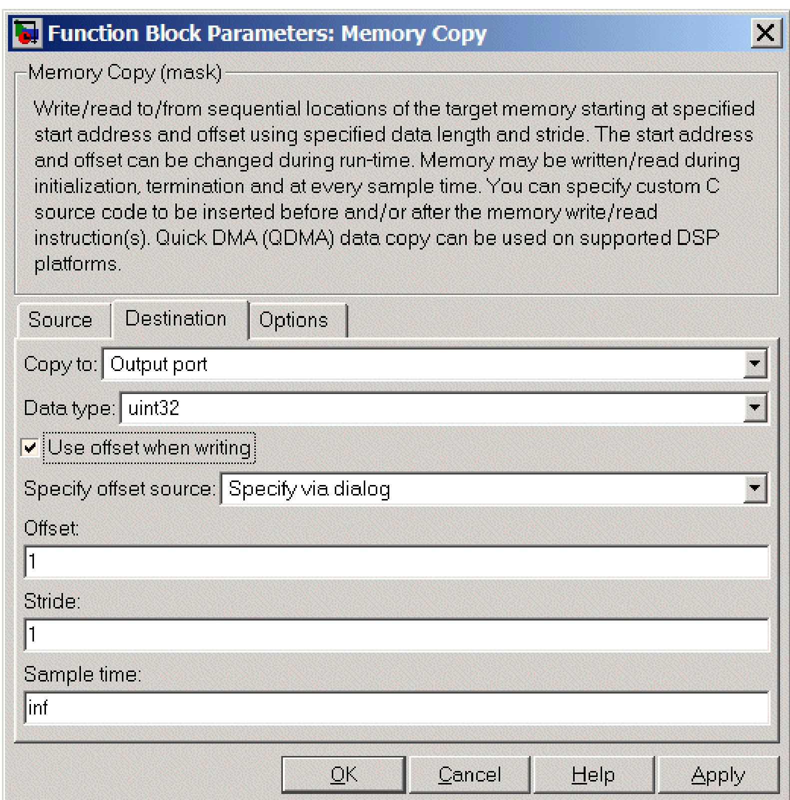



Input Stride = 1
Output Stride = 1
Number of Elements Copied = 10




Input Stride = 2
Output Stride = 1
Number of Elements Copied = 5

## Destination Parameters




**Copy to**

Select the destination for the data. Choose one of the entries on the list:

- `Output port` — Copies the data to the block output port. From the output port the block passes data to downstream blocks in the code.
- `Specified address` — Copies the data to the specified location in **Specify address source** and **Address**.
- `Specified source code symbol` — Tells the block to copy the variable or symbol (variable) to the symbol you enter in **Source code symbol**. When you select this copy to option, you enable the **Source code symbol** parameter.

Depending on the choice you make for **Copy from**, you see other parameters that let you configure the source of the data to copy.

**Specify address source**

This parameter directs the block to get the address for the variable either from an entry in **Address** or from the input port to the block. Select either `Specify via dialog` or `Input port` from the list. Selecting `Specify via dialog` activates the **Address** parameter for you to enter the address for the variable.

When you select `Input port`, the port label on the block changes to `&dst`, indicating that the block expects the destination address to come from the input port. Being able to change the address dynamically lets you use the block to copy different variables by providing the variable address from an upstream block in your model.

**Source code symbol**

Specify the symbol (variable) in the source code symbol table to copy. The symbol table for your program must include this symbol. The block does not verify that the symbol exists and uses valid syntax.

**Address**

When you select Specify via dialog for the address source, you enter the variable address here. Addresses should be in decimal form. Enter either the decimal address or the address as a hexadecimal character vector with single quotations marks and use `hex2dec` to convert the address to the expected format. This example converts `0x2000` to decimal form.

```
8192 = hex2dec('2000');
```

For this example, you could enter either `8192` or `hex2dec('2000')` as the address.

**Data type**

Use this parameter to specify the type of data that your variable uses. The list includes the supported data types, such as `int8`, `uint32`, and `Boolean`, and the option `inherit from source` for inheriting the data type for the variable from the block input port.

**Specify offset source**

The block provides two sources for the offset—`Input port` and `Specify via dialog`. Selecting `Input port` configures the block input to read the offset value by adding an input port labeled `src ofs`. This port enables your program to change the offset dynamically during execution by providing the offset value as an input to the block. If you select `Specify via dialog`, you enable the **Offset** parameter in this dialog box so you can enter the offset to use when writing the output data.

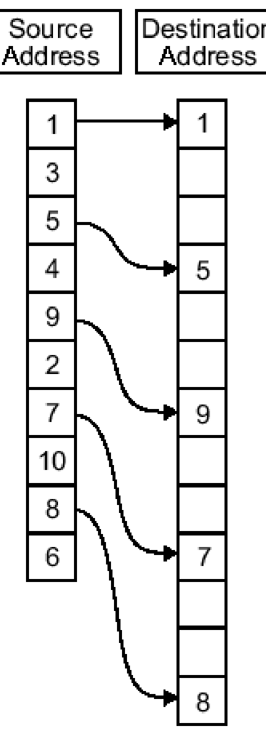
**Offset**

**Offset** tells the block whether to write the first element of the data to be copied to the first destination address location, or skip one or more locations at the destination before writing the output. **Offset** defines how many values to skip in the destination before writing the first value to the destination. One is the default offset value and **Offset** accepts only positive integers of one or greater.

**Stride**

Stride lets you specify the spacing for copying the input to the destination. By default, the stride value is one, meaning the generated code writes the input data sequentially to the destination in consecutive locations. When you add a stride value not equal to one, the output data is stored not sequentially, but by skipping addresses equal to the stride. **Stride** must be a positive integer.

This figure shows a stride value of three applied to writing the input to an output location. You can specify a stride value for the input with parameter **Stride** on the **Source** pane. As shown in the figure, you can use both an input stride and output stride at the same time to enable you to manipulate your memory more fully.

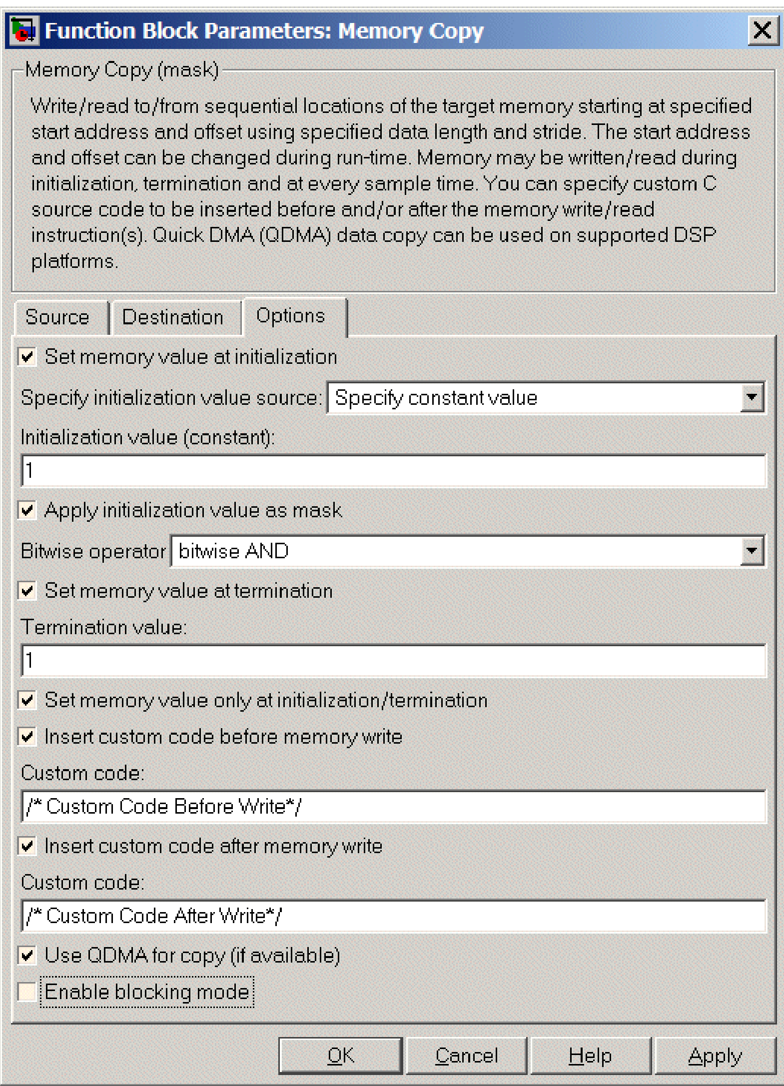



Input Stride = 2
Output Stride = 3
Number of Elements Copied = 5

**Sample time**

**Sample time** sets the rate at which the memory copy operation occurs, in seconds. The default value `Inf` tells the block to use a constant sample time. You can set **Sample time** to `-1` to direct the block to inherit the sample time from the input, or from the Simulink software model when there are no block inputs. Enter the sample time in seconds as you need.

## Options Parameters




### Set memory value at initialization

When you check this option, you direct the block to initialize the memory location to a specific value when you initialize your program at run time. After you select this

option, use the **Set memory value at termination** and **Specify initialization value source** parameters to set your desired value. Alternately, you can tell the block to get the initial value from the block input.

**Specify initialization value source**

After you check Set memory value at initialization, use this parameter to select the source of the initial value. Choose either

- `Specify constant value` — Sets a single value to use when your program initializes memory.
- `Specify source code symbol` — Specifies a variable (a symbol) to use for the initial value. Enter the symbol as a character vector.

**Initialization value (constant)**

If you check **Set memory value at initialization** and choose `Specify constant value` for **Specify initialization value source,** enter the constant value to use in this field.

**Initialization value (source code symbol)**

If you check **Set memory value at initialization** and choose `Specify source code symbol` for **Specify initialization value source,** enter the symbol to use in this field. Use a valid symbol from the symbol table for the program. When you enter the symbol, the block does not verify whether the symbol is a valid one. If it is not valid you get an error when you try to compile, link, and run your generated code.

**Apply initialization value as mask**

You can use the initialization value as a mask to manipulate register contents at the bit level. Your initialization value is treated as a string of bits for the mask.

Checking this parameter enables the **Bitwise operator** parameter for you to define how to apply the mask value.

To use your initialization value as a mask, the output from the copy has to be a specific address. It cannot be an output port, but it can be a symbol.

**Bitwise operator**

To use the initialization value as a mask, select one of the entries on the following table from the **Bitwise operator** list to describe how to apply the value as a mask to the memory value.

| Bitwise Operator List Entry | Description |
|---|---|
| `bitwise AND` | Apply the mask value as a bitwise AND to the value in the register. |
| `bitwise OR` | Apply the mask value as a bitwise OR to the value in the register. |
| `bitwise exclusive OR` | Apply the mask value as a bitwise exclusive OR to the value in the register. |
| `left shift` | Shift the bits in the register left by the number of bits represented by the initialization value. For example, if your initialization value is 3, the block shifts the register value to the left 3 bits. In this case, the value must be a positive integer. |
| `right shift` | Shift the bits in the register to the right by the number of bits represented by the initialization value. For example, if your initialization value is 6, the block shifts the register value to the right 6 bits. In this case, the value must be a positive integer. |

Applying a mask to the copy process lets you select individual bits in the result, for example, to read the value of the fifth bit by applying the mask.

**Set memory value at termination**

Along with initializing memory when the program starts to access this memory location, this parameter directs the program to set memory to a specific value when the program terminates.

**Set memory value only at initialization/termination**

This block performs operations at three periods during program execution—initialization, real-time operations, and termination. When you check this option, the block only does the memory initialization and termination processes. It does not perform copies during real-time operations.

**Insert custom code before memory write**

Select this parameter to add custom ANSI C code before the program writes to the specified memory location. When you select this option, you enable the **Custom code** parameter where you enter your ANSI C code.

**Custom code**

Enter the custom ANSI C code to insert into the generated code just before the memory write operation. Code you enter in this field appears in the generated code exactly as you enter it.

**Insert custom code after memory write**

Select this parameter to add custom ANSI C code immediately after the program writes to the specified memory location. When you select this option, you enable the **Custom code** parameter where you enter your ANSI C code.

**Custom code**

Enter the custom ANSI C code to insert into the generated code just after the memory write operation. Code you enter in this field appears in the generated code exactly as you enter it.

**Use QDMA for copy (if available)**

For processors that support quick direct memory access (QDMA), select this parameter to enable the QDMA operation and to access the blocking mode parameter.

If you select this parameter, your source and destination data types must be the same or the copy operation returns an error. Also, the input and output stride values must be one.

**Enable blocking mode**

If you select the **Use QDMA for copy** parameter, select this option to make the memory copy operations blocking processes. With blocking enabled, other processing in the program waits while the memory copy operation finishes.

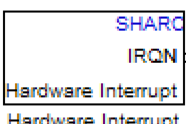
# See Also

Memory Allocate

**Introduced in R2011a**

# SHARC Hardware Interrupt

Generate Interrupt Service Routine




## Library

Embedded Coder/ Embedded Targets/ Processors/ Analog Devices SHARC®/ Scheduling

## Description

Create interrupt service routines (ISR) in the software generated by the build process. When you incorporate this block in your model, code generation results in ISRs on the processor that either run the processes that are downstream from this block or trigger an Idle Task block connected to this block.

## Parameters

**Interrupt numbers**

Specify an array of interrupt numbers for the interrupts to install. The valid ranges are 8-36 and 38-40.

The width of the block output signal corresponds to the number of interrupt numbers specified in this field. The values in this field and the preemption flag entries in **Preemption flags: preemptible-1, non-preemptible-0** define how the code and processor handle interrupts during asynchronous scheduler operations.

**Simulink task priorities**

Each output of the Hardware Interrupt block drives a downstream block (for example, a function call subsystem). Simulink model task priority specifies the priority of the downstream blocks. Specify an array of priorities corresponding to the interrupt numbers entered in **Interrupt numbers**.

Code generation requires rate transition code. Refer to Rate Transitions and Asynchronous Blocks (Simulink Coder). The task priority values facilitate absolute time integrity when the asynchronous task must obtain real time from its base rate or its caller. Typically, assign priorities for these asynchronous tasks that are higher than the priorities assigned to periodic tasks.

**Preemption flags preemptible – 1, non-preemptible – 0**

Higher-priority interrupts can preempt interrupts that have lower priority. To allow you to control preemption, use the preemption flags to specify whether an interrupt can be preempted.

- Entering `1` indicates that the interrupt can be preempted.
- Entering `0` indicates the interrupt cannot be preempted.

When **Interrupt numbers** contains more than one interrupt value, you can assign different preemption flags to each interrupt by entering a vector of flag values to correspond to the order of the interrupts in **Interrupt numbers**. If **Interrupt numbers** contains more than one interrupt, and you enter only one flag value in this field, that status applies to all interrupts.

In the default settings `[0 1]`, the interrupt with priority `18` in **Interrupt numbers** is not preemptible and the priority `39` interrupt can be preempted.

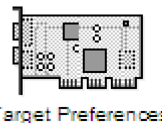
**Enable simulation input**

When you select this option, Simulink software adds an input port to the Hardware Interrupt block. This port is used in simulation only. Connect one or more simulated interrupt sources to the simulation input.

**Introduced in R2011a**

# Target Preferences (Removed)

Configure model for specific IDE, tool chain, board, and processor




## Library

Simulink Coder / Desktop Targets

Embedded Coder/ Embedded Targets

## Description

The Target Preferences block has been removed from the Simulink block libraries. The contents of the Target Preferences block have been moved to the Target Hardware Resources tab, located in the Configuration Parameters dialog. For more information, see:

- “Hardware configuration relocation from Target Preferences block to Configuration Parameters dialog box”
- “Configure Target Hardware Resources”
- “Code Generation: Coder Target Pane” on page 13-2

**Introduced in R2013a**

# UDP Receive

Receive UDP packet

## Block Library

Embedded Coder/Embedded Targets/Host Communication

## Description

The UDP Receive block receives UDP packets from an IP network port and saves them to its buffer. With each sample, the block outputs the contents of a single UDP packet as a data vector. The local IP port number on which the block receives the UDP packets is tunable in the C/C++ generated code.

The generated code for this block relies on prebuilt `.dll` files. You can run this code outside the MATLAB environment, or redeploy it, but you must account for these extra `.dll` files when doing so. The `packNGo` function creates a single zip file containing all of the pieces required to run or rebuild this code. For more details, see "How To Run a Generated Executable Outside MATLAB" (DSP System Toolbox).

## Parameters

**Local IP port**

Specify the IP port number on which to receive UDP packets. This parameter is tunable in the C/C++ generated code but not tunable during simulation. The default is `25000`. The value can be in the range [1 65535].

---

**Note** On Linux®, to set the IP port number below 1024, run MATLAB with root privileges. For example, at the Linux command line, enter:

```
sudo matlab
```

---

**Remote IP address ('0.0.0.0' to accept all)**

Specify the IP address from which to accept packets. Entering a specific IP address blocks UDP packets from other addresses. To accept packets from any IP address, enter `'0.0.0.0'`. This value defaults to `'0.0.0.0'`.

**Receive buffer size (bytes)**

Make the receive buffer large enough to avoid data loss caused by buffer overflows. This value defaults to `8192`.

**Maximum length for Message**

Specify the maximum length, in vector elements, of the data output vector. Set this parameter to a value equal or greater than the data size of a UDP packet. The system truncates data that exceeds this length. This value defaults to `255`.

If you disable **Output variable-size signal**, the block outputs a fixed-length output the same length as the **Maximum length for Message**.

**Data type for Message**

Set the data type of the vector elements in the Message output. Match the data type with the data input used to create the UDP packets. This option defaults to `uint8`.

**Output variable-size signal**

If your model supports signals of varying length, enable the **Output variable-size signal** parameter. This checkbox defaults to selected (enabled). In that case:

- The output vector varies in length, depending on the amount of data in the UDP packet.
- The block emits the data vector from a single unlabeled output.

If your model does not support signals of varying length, disable the **Output variable-size signal** parameter. In that case:

- The block emits a fixed-length output the same length as the **Maximum length for Message**.
- If the UDP packet contains less data than the fixed-length output, the difference contains invalid data.

- The block emits the data vector from the **Message** output.
- The block emits the length of the valid data from the **Length** output.
- The block dialog box displays the **Data type for Length** parameter.

In both cases, the block truncates data that exceeds the **Maximum length for Message**.

**Blocking time (seconds)**

For each sample, wait this length of time for a UDP packet before returning control to the scheduler. This value defaults to `inf`, which indicates to wait indefinitely.

**Note** This parameter appears only in the Embedded Coder UDP Receive block.

**Sample time (seconds)**

Specify how often the scheduler runs this block. Enter a value greater than zero. In real-time operation, setting this option to a smaller value reduces the likelihood of dropped UDP messages. This value defaults to a sample time of 0.01 s.

# Deprecated Parameters

**Output port width**

Specify the width of packets the block accepts. When you design the transmit end of the UDP communication channel, you decide the packet width. Set this option to a value as large or larger than a packet you expect to receive.

**Note** This parameter appears only in a deprecated version of the UDP Receive block. Replace the deprecated UDP Receive block with a current UDP Receive block.

**UDP receive buffer size (bytes)**

Specify the size of the buffer to which the system stores UDP packets. The default size is 8192 bytes. Make the buffer large enough to store UDP packets that come in while your process reads a packet from the buffer or performs other tasks. Specifying the buffer size prevents the receive buffer from overflowing.

---

**Note** This parameter appears only in a deprecated version of the UDP Receive block. Replace the deprecated UDP Receive block with a current UDP Receive block.

---

## See Also

Byte Pack, Byte Reversal, Byte Unpack, UDP Send

**Introduced in R2011a**

# UDP Send

Send UDP message

## Block Library

Embedded Coder/Embedded Targets/Host Communication

## Description

The UDP Send block transmits an input vector as a UDP message over an IP network port. The remote IP port number to which the block sends the UDP packets is tunable in the C/C++ generated code.

---

**Note** Some Simulink blocks and `.exe` files built from models that contain those blocks require shared libraries, such as `.dll` files on Windows. The UDP Send block requires the `networkdevice.dll` library file. To meet this requirement, follow the example on the `packNGo` function page to package the code files for your model. The resulting compressed folder contains the `.dll` files that the model requires, including `networkdevice.dll`. To run this type of `.exe` file outside a MATLAB environment, place the required `.dll` files in the same folder as the `.exe` file, or place them in a folder on the Windows system path. For more details, see “How To Run a Generated Executable Outside MATLAB” (DSP System Toolbox).

---

## Parameters

**IP address ('255.255.255.255' for broadcast)**

Specify the IP address or hostname to which the block sends the message. To broadcast the UDP message, retain the default value, `'255.255.255.255'`.

**Remote IP port**

Specify the port to which the block sends the message. This parameter is tunable in the C/C++ generated code but not tunable during simulation. The default is `25000`. The value can be in the range [1 65535].

---

**Note** On Linux, to set the IP port number below 1024, run MATLAB with root privileges. For example, at the Linux command line, enter:

```
sudo matlab
```

---

**Local IP port source**

To let the system automatically assign the port number, select `Assign automatically`. To specify the IP port number using the **Local IP port** parameter, select `Specify`.

**Local IP port**

Specify the IP port number from which the block sends the message.

If the receiving address expects messages from a particular port number, enter that number here.

## Deprecated Parameters

**Sample time**

Sample time tells the block how long to wait before polling for new messages.

---

**Note** This parameter only appears in a deprecated version of the UDP Send block. Replace the deprecated UDP Send block with a current UDP Send block.

---

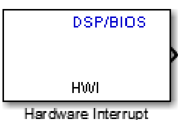
## See Also

Byte Pack, Byte Reversal, Byte Unpack, UDP Receive

**Introduced in R2011a**

# DSP/BIOS Hardware Interrupt

Generate Interrupt Service Routine




## Library

Embedded Coder Support Package for Texas Instruments C6000 Processors/ DSP/BIOS

## Description

Creates an Interrupt Service Routine (ISR) that executes the task block or subsystem that is downstream from the block. ISRs are functions that the CPU executes in response to an external event.

Interrupt numbers for C6000 family processors range from 0 to 15, with 0 reserved for the reset ISR. The following table presents the set of interrupt numbers for the C6713 processor. For more detailed and specific information about interrupts, refer to Texas Instruments technical documentation for your target processor.

| Interrupt Number | Default Event | Module |
|---|---|---|
| 0 | Reset | |
| 1 | NMI | |
| 2 | Reserved | |
| 3 | Reserved | |
| 4 | GPINT4 | GPIO |
| 5 | GPINT5 | GPIO |
| 6 | GPINT6 | GPIO |
| 7 | GPINT7 | GPIO |

| Interrupt Number | Default Event | Module |
| --- | --- | --- |
| 8 | EDMAINT | EDMA |
| 9 | EMUDTDMA | Emulation |
| 10 | SDINT | EMIF |
| 11 | EMURTDXRX | Emulation |
| 12 | EMURTDXTX | Emulation |
| 13 | DSPINT | HPI |
| 14 | TINT0 | Timer 0 |
| 15 | TINT1 | Timer 1 |

In models, you usually follow this block with either a DSP/BIOS Task or DSP/BIOS Triggered Task block.

## Parameters

**Interrupt number(s)**

Enter one or more integer values as a vector that represent interrupts. Interrupts have a value from `0`, the highest priority to `15`, lowest priority. As shown, enter the values enclosed in square brackets. For example, entering

```
[3 5 15]
```

results in three interrupt routines. `[5 8]` is the default entry, specifying two interrupts.

**Preemption flag(s)**

Higher priority interrupts can preempt interrupts that have lower priority. To allow you to control preemption, use the preemption flags to specify whether an interrupt can be preempted.

Entering `1` indicates that the interrupt can be preempted. Entering `0` indicates the interrupt cannot be preempted. When **Interrupt numbers** contains more than one interrupt priority, you can assign different preemption flags to each interrupt by entering a vector of flag values, corresponding to the order of the interrupts in **Interrupt numbers**. If **Interrupt numbers** contains more than one interrupt, and you enter only one flag value here, that status applies to all interrupts.

In the default settings [0 1], the interrupt with priority 5 in **Interrupt numbers** is not preemptible and the priority 8 interrupt can be preempted.

**Manage own timer**

The ISR generated by this block can manage its own time by reading time from the clock on the board. Selecting this option directs the ISR to maintain the time itself. When you select **Manage own timer**, you enable the **Timer resolution** option that reports the timer resolution the ISR uses.

**Timer resolution (seconds)**

When you direct the block to manage its own time, this option (available only when you select **Manage own timer**) reports the resolution of the clock. **Timer resolution** is a read-only parameter. You cannot change the value.

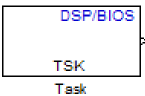
**Enable simulation input**

Selecting this option adds an input port to the block for simulating inputs in Simulink software. Connect interrupt simulation sources to the input. This option affects simulation only. It does not alter generated code.

**Introduced in R2011a**

# DSP/BIOS Task

Create task that runs as separate DSP/BIOS thread




## Library

Embedded Coder Support Package for Texas Instruments C6000 Processors/ DSP/BIOS

## Description

Creates a free-running task that runs in response to an ISR and as a separate DSP/BIOS™ thread. The spawned task runs the downstream function call subsystem in the model.

When the process runs this task, it uses a semaphore structure to enable the task and restrict access by it to other resources.

In order to use this block, set the **System target file** parameter to `idelink_ert.tlc` or `idelink_ert.tlc`. The **System target file** parameter is located on the Code Generation pane of the Model Configuration Parameters dialog, which you can view by selecting your model and pressing **Ctrl+E**.

## Parameters

**Task name (32 characters or less)**

Creates a name for the task. Enter up to 32 characters, including numbers and letters. You cannot use the standard C reserved characters, such as / and : in the name.

**Task priority (1-15)**

Sets the priority for the task, where 1 is the lowest priority and 15 the highest. Higher priority tasks can preempt tasks that have lower priority.

**Stack size (bytes)**

Specify the size of the stack the task uses. The value defaults to 4096 bytes. Each DSP/BIOS task has a separate stack. This parameter is not related to **System stack size (MAUs)** in the model Configuration Parameters.

**Stack memory segment**

Specify where the stack resides in memory.

**Manage own timer**

This block can manage its own time by reading time from the clock on the board. Selecting this option directs the task/block to maintain the time itself. When you select **Manage own timer**, you enable the **Timer resolution** option that reports the timer resolution the task uses.

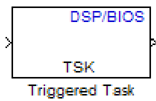
**Timer resolution (seconds)**

When you direct the block to manage its own time, this option (available only when you select **Manage own timer**) reports the resolution of the clock. **Timer resolution** is a read-only parameter. You cannot change the value.

### Introduced in R2011a

# DSP/BIOS Triggered Task

Create asynchronously triggered task




## Library

Embedded Coder Support Package for Texas Instruments C6000 Processors/ DSP/BIOS

## Description

Creates a task that runs asynchronously in response to an ISR and as a separate DSP/BIOS thread. The spawned task runs the downstream function call subsystem in the model.

When the process runs this task, it uses a semaphore structure to enable the task and restrict access by it to other resources.

## Parameters

**Task name (32 characters or less)**

Creates a name for the task. Enter up to 32 characters, including numbers and letters. You cannot use the standard C reserved characters, such as / or : in the name.

**Task priority (1-15)**

Sets the priority for the task, where 1 is the lowest priority and 15 the highest. Higher priority tasks can preempt tasks that have lower priority, unless the preemptible flag (**Preemption flag** option on the DSP/BIOS Hardware Interruptblock) prevents preempting the task.

**Stack size (bytes)**

Specify the size of the stack the task uses. The value defaults to 4096 bytes. Take care to set this value to a value that is large enough. If the task uses more than the allotted space it can write into other memory areas with unintended results.

Each DSP/BIOS task has a separate stack. This parameter is not related to **System stack size (MAUs)** in the model Configuration Parameters.

**Stack memory segment**

Specify where the stack resides in memory by specifying the memory segment. Additional information about DSP/BIOS memory segments also appears in the Target Hardware Resources tab.

**Synchronize data transfer of this task with caller task**

Specify whether this task should synchronize data transfer with the calling task. Select this option to enable synchronization. Clearing this option enables the **Timer resolution** option.

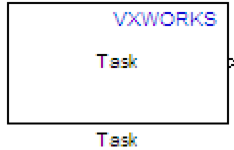
**Timer resolution**

When you direct the block not to synchronize data with the calling task (by clearing **Synchronize data transfer of this task with caller task**), **Timer resolution** reports the resolution of the timer. **Timer resolution** is a read-only parameter. You cannot change the value.

**Introduced in R2011a**

# VxWorks Task

Spawn task function as separate VxWorks thread

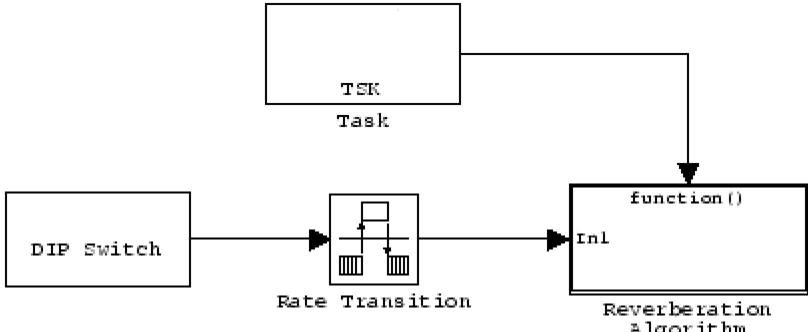



## Library

Embedded Coder Support Package for Wind River® VxWorks® RTOS

## Description

Use this block to create a task function that spawns as a separate VxWorks thread. The task function runs the code of the downstream function-call subsystem. For example:




In order to use this block, set the **System target file** parameter to `idelink_ert.tlc` or `idelink_grt.tlc`. The **System target file** parameter is located on the Code Generation pane of the Model Configuration Parameters dialog, which you can view by selecting your model and pressing **Ctrl+E**.

The VxWorks Task block uses a First In, First Out (FIFO) scheduling algorithm, which executes real-time processes without time slicing. With FIFO scheduling, a higher-priority

process preempts a lower-priority process. While the higher-priority process runs, the lower-priority process remains at the top of the list for its priority. When the scheduler blocks the higher-priority processes, the lower-priority process resumes.

For example, in the following image, task2 preempts task1. Then, task3 preempts task2. When task3 completes, task2 resumes. When task2 completes, task1 resumes.

FIFO Scheduling

High Priority

task3

task2

task2

Low Priority

task1

task1

Time

Key: Preemption Completion

# Parameters

**Task function name (32 characters or less)**

Assign a name to this task. You can enter up to 32 letters and numbers. Do not use standard C reserved characters, such as the / and : characters.

**Thread priority (0-255)**

Set the priority for the thread, from 0 to 255 (low-to-high). Higher-priority tasks can preempt lower-priority tasks.

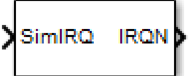
**Introduced in R2011a**

# Blocks in Simulink Coder—Alphabetical List

# 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` (Simulink Coder) 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 in the **VME interrupt number(s)** field. 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 **VME interrupt number(s)**.

The **Simulink task priority** values are required to generate a rate transition code (see “Rate Transitions and Asynchronous Blocks” (Simulink Coder)). 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 in the **VME interrupt number(s)** field.

---

**Note** The number of elements in the arrays specifying **VME interrupt vector offset(s)** and **Simulink task priority** must match the number of elements in the **VME interrupt number(s)** array.

---

### `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 the **Timer size** option.

### `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. The **Timer resolution** field 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 in the **Timer resolution** field.

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” (Simulink Coder) and “Async Interrupt Block Implementation” (Simulink Coder).

#### `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 **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 **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, the code generator uses a second 32-bit integer to address overflows.

For more information, see “Control Memory Allocation for Time Counters” (Simulink Coder). See also “Timers in Asynchronous Tasks” (Simulink Coder).

#### `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” (Simulink Coder). 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” (Simulink Coder)
“Asynchronous Events” (Simulink Coder)

**Introduced in R2006a**

# Asynchronous Task Specification

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

**Library:** Simulink Coder / Asynchronous

P: 10

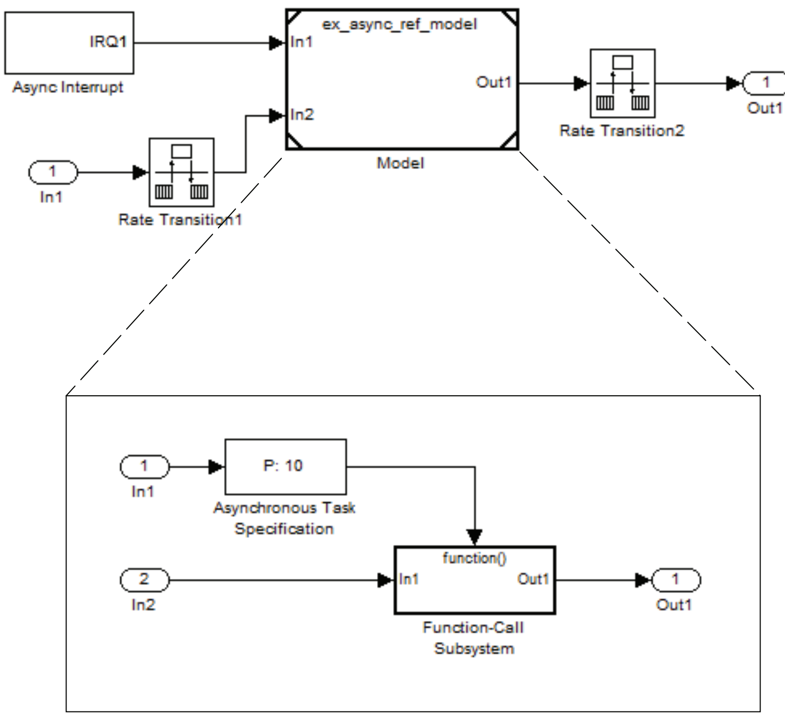
## Description

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” (Simulink Coder).

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

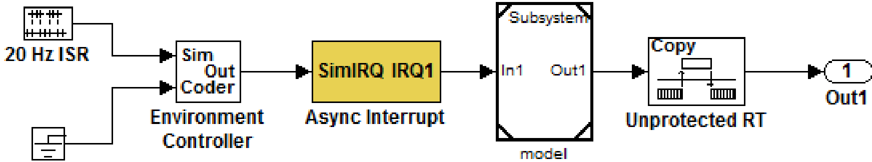
### `Task priority` — Priority of asynchronous task that calls function-call subsystem

`10` (default)

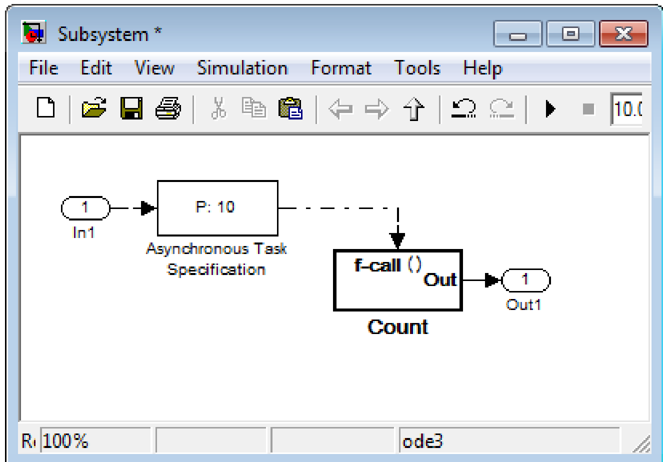
Specify an integer or [] as the priority of the asynchronous task that calls the connected function-call subsystem. The priority must be a value that generates relevant rate transition behaviors.

- 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

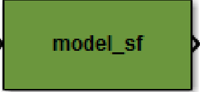
## Topics

"Asynchronous Events" (Simulink Coder)
"Spawn and Synchronize Execution of RTOS Task" (Simulink Coder)
"Pass Asynchronous Events in RTOS as Input To a Referenced Model" (Simulink Coder)
"Convert an Asynchronous Subsystem into a Model Reference" (Simulink Coder)
"Rate Transitions and Asynchronous Blocks" (Simulink Coder)
"Asynchronous Support" (Simulink Coder)
"Asynchronous Events" (Simulink Coder)
"Model Referencing" (Simulink)

**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” (Simulink Coder).

### 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” (Simulink Coder).
- 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” (Simulink Coder)).

# 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

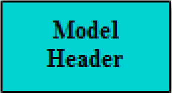
## Topics

"Generate S-Function from Subsystem" (Simulink Coder)
"Create S-Function Blocks from a Subsystem" (Simulink Coder)

**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.

---

**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

### `Top of Model Header` — Code to add near top of generated header file

Specify code that you want the code generator to add near the top of the header file for the model. The code generator places the code in the section labeled `user code (top of header file)`.

### `Bottom of Model Header` — Code to add at bottom of generated header file

Specify code that you want the code generator to add at the bottom of the header file for the model. The code generator places the code in the section labeled `user code (bottom of header file)`.

## See Also

Model Source | System Disable | System Outputs | System Update | System Derivatives | System Enable | System Initialize | System Start | System Terminate

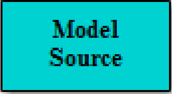
## Topics

"Place External C/C++ Code in Generated Code"

**Introduced in R2006a**

# Model Source

Specify external source code




## Description

For a model that includes the Model Source block, the code generator adds external code that you specify to the source file (`model.c` or `model.cpp`) that it generates. You can specify code for the code generator to add near the top and bottom of the source file.

---

**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

### `Top of Model Header` — 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 Header` — 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" (Simulink Coder).

## See Also

Model Header | System Disable | System Outputs | System Update | System Derivatives | System Enable | System Initialize | System Start | System Terminate

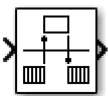
### Topics

"Place External C/C++ Code in Generated Code"

**Introduced in R2006a**

# Protected RT

Handle transfer of data between blocks operating at different rates and maintain data integrity



## Library

VxWorks (`vxlib1`)

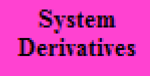
## Description

The Protected RT block is a Rate Transition block that is preconfigured to maintain data integrity during data transfers. For more information, see Rate Transition in the Simulink Reference.

**Introduced in R2006a**

# System Derivatives

Specify external system derivative code




## Description

For a model or nonvirtual subsystem that includes the System Derivatives block and a block that computes continuous states, the code generator adds external code, which you specify, to the `SystemDerivatives` function that it generates. You can specify code for the code generator to add to the declaration, execution, and exit sections of the function code.

---

**Note** If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

---

## Parameters

### `System Derivatives Function Declaration Code` — Code to add to the declaration section of the generated function

Specify code that you want the code generator to add to the declaration section of the `SystemDerivatives` function for the model or subsystem.

### `System Derivatives Function Execution Code` — Code to add to the execution section of the generated function

Specify code that you want the code generator to add to the execution section of the `SystemDerivatives` function for the model or subsystem.

### `System Derivatives Function Exit Code` — Code to add to the exit section of the generated function

Specify code that you want the code generator to add to the exit section of the `SystemDerivatives` function for the model or subsystem.

# See Also

Model Header | Model Source | System Initialize | System Disable | System Enable | System Outputs | System Start | System Terminate | System Update

## Topics

"Place External C/C++ Code in Generated Code"

**Introduced in R2006a**

# System Disable

Specify external system disable code

## Description

For a model or nonvirtual subsystem that includes the System Disable block and a block that uses a `SystemDisable` function, the code generator adds external code, which you specify, to the `SystemDisable` function that it generates. You can specify code for the code generator to add to the declaration, execution, and exit sections of the function code.

---

**Note** If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

---

## Parameters

**`System Disable Function Declaration Code` — Code to add to the declaration section of the generated function**

Specify code that you want the code generator to add to the declaration section of the `SystemDisable` function for the model or subsystem.

**`System Disable Function Execution Code` — Code to add to the execution section of the generated function**

Specify code that you want the code generator to add to the execution section of the `SystemDisable` function for the model or subsystem.

**`System Disable Function Exit Code` — Code to add to the exit section of the generated function**

Specify code that you want the code generator to add to the exit section of the `SystemDisable` function for the model or subsystem.

## See Also

Model Header | Model Source | System Initialize | System Derivatives | System Enable | System Outputs | System Start | System Terminate | System Update

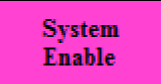
### Topics

"Place External C/C++ Code in Generated Code"

**Introduced in R2006a**

# System Enable

Specify external system enable code




## Description

For a model or nonvirtual subsystem that includes the System Enable block and a block that uses a `SystemEnable` function, the code generator adds external code, which you specify, to the `SystemEnable` function that it generates. You can specify code for the code generator to add to the declaration, execution, and exit sections of the function code.

---

**Note** If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

---

## Parameters

### `System Enable Function Declaration Code` — Code to add to the declaration section of the generated function

Specify code that you want the code generator to add to the declaration section of the `SystemEnable` function for the model or subsystem.

### `System Enable Function Execution Code` — Code to add to the execution section of the generated function

Specify code that you want the code generator to add to the execution section of the `SystemEnable` function for the model or subsystem.

### `System Enable Function Exit Code` — Code to add to the exit section of the generated function

Specify code that you want the code generator to add to the exit section of the `SystemEnable` function for the model or subsystem.

## See Also

Model Header | Model Source | System Initialize | System Derivatives | System Disable | System Outputs | System Start | System Terminate | System Update

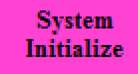
### Topics

"Place External C/C++ Code in Generated Code"

**Introduced in R2006a**

# System Initialize

Specify external system initialization code




## Description

For a model or nonvirtual subsystem that includes the System Initialize block and a block that uses a `SystemInitialize` function, the code generator adds external code, which you specify, to the `SystemInitialize` function that it generates. You can specify code for the code generator to add to the declaration, execution, and exit sections of the function code.

---

**Note** If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

---

## Parameters

**`System Initialize Function Declaration Code` — Code to add to the declaration section of the generated function**

Specify code that you want the code generator to add to the declaration section of the `SystemInitialize` function for the model or subsystem.

**`System Initialize Function Execution Code` — Code to add to the execution section of the generated function**

Specify code that you want the code generator to add to the execution section of the `SystemInitialize` function for the model or subsystem.

**`System Initialize Function Exit Code` — Code to add to the exit section of the generated function**

Specify code that you want the code generator to add to the exit section of the `SystemInitialize` function for the model or subsystem.

## See Also

Model Header | Model Source | System Enable | System Derivatives | System Disable | System Outputs | System Start | System Terminate | System Update

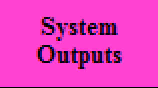
### Topics

"Place External C/C++ Code in Generated Code"

**Introduced in R2006a**

# System Outputs

Specify external system outputs code




## Description

For a model or nonvirtual subsystem that includes the System Outputs block and a block that uses a `SystemOutputs` function, the code generator adds external code, which you specify, to the `SystemOutputs` function that it generates. You can specify code for the code generator to add to the declaration, execution, and exit sections of the function code.

---

**Note** If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

---

## Parameters

### `System Outputs Function Declaration Code` — Code to add to the declaration section of the generated function

Specify code that you want the code generator to add to the declaration section of the `SystemOutputs` function for the model or subsystem.

### `System Outputs Function Execution Code` — Code to add to the execution section of the generated function

Specify code that you want the code generator to add to the execution section of the `SystemOutputs` function for the model or subsystem.

### `System Outputs Function Exit Code` — Code to add to the exit section of the generated function

Specify code that you want the code generator to add to the exit section of the `SystemOutputs` function for the model or subsystem.

## See Also

Model Header | Model Source | System Enable | System Derivatives | System Disable | System Initialize | System Start | System Terminate | System Update

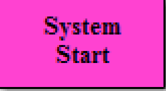
### Topics

"Place External C/C++ Code in Generated Code"

**Introduced in R2006a**

# System Start

Specify external system startup code




## Description

For a model or nonvirtual subsystem that includes the System Start block and a block that uses a `SystemStart` function, the code generator adds external code, which you specify, to the `SystemStart` function that it generates. You can specify code for the code generator to add to the declaration, execution, and exit sections of the function code.

---

**Note** If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

---

## Parameters

### `System Start Function Declaration Code` — Code to add to the declaration section of the generated function

Specify code that you want the code generator to add to the declaration section of the `SystemStart` function for the model or subsystem.

### `System Start Function Execution Code` — Code to add to the execution section of the generated function

Specify code that you want the code generator to add to the execution section of the `SystemStart` function for the model or subsystem.

### `System Start Function Exit Code` — Code to add to the exit section of the generated function

Specify code that you want the code generator to add to the exit section of the `SystemStart` function for the model or subsystem.

## See Also

Model Header | Model Source | System Enable | System Terminate | System Derivatives | System Disable | System Initialize | System Outputs | System Update

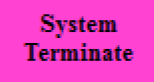
### Topics

"Place External C/C++ Code in Generated Code"

**Introduced in R2006a**

# System Terminate

Specify external system termination code




## Description

For a model or nonvirtual subsystem that includes the System Terminate block and a block that uses a `SystemTerminate` function, the code generator adds external code, which you specify, to the `SystemTerminate` function that it generates. You can specify code for the code generator to add to the declaration, execution, and exit sections of the function code.

---

**Note** If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

---

## Parameters

#### `System Terminate Function Declaration Code` — Code to add to the declaration section of the generated function

Specify code that you want the code generator to add to the declaration section of the `SystemTerminate` function for the model or subsystem.

#### `System Disable Terminate Execution Code` — Code to add to the execution section of the generated function

Specify code that you want the code generator to add to the execution section of the `SystemTerminate` function for the model or subsystem.

#### `System Disable Terminate Exit Code` — Code to add to the exit section of the generated function

Specify code that you want the code generator to add to the exit section of the `SystemTerminate` function for the model or subsystem.

## See Also

Model Header | Model Source | System Enable | System Start | System Derivatives | System Disable | System Initialize | System Outputs | System Update

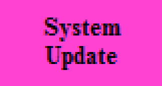
### Topics

"Place External C/C++ Code in Generated Code"

**Introduced in R2006a**

# System Update

Specify external system update code




## Description

For a model or nonvirtual subsystem that includes the System Update block and a block that uses a `SystemUpdate` function, the code generator adds external code, which you specify, to the `SystemUpdate` function that it generates. You can specify code for the code generator to add to the declaration, execution, and exit sections of the function code.

**Note** If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

## Parameters

### `System Update Function Declaration Code` — Code to add to the declaration section of the generated function

Specify code that you want the code generator to add to the declaration section of the `SystemUpdate` function for the model or subsystem.

### `System Update Function Execution Code` — Code to add to the execution section of the generated function

Specify code that you want the code generator to add to the execution section of the `SystemUpdate` function for the model or subsystem.

### `System Update Function Exit Code` — Code to add to the exit section of the generated function

Specify code that you want the code generator to add to the exit section of the `SystemUpdate` function for the model or subsystem.

# See Also

Model Header | Model Source | System Enable | System Start | System Derivatives | System Disable | System Initialize | System Outputs | System Terminate

## Topics

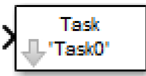
"Place External C/C++ Code in Generated Code"

**Introduced in R2006a**

# Task Sync

Run code of downstream function-call subsystem or Stateflow chart by spawning an example RTOS (VxWorks) task

**Library:** Simulink Coder / Asynchronous / Interrupt Templates




## Description

The Task Sync block spawns an example RTOS (VxWorks) task that calls a function-call subsystem or Stateflow chart. Typically, you place the Task Sync block between an Async Interrupt block and a function-call subsystem block or Stateflow chart. Alternatively, you could connect the Task Sync block to the output port of a Stateflow diagram that has an event, `Output to Simulink`, configured as a function call.

The Task Sync block:

- Uses the RTOS (VxWorks) system call `taskSpawn` to spawn an independent task. When the task is activated, it calls the downstream function-call subsystem code or Stateflow chart. The block calls `taskDelete` to delete the task during model termination.
- Creates a semaphore to synchronize the connected subsystem with execution of the block.
- Wraps the spawned task in an infinite `for` loop. In the loop, the spawned task listens for the semaphore by using `semTake`. The first call to `semTake` specifies `NO_WAIT`. This setting lets the task determine whether a second `semGive` has occurred before the completion of the function-call subsystem or chart. This sequence indicates that the interrupt rate is too fast or the task priority is too low.
- Generates synchronization code (for example, `semGive()`). This code lets the spawned task run. The task in turn calls the connected function-call subsystem code. The synchronization code can run at interrupt level. The connection between the Async Interrupt and Task Sync blocks accomplishes this operation and triggers execution of the Task Sync block within an ISR.
- Supplies absolute time if blocks in the downstream algorithmic code require it. The time comes from the timer maintained by the Async Interrupt block or comes from an independent timer maintained by the task associated with the Task Sync block.

When you design your application, consider when timer and signal input values could be taken for the downstream function-call subsystem that is connected to the Task Sync block. By default, the time and input data are read when the RTOS (VxWorks) activates the task. For this case, the data (input and time) are synchronized to the task itself. If you select the **Synchronize the data transfer of this task with the caller task** option and the Task Sync block driver is an Async Interrupt block, the time and input data are read when the interrupt occurs (that is, within the ISR). For this case, data is synchronized with the caller of the Task Sync block.

**Note** You can use the blocks in the `vxlib1` (Simulink Coder) 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”

(Simulink Coder)). 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” (Simulink Coder). See also “Timers in Asynchronous Tasks” (Simulink Coder).

## See Also

Async Interrupt

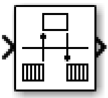
### Topics

“Asynchronous Events” (Simulink Coder)

**Introduced in R2006a**

# Unprotected RT

Handle transfer of data between blocks operating at different rates and maintain determinism



## Library

VxWorks (`vxlib1`)

## Description

The Unprotected RT block is a Rate Transition block that is preconfigured to conduct deterministic data transfers. For more information, see Rate Transition in the Simulink Reference.

**Introduced in R2006a**

# Embedded Coder Parameters: Advanced Parameters

# Create block

## Description

Generate a SIL or PIL block

**Category:** Code Generation > Verification

## Settings

**Default:** `None`

`None`

SIL or PIL block not generated.

`SIL`

Generate a SIL block that represents a top-model or subsystem.

If you select this option, the software creates and opens an untitled model with a SIL block. The SIL block contains an S-function, through which the software runs compiled object code on the host computer. With this block, you can verify the behavior of source code generated from top-model or subsystem components.

If the subsystem is an export-function subsystem, the software creates a Model block with **Simulation mode** set to `Software-in-the-loop (SIL)`.

`PIL`

Generate a PIL block that represents a top-model or subsystem.

If you select this option, the software creates and opens an untitled model with a PIL block. The PIL block contains an S-function, through which the software runs cross-compiled object code on a target processor or instruction set simulator. With this block, you can verify the behavior of object code generated from top-model or subsystem components.

If the subsystem is an export-function subsystem, the software creates a Model block with **Simulation mode** set to `Processor-in-the-loop (PIL)`.

To control the way code compiles and executes in the target environment, use Target Connectivity API.

### Command-Line Information

**Parameter:** `CreateSILPILBlock`
**Type:** character vector
**Value:** `'None'` | `'SIL'` | `'PIL'`
**Default:** `'None'`

### Recommended Settings

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

## See Also

### Related Examples

- “Simulation with Blocks From Subsystems”
- “Generate Code for Export-Function Subsystems”
- “SIL and PIL Simulations”
- “Generate Code for Export-Function Subsystems”

# Existing shared code

## Description

Specify folder that contains existing shared code

**Category:** Code Generation Advanced Parameters

## Settings

**Default:** none

Path to folder that contains existing shared code. Specify the absolute path or a path relative to the Simulink preference **Code generation folder** (`CodeGenFolder`). The model build process uses the code in this folder instead of locally generated shared utility code.

## Command-Line Information

**Parameter:** `ExistingSharedCode`
**Type:** character vector
**Value:** valid MATLAB variable name
**Default:** none

## Recommended Settings

| Application | Setting |
|---|---|
| Debugging | No impact |
| Traceability | Valid MATLAB variable name |
| Efficiency | No impact |
| Safety precaution | No impact |

# See Also

`sharedCodeUpdate`

## Related Examples

- “Cross-Release Shared Utility Code Reuse”
- “Cross-Release Code Integration”

# Use only existing shared code

## Description

Check whether build process requires shared code that is not present in the existing shared code folder.

**Category:** Diagnostics

## Settings

**Default:** `none`

`none`

Simulink software takes no action.

`warning`

Simulink software displays a warning.

`error`

Simulink software terminates the simulation and displays an error message.

## Dependency

You can specify this parameter only if you specify a folder in the **Existing shared code** field. Otherwise the field appears dimmed.

## Command-Line Information

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

## Recommended Settings

| Application | Setting |
| --- | --- |
| Debugging | No impact |

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

# See Also

## Related Examples

- “Cross-Release Shared Utility Code Reuse”
- “Cross-Release Code Integration”

# Use Embedded Coder Features

## Description

Enable “Embedded Coder” features for models deployed to “Simulink Supported Hardware” (Simulink).

---

**Note** If you enable this parameter in a model where Embedded Coder is not installed or available in the environment, a question dialog box prompts you to update the model to build without Embedded Coder features.

---

**Category:** Hardware Implementation

## Settings

**Default:** `On`

On

Enable advanced Embedded Coder configuration parameters.

---

**Note** Enabling `Use Embedded Coder Features` also enables the “Use Simulink Coder Features” (Simulink Coder) parameter.

---

Off

Disable advanced Embedded Coder configuration parameters.

## Dependencies

This parameter requires an Embedded Coder license.

## Command-Line Information

**Parameter:** `UseEmbeddedCoderFeatures`
**Value:** `'on'` or `'off'`

**Default:** `'on'`

# See Also

## Related Examples

- "Hardware Implementation Pane" (Simulink)

# Remove reset function

## Description

Remove unreachable (dead-code) instances of the `reset` functions from the generated code for ERT-based systems that include model referencing hierarchies. If you enable this parameter, Simulink checks that live code will be removed and errors if it finds such code.

**Category:** Code Generation > Interface

## Settings

**Default:** `On`

On

Remove unreachable instances of the `reset` functions from the generated code for ERT-based systems that include model referencing hierarchies.

Off

Generate code without removing unreachable instances of the `reset` function.

## Dependencies

This parameter requires an Embedded Coder license.

To set the **Remove reset function** parameter, set **Configuration Parameters** > **Code Generation** > **System target file** parameter to an ERT-based system target file, such as `ert.tlc`.

## Command-Line Information

**Parameter:** `RemoveResetFunc`
**Value:** `'on'` or `'off'`
**Default:** `'on'`

# See Also

## Related Examples

- “Remove disable function” on page 5-12
- “Model Configuration Parameters: Code Generation Interface” (Simulink Coder)

# Remove disable function

## Description

Remove unreachable (dead-code) instances of the `disable` functions from the generated code for ERT-based systems that include model referencing hierarchies. If you enable this parameter, Simulink checks that live code will be removed and errors if it finds such code.

**Category:** Code Generation > Interface

## Settings

**Default:** `Off`

On

Remove unreachable instances of the `disable` functions from the generated code for ERT-based systems that include model referencing hierarchies.

Off

Generate code without removing unreachable instances of the `disable` function.

## Dependencies

This parameter requires an Embedded Coder license.

To set the **Remove disable function** parameter, set **Configuration Parameters** > **Code Generation** > **System target file** parameter to an ERT-based system target file, such as `ert.tlc`.

## Command-Line Information

**Parameter:** `RemoveDisableFunc`
**Value:** `'on'` or `'off'`
**Default:** `'off'`

# See Also

## Related Examples

- “Remove reset function” on page 5-10
- “Model Configuration Parameters: Code Generation Interface” (Simulink Coder)

# Code Generation Parameters: AUTOSAR

# Model Configuration Parameters: Code Generation AUTOSAR

The **Code Generation > AUTOSAR Code Generation Options** category includes parameters for controlling AUTOSAR code generation. On the Configuration Parameters dialog box, the following configuration parameters are on the **Commonly Used** tab on the **Code Generation > AUTOSAR Code Generation Options** pane.

| Parameter | Description |
|---|---|
| "Generate XML file for schema version" on page 6-4 | Select the AUTOSAR schema version to use when generating XML files. |
| "Maximum SHORT-NAME length" on page 6-6 | Specify maximum length for `SHORT-NAME` XML elements. |
| "Use AUTOSAR compiler abstraction macros" on page 6-7 | Specify use of AUTOSAR macros to abstract compiler directives. |
| "Support root-level matrix I/O using one-dimensional arrays" on page 6-9 | Allow root-level matrix I/O. |

## See Also

### More About

- "AUTOSAR Code Generation"
- "Model Configuration"

# Code Generation: AUTOSAR Code Generation Options Tab Overview

Parameters for controlling AUTOSAR code generation options.

### Configuration

This pane appears only if you specify the `autosar.tlc` system target file.

### To get help on an option

1. Right-click the option text label.
2. From the context menu, select **What's This**.

### Tip

From the Simulink **Code** menu, select **C/C++ Code** > **Configure Model as AUTOSAR Component** to open a dialog box where you can configure other AUTOSAR options.

## See Also

### Related Examples

- “Model Configuration Parameters: Code Generation AUTOSAR” on page 6-2
- “AUTOSAR Code Generation”

# Generate XML file for schema version

## Description

Select the AUTOSAR schema version to use when generating XML files.

**Category:** Code Generation > AUTOSAR Code Generation Options

## Settings

**Default:** `4.3`

`4.3`
: Use schema version 4.3 (revision 4.3.0)

`4.2`
: Use schema version 4.2 (revision 4.2.2)

`4.1`
: Use schema version 4.1 (revision 4.1.3)

`4.0`
: Use schema version 4.0 (revision 4.0.3)

`3.2`
: Use schema version 3.2 (revision 3.2.2)

`3.1`
: Use schema version 3.1 (revision 3.1.4)

`3.0`
: Use schema version 3.0 (revision 3.0.2)

`2.1`
: Use schema version 2.1 (XSD rev 0017)

## Tip

- Selecting the AUTOSAR target for your model for the first time sets the schema version parameter to the default value, `4.3`.

- When you import `arxml` code into Simulink, the `arxml` importer detects the schema version and sets the schema version parameter in the model. For a list of AUTOSAR schema revisions supported for `arxml` import, see “Select an AUTOSAR Schema”.
- Must be set to the same value for top and referenced models.
- To configure other AUTOSAR XML options, from the Simulink **Code** menu, select **C/C++ Code** > **Configure AUTOSAR Dictionary**. Select **XML Options**.

### Command-Line Information

**Parameter:** `AutosarSchemaVersion`
**Type:** character vector
**Value:** `'4.3'` | `'4.2'` | `'4.1'` | `'4.0'` | `'3.2'` | `'3.1'` | `'3.0'` | `'2.1'`
**Default:** `'4.3'`

## See Also

### Related Examples

- “Model Configuration Parameters: Code Generation AUTOSAR” on page 6-2
- “AUTOSAR Code Generation”

# Maximum SHORT-NAME length

## Description

Specify maximum length for `SHORT-NAME` XML elements

**Category:** Code Generation > AUTOSAR Code Generation Options

## Settings

**Default:** 128

The AUTOSAR standard specifies that the length of `SHORT-NAME` XML elements cannot be greater than 128 characters, for schema version 4.x or later, or 32 characters, for earlier schema versions. Use this parameter to specify a maximum length for SHORT-NAME elements exported by the code generator, up to 128 characters.

## Tip

Must be set to the same value for top and referenced models.

## Command-Line Information

**Parameter:** `AutosarMaxShortNameLength`
**Type:** integer
**Value:** an integer less or equal to 128
**Default:** 128

# See Also

## Related Examples

- "Model Configuration Parameters: Code Generation AUTOSAR" on page 6-2
- "Specify Maximum SHORT-NAME Length"

# Use AUTOSAR compiler abstraction macros

## Description

Specify use of AUTOSAR macros to abstract compiler directives

**Category:** Code Generation > AUTOSAR Code Generation Options

## Settings

**Default:** Off

On

Software generates code with C macros that are abstracted compiler directives (near/far memory calls)

Off

Software generates code that does *not* contain AUTOSAR compiler abstraction macros.

## Tip

Must be the same for top and referenced models.

## Command-Line Information

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

# See Also

## Related Examples

- “Model Configuration Parameters: Code Generation AUTOSAR” on page 6-2

- “Configure AUTOSAR Compiler Abstraction Macros”

# Support root-level matrix I/O using one-dimensional arrays

## Description

Allow root-level matrix I/O

**Category:** Code Generation > AUTOSAR Code Generation Options

## Settings

**Default:** Off

On

Software supports matrix I/O at the root-level by generating code that implements matrices as one-dimensional arrays.

Off

Software does not allow matrix I/O at the root-level. If you try to build a model that has matrix I/O at the root-level, the software produces an error.

## Tip

Must be the same for top and referenced models.

## Command-Line Information

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

# See Also

## Related Examples

- “Model Configuration Parameters: Code Generation AUTOSAR” on page 6-2
- “Root-Level Matrix I/O”

# Code Generation Parameters: Code Placement

# Model Configuration Parameters: Code Generation Code Placement

The **Code Generation > Code Placement** category includes parameters for configuring the appearance of the generated code. On the Configuration Parameters dialog box, the following configuration parameters are on the **Commonly Used** tab on the **Code Generation > Code Placement** pane.

| Parameter | Description |
|---|---|
| "Data definition" on page 7-5 | Specify where to place definitions of global variables. |
| "Data definition filename" on page 7-7 | Specify the name of the file that is to contain data definitions. |
| "Data declaration" on page 7-9 | Specify where `extern`, `typedef`, and `#define` statements are to be declared. |
| "Data declaration filename" on page 7-11 | Specify the name of the file that is to contain data declarations. |
| "#include file delimiter" on page 7-15 | Specify the type of `#include` file delimiter to use in generated code. |
| "Use owner from data object for data definition placement" on page 7-13 | Specify whether the model uses or ignores the ownership setting of a data object for data definition in code generation. |
| "Signal display level" on page 7-17 | Specify the persistence level for MPT signal data objects. |
| "Parameter tune level" on page 7-19 | Specify the persistence level for MPT parameter data objects. |
| "File packaging format" on page 7-21 | Specify whether code generation modularizes the code components into many files or compacts the generated code into a few files. |
| "Header files" on page 7-24 | Specify customized name for generated header files. |
| "Source files" on page 7-26 | Specify customized name for generated source files. |

| Parameter | Description |
| --- | --- |
| “Data files” on page 7-28 | Specify customized name for generated data files. |
| “Rate Transition block code” on page 7-30 | Specify the format for Rate Transition block code and data. |

# See Also

## More About

- “Model Configuration”

# Code Generation: Code Placement Tab Overview

Specify the data placement in the generated code.

### Configuration

This tab appears only if you specify an ERT based system target file (Simulink Coder).

### To get help on an option

1. Right-click the option text label.
2. From the context menu, select **What's This**.

## See Also

### Related Examples

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

# Data definition

## Description

Specify where to place definitions of global variables.

**Category:** Code Generation > Code Placement

## Settings

**Default:** `Auto`

`Auto`

Lets the code generator determine where the definitions should be located.

`Data defined in source file`

Places definitions in `.c` source files where functions are located. The code generator places the definitions in one or more function `.c` files, depending on the number of function source files and the file partitioning previously selected in the Simulink model.

`Data defined in a single separate source file`

Places definitions in the source file specified in the **Data definition filename** field. The code generator organizes and formats the definitions based on the data source template specified by the **Source file (*.c) template** parameter in the data section of the **Templates** pane.

## Dependencies

- This parameter applies to data with custom storage classes only.
- This parameter enables **Data definition filename**.

## Command-Line Information

**Parameter:** `GlobalDataDefinition`
**Type:** character vector
**Value:** `'Auto'` | `'InSourceFile'` | `'InSeparateSourceFile'`
**Default:** `'Auto'`

## Recommended Settings

| Application | Setting |
|---|---|
| Debugging | No impact |
| Traceability | A valid value |
| Efficiency | No impact |
| Safety precaution | No impact |

# See Also

## Related Examples

- "Model Configuration Parameters: Code Generation Code Placement" on page 7-2
- "Control Placement of Global Data Definitions and Declarations in Generated Files"

# Data definition filename

## Description

Specify the name of the file that is to contain data definitions.

**Category:** Code Generation > Code Placement

## Settings

**Default:** `global.c`

The code generator organizes and formats the data definitions in the specified file based on the data source template specified by the **Source file (*.c) template** parameter in the data section of the **Code Generation** pane: **Templates** tab.

You do not need to specify an extension for the file name. If you want to specify an extension, you must use a `.c` extension. In either case:

- If you select C as the target language, the code generator creates a file with a `.c` extension.
- If you select C++ as the target language, the code generator creates a file with a `.cpp` extension.

**Limitation**

The code generator does not check for unique filenames. Specify filenames that do not collide with default filenames from code generation.

## Dependency

This parameter is enabled by **Data definition**.

## Command-Line Information

**Parameter:** `DataDefinitionFile`
**Type:** character vector
**Value:** a valid file
**Default:** `'global.c'`

### Recommended Settings

| Application | Setting |
| --- | --- |
| Debugging | No impact |
| Traceability | A valid file |
| Efficiency | No impact |
| Safety precaution | No impact |

## See Also

### Related Examples

- “Model Configuration Parameters: Code Generation Code Placement” on page 7-2
- Selecting and Defining Templates
- Custom File Processing

# Data declaration

## Description

Specify where `extern`, `typedef`, and `#define` statements are to be declared.

**Category:** Code Generation > Code Placement

## Settings

**Default:** `Auto`

`Auto`

Lets the code generator determine where the declarations should be located.

`Data declared in source file`

Places declarations in `.c` source files where functions are located. The data header template file is not used. The code generator places the declarations in one or more function `.c` files, depending on the number of function source files and the file partitioning previously selected in the Simulink model.

`Data defined in a single separate source file`

Places declarations in the data header file specified in the **Data declaration filename** field. The code generator organizes and formats the declarations based on the data header template specified by the **header file (*.h) template** parameter in the data section of the **Code Generation** pane: **Templates** tab.

## Dependencies

- This parameter applies to data with custom storage classes only.
- This parameter enables **Data declaration filename**.

## Command-Line Information

**Parameter:** `GlobalDataReference`
**Type:** character vector
**Value:** `'Auto'` | `'InSourceFile'` | `'InSeparateHeaderFile'`
**Default:** `'Auto'`

### Recommended Settings

| Application | Setting |
|---|---|
| Debugging | No impact |
| Traceability | A valid value |
| Efficiency | No impact |
| Safety precaution | No impact |

## See Also

### Related Examples

- “Model Configuration Parameters: Code Generation Code Placement” on page 7-2
- “Control Placement of Global Data Definitions and Declarations in Generated Files”

# Data declaration filename

## Description

Specify the name of the file that is to contain data declarations.

**Category:** Code Generation > Code Placement

## Settings

**Default:** `global.h`

The code generator organizes and formats the data declarations in the specified file based on the data header template specified by the **Header file (*.h) template** parameter in the data section of the **Code Generation** pane: **Templates** tab.

### Limitation

The code generator does not check for unique filenames. Specify filenames that do not collide with default filenames from code generation.

## Dependency

This parameter is enabled by **Data declaration**.

## Command-Line Information

**Parameter:** `DataReferenceFile`
**Type:** character vector
**Value:** a valid file
**Default:** `'global.h'`

## Recommended Settings

| Application | Setting |
|---|---|
| Debugging | No impact |
| Traceability | A valid file |

| Application | Setting |
|---|---|
| Efficiency | No impact |
| Safety precaution | No impact |

# See Also

## Related Examples

- “Model Configuration Parameters: Code Generation Code Placement” on page 7-2
- Selecting and Defining Templates
- Custom File Processing

# Use owner from data object for data definition placement

## Description

Specify whether the model uses or ignores the ownership setting of a data object for data definition in code generation.

**Category:** Code Generation > Code Placement

## Settings

**Default:** off

On

Uses the ownership setting of the data object for data definition.

Off

Ignores the ownership setting of the data object for data definition.

## Command-Line Information

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

## Recommended Settings

| Application | Setting |
| --- | --- |
| Debugging | No impact |
| Traceability | A valid value |
| Efficiency | No impact |
| Safety precaution | No impact |

## See Also

### Related Examples

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

# #include file delimiter

## Description

Specify the type of `#include` file delimiter to use in generated code.

**Category:** Code Generation > Code Placement

## Settings

**Default:** `Auto`

`Auto`

Lets the code generator choose the `#include` file delimiter

`#include “header.h”`

Uses double quote (" ") characters to delimit file names in `#include` statements.

`#include <header.h>`

Uses angle brackets (< >) to delimit file names in `#include` statements.

## Dependency

The delimiter format that you use when specifying parameter and signal object property values overrides what you set for this parameter.

## Command-Line Information

**Parameter:** `IncludeFileDelimiter`
**Type:** character vector
**Value:** `'Auto'` | `'UseQuote'` | `'UseBracket'`
**Default:** `'Auto'`

## Recommended Settings

| Application | Setting |
| --- | --- |
| Debugging | No impact |

| Application | Setting |
| --- | --- |
| Traceability | A valid value |
| Efficiency | No impact |
| Safety precaution | No impact |

# See Also

## Related Examples

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

# Signal display level

## Description

Specify the persistence level for MPT signal data objects.

**Category:** Code Generation > Code Placement

## Settings

**Default:** 10

Specify an integer value indicating the persistence level for MPT signal data objects. This value indicates the level at which to declare signal data objects as global data in the generated code. The persistence level allows you to make intermediate variables global during initial development so you can remove them during later stages of development to gain efficiency.

This parameter is related to the **Persistence level** value that you can specify for a specific MPT signal data object in the Model Explorer signal properties dialog.

## Dependency

This parameter must be the same for top-level and referenced models.

## Command-Line Information

**Parameter:** `SignalDisplayLevel`
**Type:** integer
**Value:** a valid integer
**Default:** `10`

## Recommended Settings

| Application | Setting |
|---|---|
| Debugging | No impact |
| Traceability | A valid integer |

| Application | Setting |
| --- | --- |
| Efficiency | No impact |
| Safety precaution | No impact |

# See Also

## Related Examples

- “Model Configuration Parameters: Code Generation Code Placement” on page 7-2
- Selecting Persistence Level for Signals and Parameters

# Parameter tune level

## Description

Specify the persistence level for MPT parameter data objects.

**Category:** Code Generation > Code Placement

## Settings

**Default:** 10

Specify an integer value indicating the persistence level for MPT parameter data objects. This value indicates the level at which to declare parameter data objects as tunable global data in the generated code. The persistence level allows you to make intermediate variables global and tunable during initial development so you can remove them during later stages of development to gain efficiency.

This parameter is related to the **Persistence level** value you that can specify for a specific MPT parameter data object in the Model Explorer parameter properties dialog.

## Dependency

This parameter must be the same for top-level and referenced models.

## Command-Line Information

**Parameter:** `ParamTuneLevel`
**Type:** integer
**Value:** a valid integer
**Default:** `10`

## Recommended Settings

| Application | Setting |
| --- | --- |
| Debugging | No impact |
| Traceability | A valid integer |

| Application | Setting |
| --- | --- |
| Efficiency | No impact |
| Safety precaution | No impact |

# See Also

## Related Examples

- “Model Configuration Parameters: Code Generation Code Placement” on page 7-2
- Selecting Persistence Level for Signals and Parameters

# File packaging format

## Description

Specify whether code generation modularizes the code components into many files or compacts the generated code into a few files. You can specify a different file packaging format for each referenced model.

**Category:** Code Generation > Code Placement

## Settings

**Default:** `Modular`

`Modular`

- Outputs *`model`*`_data.c`, *`model`*`_private.h`, and *`model`*`_types.h`, in addition to generating *`model`*`.c` and *`model`*`.h`. For the contents of these files, see the table in "Generated Code Modules".
- Supports generating separate source files for subsystems. For more information on generating code for subsystems, see "Control Generation of Subsystem Functions" (Simulink Coder).
- If you specify **Shared code placement** as `Auto` on the **Code Generation** > **Interface** pane of the Configuration Parameter dialog box, some utility files are in the build directory. If you specify **Shared code placement** as `Shared location`, separate files are generated for utility code in a shared location.

`Compact (with separate data file)`

- Conditionally outputs *`model`*`_data.c`, in addition to generating *`model`*`.c` and *`model`*`.h`.
- If you specify **Shared code placement** as `Auto` on the **Code Generation** > **Interface** pane of the Configuration Parameter dialog box, utility algorithms are defined in *`model`*`.c`. If you specify **Shared code placement** as `Shared location`, separate files are generated for utility code in a shared location.
- Does not support separate source files for subsystems.
- Does not support models with noninlined S-functions.

`Compact`

- The contents of *`model`*`_data.c` are in *`model`*`.c`.
- The contents of *`model`*`_private.h` and *`model`*`_types.h` are in *`model`*`.h` or *`model`*`.c`.
- If you specify **Shared code placement** as `Auto` on the **Code Generation** > **Interface** pane of the Configuration Parameter dialog box, utility algorithms are defined in *`model`*`.c`. If you specify **Shared code placement** as `Shared location`, separate files are generated for utility code in a shared location.
- Does not support separate source files for subsystems.
- Does not support models with noninlined S-functions.

## Command-Line Information

**Parameter:** `ERTFilePackagingFormat`
**Type:** character vector
**Value:** `'Modular'` | `'CompactWithDataFile'` | `'Compact'`
**Default:** `'Modular'`

## Recommended Settings

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

# See Also

## Related Examples

- “Model Configuration Parameters: Code Generation Code Placement” on page 7-2
- “Customize Generated Code Modules”
- “Manage File Packaging of Generated Code Modules”

- “Customize Post-Code-Generation Build Processing” (Simulink Coder)
- “Generate Shared Utility Code” (Simulink Coder)

# Header files

## Description

Specify customized name for generated header files.

**Category:** Code Generation > Code Placement

## Settings

**Default:** `$R$E`

Enter a macro that specifies whether, and in what order, certain text is to be included in the generated identifier. The macro can include a combination of any custom user text and these format tokens.

| Token | Description |
|---|---|
| `$E` | Insert the file type. `$E` represents these instances of file types:<br>• `capi`<br>• `capi_host`<br>• `dt`<br>• `testinterface`<br>• `private`<br>• `types`<br><br>Required. |
| `$R` | Insert root model name into identifier, replacing unsupported characters with the underscore (_) character.<br><br>Required for model referencing. |
| `$U` | Insert text that you specify for the `$U` token. To specify this text, use the **Custom token text** (Simulink Coder) parameter. |

Custom naming is supported only for `.h` and `.hpp` files. When you have model hierarchy, custom naming is applicable to only the root model.

### Dependency

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

### Command-Line Information

**Parameter:** `ERTHeaderFileRootName`
**Type:** character vector
**Value:** Valid combination of tokens and custom text
**Default:** `$R$E`

### Recommended Settings

| Application | Setting |
| --- | --- |
| Debugging | No impact |
| Traceability | Use default |
| Efficiency | No impact |
| Safety precaution | No recommendation |

## See Also

### Related Examples

- “Model Configuration Parameters: Code Generation Code Placement” on page 7-2
- “Customize Generated File Names”
- “Identifier Format Control”

# Source files

## Description

Specify customized name for generated source files.

**Category:** Code Generation > Code Placement

## Settings

**Default:** `$R$E`

Enter a macro that specifies whether, and in what order, certain text is to be included in the generated identifier. The macro can include a combination of any custom user text and these format tokens:

| Token | Description |
| --- | --- |
| `$E` | Insert the file type. `$E` represents these instances of file types:<br>• `capi`<br>• `capi_host`<br>• `dt`<br>• `testinterface`<br>• `private`<br>• `types`<br><br>Required. |
| `$R` | Insert root model name into identifier, replacing unsupported characters with the underscore (_) character.<br><br>Required for model referencing. |
| `$U` | Insert text that you specify for the `$U` token. To specify this text, use the **Custom token text** (Simulink Coder) parameter. |

Custom naming is supported only for `.c` and `.cpp` files. When you have model hierarchy, custom naming is applicable to only the root model.

### Dependency

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

### Command-Line Information

**Parameter:** `ERTSourceFileRootName`
**Type:** character vector
**Value:** Valid combination of tokens and custom text
**Default:** `$R$E`

### Recommended Settings

| Application | Setting |
|---|---|
| Debugging | No impact |
| Traceability | Use default |
| Efficiency | No impact |
| Safety precaution | No recommendation |

## See Also

### Related Examples

- “Model Configuration Parameters: Code Generation Code Placement” on page 7-2
- “Customize Generated File Names”
- “Identifier Format Control”

# Data files

## Description

Specify customized name for generated data files.

**Category:** Code Generation > Code Placement

## Settings

**Default:** `$R_data`

Enter a macro that specifies whether, and in what order, certain text is to be included in the generated identifier. The macro can include a combination of any custom user text and these format tokens:

| Token | Description |
| --- | --- |
| `$R` | Insert root model name into identifier, replacing unsupported characters with the underscore (_) character.<br><br>Required for model referencing. |
| `$U` | Insert text that you specify for the `$U` token. To specify this text, use the **Custom token text** (Simulink Coder) parameter. |

Custom naming is supported only for `.c` and `.cpp` files. When you have model hierarchy, custom naming is applicable to only the root model.

## Dependency

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.
- `Compact (with separate data file)` option for **File packaging format** on page 7-21 enables this parameter.

### Command-Line Information

**Parameter:** `ERTDataFileRootName`
**Type:** character vector
**Value:** Valid combination of tokens and custom text
**Default:** `$R_data`

### Recommended Settings

| Application | Setting |
| --- | --- |
| Debugging | No impact |
| Traceability | Use default |
| Efficiency | No impact |
| Safety precaution | No recommendation |

## See Also

### Related Examples

- “Model Configuration Parameters: Code Generation Code Placement” on page 7-2
- “Customize Generated File Names”
- “Identifier Format Control”

# Rate Transition block code

## Description

Specify the format for Rate Transition block code and data. Inline the code with the model code or create separate functions that the model code calls with state data in a dedicated structure.

**Category:** Code Generation > Code Placement

## Settings

**Default:** `Inline`

`Inline`

Inline Rate Transition block code with model code. Declare Rate Transition block state data in global block state structure.

`Function`

Separate Rate Transition block code and data from the model code and data. The generated code contains separate `get` and `set` functions that the *`model`*`_step` functions call and a dedicated structure for state data. The generated code also contains separate start and initialize functions that the *`model`*`_initialize` function calls.

## Dependencies

- This parameter requires an Embedded Coder license.
- Appears only for ERT-based targets.

## Command-Line Information

**Parameter:** `RateTransitionBlockCode`
**Value:** `'Inline'` | `'Function'` |
**Default:** `'Inline'`

## Recommended Settings

| Application | Setting |
| --- | --- |
| Debugging | Function |
| Traceability | Function |
| Efficiency | Inline |
| Safety precaution | No impact |

**Note**

- The code generator does not separate code and data for Rate Transition blocks that have variable-size signals or are inside a For Each Subsystem block.
- In the Rate Transition block parameters dialog box, you must select the **Ensure data integrity during data transfer** parameter. If you do not select this parameter, the model produces an error during code generation.
- In Configuration Parameters dialog box, the **Multitask rate transition** parameter must be set to `error`. If this parameter is not set to `error`, Embedded Coder disables the **Rate Transition block code** parameter and the code generator inlines Rate Transition block code.

# See Also

## Related Examples

- “Time-Based Scheduling”

8

# Code Generation Parameters: Code Style

# Model Configuration Parameters: Code Generation Code Style

The **Code Generation > Code Style** category includes parameters for configuring the appearance of the generated code. These parameters require a Simulink Coder license. Additional parameters available with an ERT-based target require an Embedded Coder license.

On the Configuration Parameters dialog box, the following configuration parameters are on the **Commonly Used** tab on the **Code Generation > Code Style** pane.

| Parameter | Description |
|---|---|
| "Parentheses level" on page 8-5 | Specify parenthesization style for generated code. |
| "Preserve operand order in expression" on page 8-7 | Specify whether to preserve order of operands in expressions. |
| "Preserve condition expression in if statement" on page 8-9 | Specify whether to preserve empty primary condition expressions in `if` statements. |
| "Convert if-elseif-else patterns to switch-case statements" on page 8-11 | Specify whether to generate code for `if-elseif-else` decision logic as `switch-case` statements. |
| "Preserve extern keyword in function declarations" on page 8-13 | Specify whether to include the `extern` keyword in function declarations in the generated code. |
| "Preserve static keyword in function declarations" on page 8-15 | Specify whether to include the `static` keyword in function declarations in the generated code. |
| "Suppress generation of default cases for Stateflow switch statements if unreachable" on page 8-17 | Specify whether to generate default cases for switch-case statements in the code for Stateflow charts. |
| "Replace multiplications by powers of two with signed bitwise shifts" on page 8-19 | Specify whether to replace multiplications by powers of two with signed bitwise shifts. |
| "Allow right shifts on signed integers" on page 8-21 | Specify whether to allow signed right bitwise shifts in the generated C/C++ code. |

| Parameter | Description |
|---|---|
| “Casting modes” on page 8-23 | Specify how the code generator casts data types for variables. |
| “Indent style” on page 8-25 | Specify style for the placement of braces in generated code. |
| “Indent size” on page 8-27 | Specify indent size for generated code. |
| “Newline style” on page 8-29 | Specify the newline style for generated code. |

# See Also

## More About

- “Code Appearance”
- “Model Configuration”

# Code Generation: Code Style Tab Overview

Control optimizations for readability in generated code.

## Configuration

This tab appears only if you specify an ERT based system target file (Simulink Coder).

## To get help on an option

1. Right-click the option text label.
2. From the context menu, select **What's This**.

# See Also

## Related Examples

- “Model Configuration Parameters: Code Generation Code Style” on page 8-2
- “Control Code Style”

# Parentheses level

## Description

Specify parenthesization style for generated code.

**Category:** Code Generation > Code Style

## Settings

**Default:** `Nominal (Optimize for readability)`

**Minimum (Rely on C/C++ operators for precedence)**

Inserts parentheses only where required by ANSI[1] C or C++, or to override default precedence. For example:

```
Out = In2 - In1 > 1.0 && In2 > 2.0;
```

If you generate C/C++ code using the minimum level, for certain settings in some compilers, you can receive compiler warnings. To eliminate these warnings, try the nominal level.

**Nominal (Optimize for readability)**

Inserts parentheses in a way that compromises between readability and visual complexity. For example:

```
Out = ((In2 - In1 > 1.0) && (In2 > 2.0));
```

**Maximum (Specify precedence with parentheses)**

Includes parentheses to specify meaning without relying on operator precedence. Code generated with this setting conforms to MISRA®[2] requirements. For example:

```
Out = (((In2 - In1) > 1.0) && (In2 > 2.0));
```

## Command-Line Information

**Parameter:** `ParenthesesLevel`
**Type:** character vector

1. ANSI is a registered trademark of the American National Standards Institute, Inc.
2. MISRA is a registered trademarks of MIRA Ltd, held on behalf of the MISRA Consortium.

**Value:** 'Minimum' | 'Nominal' | 'Maximum'
**Default:** 'Nominal'

### Recommended Settings

| Application | Setting |
| --- | --- |
| Debugging | Nominal (Optimized for readability) |
| Traceability | Nominal (Optimized for readability) |
| Efficiency | Minimum (Rely on C/C++ operators for precedence) |
| Safety precaution | No recommendation |

## See Also

### Related Examples

- "Model Configuration Parameters: Code Generation Code Style" on page 8-2
- Control Parentheses in Generated Code

# Preserve operand order in expression

## Description

Specify whether to preserve order of operands in expressions.

**Category:** Code Generation > Code Style

## Settings

**Default:** off

On

Preserves the expression order specified in the model. Select this option to increase readability of the code or for code traceability purposes.

```
A*(B+C)
```

Off

Optimizes efficiency of code for nonoptimized compilers by reordering commutable operands to make expressions left-recursive. For example:

```
(B+C)*A
```

## Command-Line Information

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

## Recommended Settings

| Application | Setting |
| --- | --- |
| Debugging | On |
| Traceability | On |
| Efficiency | Off |

| Application | Setting |
| --- | --- |
| Safety precaution | No recommendation |

## See Also

### Related Examples

- “Model Configuration Parameters: Code Generation Code Style” on page 8-2
- “Optimize Code by Reordering Commutable Operands”

# Preserve condition expression in if statement

## Description

Specify whether to preserve empty primary condition expressions in `if` statements.

**Category:** Code Generation > Code Style

## Settings

**Default:** off

On

Preserves empty primary condition expressions in `if` statements, such as the following, to increase the readability of the code or for code traceability purposes.

```
if expression1
else
       statements2;
end
```

Off

Optimizes empty primary condition expressions in `if` statements by negating them. For example, consider the following `if` statement:

```
if expression1
else
       statements2;
end
```

By default, the code generator negates this statement as follows:

```
if ~expression1
        statements2;
end
```

## Command-Line Information

**Parameter:** `PreserveIfCondition`
**Type:** character vector

**Value:** `'on'` | `'off'`
**Default:** `'off'`

### Recommended Settings

| Application | Setting |
|---|---|
| Debugging | On |
| Traceability | On |
| Efficiency | Off |
| Safety precaution | No recommendation |

## See Also

### Related Examples

- “Model Configuration Parameters: Code Generation Code Style” on page 8-2
- “Optimize Generated Code by Consolidating Redundant If-Else Statements”

# Convert if-elseif-else patterns to switch-case statements

## Description

Specify whether to generate code for `if-elseif-else` decision logic as `switch-case` statements.

This readability optimization works on a per-model basis and applies only to:

- Flow charts in Stateflow charts
- MATLAB functions in Stateflow charts
- MATLAB Function blocks in that model

**Category:** Code Generation > Code Style

## Settings

**Default:** on

On

Generate code for `if-elseif-else` decision logic as `switch-case` statements.

For example, assume that you have the following logic pattern:

```
if (x == 1) {
       y = 1;
} else if (x == 2) {
       y = 2;
} else if (x == 3) {
       y = 3;
} else {
       y = 4;
}
```

Selecting this check box converts the `if-elseif-else` pattern to the following `switch-case` statements:

```
switch (x) {
     case 1:
```

```
        y = 1; break;
      case 2:
        y = 2; break;
      case 3:
        y = 3; break;
      default:
        y = 4; break;
}
```

☐ Off

Preserve `if-elseif-else` decision logic in generated code.

### Command-Line Information

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

### Recommended Settings

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

## See Also

### Related Examples

- "Model Configuration Parameters: Code Generation Code Style" on page 8-2
- "Enhance Readability of Code for Flow Charts"
- "Enhance Code Readability for MATLAB Function Blocks" (Simulink)
- "Control Code Style"

# Preserve extern keyword in function declarations

## Description

Specify whether to include the `extern` keyword in function declarations in the generated code.

---

**Note** The `extern` keyword is optional for functions with external linkage. It is considered good programming practice to include the `extern` keyword in function declarations for code readability.

---

**Category:** Code Generation > Code Style

## Settings

**Default:** on

On

Include the `extern` keyword in function declarations in the generated code. For example, the generated code for the model `rtwdemo_hyperlinks` contains the following function declarations in `rtwdemo_hyperlinks.h`:

```
/* Model entry point functions */
extern void rtwdemo_hyperlinks_initialize(void);
extern void rtwdemo_hyperlinks_step(void);
```

The `extern` keyword explicitly indicates that the function has external linkage. The function definitions in this example are in the generated file `rtwdemo_hyperlinks.c`.

Off

Remove the `extern` keyword from function declarations in the generated code.

## Command-Line Information

**Parameter:** `PreserveExternInFcnDecls`
**Type:** character vector

**Value:** `'on'` | `'off'`
**Default:** `'on'`

### Recommended Settings

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

## See Also

### Related Examples

- "Model Configuration Parameters: Code Generation Code Style" on page 8-2

# Preserve static keyword in function declarations

## Description

Specify whether to include the `static` keyword in function declarations in the generated code.

**Category:** Code Generation > Code Style

## Settings

**Default:** on

On

Include the `static` keyword in function declarations in the generated code. You can link different executables generated from different models that refer to locally scoped subsystem and utility functions with the same name. This parameter also impacts these functions:

- Stateflow graphical function
- Variant subsystem
- MATLAB subfunction
- Privately scoped Simulink function

When you select this parameter, the generated code is compliant with MISRA C:2012 Rule 8.10.

Off

Remove the `static` keyword in function declarations in the generated code.

## Dependency

- This parameter requires Embedded Coder license when you generate code.
- This parameter appears only for ERT-based targets.
- This parameter is enabled when you select `Compact/Compact(with separate data file)` file packaging.

### Command-Line Information

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

### Recommended Settings

| Application | Setting |
|---|---|
| Debugging | No impact |
| Traceability | No impact |
| Efficiency | On (execution, ROM) |
| Safety precaution | No impact |

## See Also

### Related Examples

- "Model Configuration Parameters: Code Generation Code Style" on page 8-2
- `MISRA C:2012 Rule 8.10`

# Suppress generation of default cases for Stateflow switch statements if unreachable

## Description

Specify whether to generate default cases for switch-case statements in the code for Stateflow charts. This optimization works on a per-model basis. It applies to the code generated for a state that has multiple substates. For a list of the state functions in the generated code, see "Inline State Functions in Generated Code" (Simulink Coder).

**Category:** Code Generation > Code Style

## Settings

**Default:** on

On

Do not generate the default case when it is unreachable. This setting enables better code coverage because every branch in the generated code is falsifiable.

Off

Generate a default case whether or not it is reachable. This setting supports MISRA C compliance and provides a backup in case of RAM corruption.

For example, when the state has a nontrivial `entry` function, the following default case appears in the generated code for the `during` function:

```
default:
 entry_internal();
 break;
```

In this case, the code marks the corresponding substate as active.

## Command-Line Information

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

## Recommended Settings

| Application | Setting |
|---|---|
| Debugging | Noimpact |
| Traceability | On |
| Efficiency | On (execution, ROM), Noimpact (RAM) |
| Safety precaution | No recommendation |

# See Also

## Related Examples

- “Model Configuration Parameters: Code Generation Code Style” on page 8-2
- “Suppress Generation of Default Cases for Unreachable Stateflow Switch Statements”

# Replace multiplications by powers of two with signed bitwise shifts

## Description

Specify whether to replace multiplications by powers of two with signed bitwise shifts. Some coding standards, such as MISRA, do not allow bitwise operations on signed integers. Clearing this option increases the likelihood of generating MISRA C compliant code.

**Category:** Code Generation > Code Style

## Settings

**Default:** on

On

Generate code that replaces multiplications by powers of two with signed bitwise shifts.

For example, when you select this option, multiplications by 8 are left-shifted in the generated code:

```
Y.Out1 = (U.In1 << ((int8_T)3));
```

Similarly, multiplications by 16 are left-shifted in the generated code:

```
Y.Out4 = (U.In2 << ((int8_T)4));
```

Off

Do not allow replacement of multiplications by powers of two with signed shifts. Clearing this option supports MISRA C compliance.

For example, when you clear this option, multiplications by 8 are not replaced by bitwise shifts:

```
Y.Out1 = U.In1 * ((int64_T)8);
```

Similarly, multiplications by 16 are not replaced by bitwise shifts:

```
Y.Out4 = U.In2 * ((int32_T)16);
```

### Command-Line Information

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

### Recommended Settings

| Application | Setting |
|---|---|
| Debugging | Noimpact |
| Traceability | Noimpact |
| Efficiency | On |
| Safety precaution | Noimpact |

## See Also

### Related Examples

- “Model Configuration Parameters: Code Generation Code Style” on page 8-2
- “Replace Multiplication by Powers of Two with Signed Bitwise Shifts”

# Allow right shifts on signed integers

## Description

Specify whether to allow signed right bitwise shifts in the generated C/C++ code. Some coding standards, such as MISRA, do not allow bitwise operations on signed integers. Clearing this option increases the likelihood of generating MISRA-C:2004 compliant code.

**Category:** Code Generation > Code Style

## Settings

**Default:** on

On

Generate code that uses right bitwise shifts on signed integers.

For example, when you select this option, right shifts appear in the generated code.

```
i >>= 3
```

Off

Do not allow right shifts on signed integers. Clearing this option supports MISRA C compliance.

For example, when you clear this option, right shifts are replaced with a function call.

```
i = asr_s32(i, 3U);
```

## Command-Line Information

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

### Recommended Settings

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

## See Also

### Related Examples

- "Model Configuration Parameters: Code Generation Code Style" on page 8-2
- "Generate Code with Right Shifts on Signed Integers"

# Casting modes

## Description

Specify how the code generator casts data types for variables.

**Category:** Code Generation > Code Style

## Settings

**Default:** `Nominal`

**Nominal**

Generate code that uses default C compiler data type casting.

```
void rtwdemo_rtwecintro_step(void)
{
  boolean_T rtb_equal_to_count;
  rtDWork.X++;
  rtb_equal_to_count = (rtDWork.X != 16);
  if (rtb_equal_to_count && (rtPrevZCSigState.Amplifier_Trig_ZCE != POS_ZCSIG))
  {
    rtY.Output = rtU.Input << 1;
  }
```

**Standards Compliant**

Generate code that casts data types to conform to MISRA standards.

```
void rtwdemo_rtwecintro_step(void)
{
  boolean_T rtb_equal_to_count;
     rtDWork.X++;
  rtb_equal_to_count = (boolean_T)(int32_T)((int32_T)rtDWork.X != (int32_T)16);
  if (((int32_T)rtb_equal_to_count) && (rtPrevZCSigState.Amplifier_Trig_ZCE !=
       POS_ZCSIG)) {
    rtY.Output = (int32_T)(uint32_T)((uint32_T)rtU.Input << (uint32_T)(int8_T)1);
  }
```

**Explicit**

Generate code that casts data type values explicitly.

```
/* Model step function */
void rtwdemo_rtwecintro_step(void)
{
  boolean_T rtb_equal_to_count;
  rtDWork.X = (uint8_T)(1U + (uint32_T)(int32_T)rtDWork.X);
  rtb_equal_to_count = (boolean_T)((int32_T)rtDWork.X != 16);
  if (((int32_T)rtb_equal_to_count) && ((int32_T)((int32_T)
        rtPrevZCSigState.Amplifier_Trig_ZCE != (int32_T)POS_ZCSIG))) {
    rtY.Output = rtU.Input << 1;
  }
```

### Command-Line Information

**Parameter:** `CastingMode`
**Type:** character vector
**Value:** `'Nominal'` | `'Standards'` | `'Explicit'`
**Default:** `'Nominal'`

### Recommended Settings

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

## See Also

### Related Examples

- “Model Configuration Parameters: Code Generation Code Style” on page 8-2
- “Control Cast Expressions in Generated Code”
- “MISRA C Guidelines”

# Indent style

## Description

Specify style for the placement of braces in generated code.

**Category:** Code Generation > Code Style

## Settings

**Default:** K&R

**K&R**

For blocks within a function, an opening brace is on the same line as its control statement. For example:

```
void rt_OneStep(void)
{
  static boolean_T OverrunFlag = 0;
  if (OverrunFlag) {
    rtmSetErrorStatus(rtwdemo_counter_M, "Overrun");
    return;
  }

  OverrunFlag = TRUE;
  rtwdemo_counter_step();
  OverrunFlag = FALSE;
}
```

**Allman**

For blocks within a function, an opening brace is on its own line at the same level of indentation as its control statement. For example:

```
void rt_OneStep(void)
{
  static boolean_T OverrunFlag = 0;
  if (OverrunFlag)
  {
    rtmSetErrorStatus(rtwdemo_counter_M, "Overrun");
    return;
  }
```

```
    OverrunFlag = TRUE;
    rtwdemo_counter_step();
    OverrunFlag = FALSE;
  }
```

### Command-Line Information

**Parameter:** `IndentStyle`
**Type:** character vector
**Value:** `'K&R'` | `'Allman'`
**Default:** `'K&R'`

### Recommended Settings

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

## See Also

### Related Examples

- “Model Configuration Parameters: Code Generation Code Style” on page 8-2
- “Control Indentation Style in Generated Code”

# Indent size

## Description

Specify indent size for generated code.

**Category:** Code Generation > Code Style

## Settings

**Default:** 2

Specify an integer value that indicates the number of characters per indent level. Possible values range from 2–8 characters.

## Command-Line Information

**Parameter:** `IndentSize`
**Type:** integer
**Value:** integer from 2–8
**Default:** 2

## Recommended Settings

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

# See Also

## Related Examples

- “Model Configuration Parameters: Code Generation Code Style” on page 8-2

- “Control Indentation Style in Generated Code”

# Newline style

## Description

Specify the newline character in the generated code.

**Category:** Code Generation > Code Style

## Settings

**Default:** `Default`

**Default**

Generates the newline character based on the operating system that the code is generated on.

**LF (Line Feed)**

Generates the Line Feed character as the newline character in the generated code. `"\n"` is inserted as the newline character.

**CR+LF (Carriage Return + Line Feed)**

Generates the Carriage Return + Line Feed character as the newline character in the generated code. `"\r\n"` is inserted as the newline character.

## Command-Line Information

**Parameter:** `NewlineStyle`
**Type:** character vector
**Value:** `'Default'|'LF'|'CRLF'`
**Default:** `'Default'`

## Recommended Settings

| Application | Setting |
|---|---|
| Debugging | No impact |
| Traceability | No impact |
| Efficiency | No impact |

| Application | Setting |
| --- | --- |
| Safety precaution | No impact |

## See Also

### Related Examples

- “Model Configuration Parameters: Code Generation Code Style” on page 8-2
- “Control Newline Style in Generated Code”

# Code Generation Parameters: Data Type Replacement

# Model Configuration Parameters: Code Generation Data Type Replacement

The **Code Generation > Data Type Replacement** category includes parameters for replacing built-in data type names with user-defined names in the generated code. On the Configuration Parameters dialog box, the following configuration parameters are on the **Commonly Used** tab on the **Code Generation > Data Type Replacement** pane.

| Parameter | Description |
|---|---|
| "Replace data type names in the generated code" on page 9-5 | Specify whether to replace built-in data type names with user-defined data type names in generated code. |
| "Replacement Name: double" on page 9-7 | Specify a name for `double` built-in data types in generated code. |
| "Replacement Name: single" on page 9-9 | Specify a name for `single` built-in data types in generated code. |
| "Replacement Name: int32" on page 9-11 | Specify a name for `int32_T` built-in data types in generated code. |
| "Replacement Name: int16" on page 9-13 | Specify a name for `int16_T` built-in data types in generated code. |
| "Replacement Name: int8" on page 9-15 | Specify a name for `int8_T` built-in data types in generated code. |
| "Replacement Name: uint32" on page 9-17 | Specify a name for `uint32_T` built-in data types in generated code. |
| "Replacement Name: uint16" on page 9-19 | Specify a name for `uint16_T` built-in data types in generated code. |
| "Replacement Name: uint8" on page 9-21 | Specify a name for `uint8_T` built-in data types in generated code. |
| "Replacement Name: boolean" on page 9-23 | Specify a name for `boolean_T` built-in data types in generated code. |
| "Replacement Name: int" on page 9-26 | Specify a name for `int_T` built-in data types in generated code. |
| "Replacement Name: uint" on page 9-28 | Specify a name for `uint_T` built-in data types in generated code. |

| Parameter | Description |
| --- | --- |
| "Replacement Name: char" on page 9-30 | Specify a name for char_T built-in data types in generated code. |

## Configure Data Type Replacements Programmatically

To programmatically replace the built-in data type names for your model, adjust the ReplacementTypes model parameter, which is a structure. This example code shows how to modify the ReplacementTypes parameter to replace the built-in data type names int8, uint8, and boolean with the custom data type names my_T_S8, my_T_U8, and my_T_BOOL.

```
model = bdroot;
cs = getActiveConfigSet(model);
set_param(cs,'EnableUserReplacementTypes','on');

struc = get_param(cs,'ReplacementTypes');
struc.int8 = 'my_T_S8';
struc.uint8 = 'my_T_U8';
struc.boolean = 'my_T_BOOL';

set_param(cs,'ReplacementTypes',struc);
```

# See Also

## More About

- "Control Data Type Names in Generated Code"
- "Data Type Replacement Limitations"
- "Model Configuration"

# Code Generation: Data Type Replacement Tab

Replace built-in data type names with user-defined replacement data type names in the generated code for your model.

## Configuration

This tab is visible only if you specify an ERT-based system target file (Simulink Coder).

1. Select **Replace data type names in the generated code**.
2. In the **Replacement Name** fields, selectively specify replacement data type names to use for built-in Simulink data types.

## To get help on an option

1. Right-click the option text label.
2. From the context menu, select **What's This**.

# See Also

## Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 9-2
- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”

# Replace data type names in the generated code

## Description

Specify whether to replace built-in data type names with user-defined data type names in generated code.

**Category:** Code Generation > Data Type Replacement

## Settings

**Default:** off

On

Displays the **Data type names** table. The table provides a way for you to replace the names of built-in data types used in generated code. This mechanism can be particularly useful for generating code that adheres to application or site data type naming standards.

You can choose to specify new data type names for some or all Simulink built-in data types listed in the table. Specify the replacement name as one of the following:

- A Simulink.AliasType object.
- A Simulink.NumericType object.
- The **Simulink Name** built-in data type name.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Off

Uses Simulink Coder names for built-in Simulink data types in generated code.

### Dependencies

This parameter enables replacement for all built-in data type name in the **Data type names** table with user-defined data type names in generated code.

### Command-Line Information

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

### Recommended Settings

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

## See Also

### Related Examples

- "Model Configuration Parameters: Code Generation Data Type Replacement" on page 9-2
- "Control Data Type Names in Generated Code"
- "Data Type Replacement Limitations"
- `Simulink.AliasType`
- `Simulink.NumericType`

# Replacement Name: double

## Description

Specify a name for double built-in data types in generated code.

**Category:** Code Generation > Data Type Replacement

## Settings

**Default:** ''

If a value is not specified, the code generator uses the **Code Generation Name**, real_T.

Specify a character vector for the code generator to use as a name for double built-in data types.

Specify the replacement name as one of the following:

- A Simulink.AliasType object.
- A Simulink.NumericType object.
- The **Simulink Name** built-in data type name.

To use the **Simulink Name**, specify double in the **Replacement Name** column.

To replace the **Code Generation Name** for double with an object:

- For a Simulink.AliasType object, set the BaseType object property to double.
- For a Simulink.NumericType object, set the DataTypeMode object property to Double.
- Specify the object name in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The Simulink.AliasType object has the **Data scope** parameter set to Exported.

### Dependency

This parameter is enabled by **Replace data type names in the generated code**.

### Command-Line Information

**Parameter:** `ReplacementTypes, replacementName.double`
**Type:** character vector
**Value:** The **Simulink Name** , a `Simulink.AliasType` object, or a `Simulink.NumericType` object, where the object exists in the base workspace.
**Default:** `''`

### Recommended Settings

| Application | Setting |
| --- | --- |
| Debugging | No impact |
| Traceability | A valid character vector |
| Efficiency | No impact |
| Safety precaution | No recommendation |

## See Also

### Related Examples

- "Model Configuration Parameters: Code Generation Data Type Replacement" on page 9-2
- "Control Data Type Names in Generated Code"
- "Data Type Replacement Limitations"
- `Simulink.AliasType`
- `Simulink.NumericType`

# Replacement Name: single

## Description

Specify a name for `single` built-in data types in generated code.

**Category:** Code Generation > Data Type Replacement

## Settings

**Default:** `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `real32_T`.

Specify a character vector for the code generator to use as a name for `single` built-in data types.

Specify the replacement name as one of the following:

- A Simulink.AliasType object.
- A Simulink.NumericType object.
- The **Simulink Name** built-in data type name.

To use the **Simulink Name**, specify `single` in the **Replacement Name** column.

To replace the **Code Generation Name** for `single` with an object:

- For a `Simulink.AliasType` object, set the `BaseType` object property to `single`.
- For a `Simulink.NumericType` object, set the `DataTypeMode` object property to `Single`.
- Specify the object name in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

### Dependency

This parameter is enabled by **Replace data type names in the generated code**.

### Command-Line Information

**Parameter:** `ReplacementTypes, replacementName.single`
**Type:** character vector
**Value:** The **Simulink Name** , the name of a `Simulink.AliasType` object, or the name of a `Simulink.NumericType` object, where the object exists in the base workspace.
**Default:** `''`

### Recommended Settings

| Application | Setting |
| --- | --- |
| Debugging | No impact |
| Traceability | A valid character vector |
| Efficiency | No impact |
| Safety precaution | `''` |

## See Also

### Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 9-2
- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- `Simulink.AliasType`
- `Simulink.NumericType`

# Replacement Name: int32

## Description

Specify names for built-in Simulink data types in generated code.

**Category:** Code Generation > Data Type Replacement

## Settings

**Default:** `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `int32_T`.

Specify character vectors for the code generator to use as names for built-in Simulink data types.

- A Simulink.AliasType object.
- The **Simulink Name** built-in data type name.

To replace the **Code Generation Name** `int32_T`:

- For a `Simulink.AliasType` object, set the `BaseType` object property to `int32`.
- To use the built-in data type name that matches the **Code Generation Name**, specify `int32` in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

## Dependency

This parameter is enabled by **Replace data type names in the generated code**.

## Command-Line Information

**Parameter:** `ReplacementTypes, replacementName.int32`
**Type:** character vector
**Value:** The **Simulink Name** or the name of a `Simulink.AliasType` object, where the object exists in the base workspace.
**Default:** `''`

## Recommended Settings

| Application | Setting |
|---|---|
| Debugging | No impact |
| Traceability | A valid character vector |
| Efficiency | No impact |
| Safety precaution | `''` |

# See Also

## Related Examples

- "Model Configuration Parameters: Code Generation Data Type Replacement" on page 9-2
- "Control Data Type Names in Generated Code"
- "Data Type Replacement Limitations"
- `Simulink.AliasType`

# Replacement Name: int16

## Description

Specify names for built-in Simulink data types in generated code.

**Category:** Code Generation > Data Type Replacement

## Settings

**Default:** `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `int16_T`.

Specify character vectors for the code generator to use as names for built-in Simulink data types.

- A Simulink.AliasType object.
- The **Simulink Name** built-in data type name.

To replace the **Code Generation Name** `int16_T`:

- For a `Simulink.AliasType` object, set the `BaseType` object property to `int16`.
- To use the built-in data type name that matches the **Code Generation Name**, specify `int16` in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

## Dependency

This parameter is enabled by **Replace data type names in the generated code**.

### Command-Line Information

**Parameter:** `ReplacementTypes, replacementName.int16`
**Type:** character vector
**Value:** The **Simulink Name** or the name of a `Simulink.AliasType` object, where the object exists in the base workspace.
**Default:** `''`

### Recommended Settings

| Application | Setting |
| --- | --- |
| Debugging | No impact |
| Traceability | A valid character vector |
| Efficiency | No impact |
| Safety precaution | `''` |

## See Also

### Related Examples

- "Model Configuration Parameters: Code Generation Data Type Replacement" on page 9-2
- "Control Data Type Names in Generated Code"
- "Data Type Replacement Limitations"
- `Simulink.AliasType`

# Replacement Name: int8

## Description

Specify names for built-in Simulink data types in generated code.

**Category:** Code Generation > Data Type Replacement

## Settings

**Default:** `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `int8_T`.

Specify character vectors for the code generator to use as names for built-in Simulink data types.

To replace the **Code Generation Name** `int8_T`:

- A Simulink.AliasType object.
- The **Simulink Name** built-in data type name.

- For a `Simulink.AliasType` object, set the `BaseType` object property to `int8`.
- To use the built-in data type name that matches the **Code Generation Name**, specify `int8` in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

## Dependency

This parameter is enabled by **Replace data type names in the generated code**.

## Command-Line Information

**Parameter:** `ReplacementTypes, replacementName.int8`

**Type:** character vector
**Value:** The **Simulink Name** or the name of a `Simulink.AliasType` object, where the object exists in the base workspace.
**Default:** `''`

### Recommended Settings

| Application | Setting |
|---|---|
| Debugging | No impact |
| Traceability | A valid character vector |
| Efficiency | No impact |
| Safety precaution | `''` |

## See Also

### Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 9-2
- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- `Simulink.AliasType`

# Replacement Name: uint32

## Description

Specify names for built-in Simulink data types in generated code.

**Category:** Code Generation > Data Type Replacement

## Settings

**Default:** `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `uint32_T`.

Specify character vectors for the code generator to use as names for built-in Simulink data types.

- A Simulink.AliasType object.
- The **Simulink Name** built-in data type name.

To replace the **Code Generation Name** `uint32_T`:

- For a `Simulink.AliasType` object, set the `BaseType` object property to `uint32`.
- To use the built-in data type name that matches the **Code Generation Name**, specify `uint32 c` in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

## Dependency

This parameter is enabled by **Replace data type names in the generated code**.

### Command-Line Information

**Parameter:** `ReplacementTypes, replacementName.uint32`
**Type:** character vector
**Value:** The **Simulink Name** or the name of a `Simulink.AliasType` object, where the object exists in the base workspace.
**Default:** `''`

### Recommended Settings

| Application | Setting |
|---|---|
| Debugging | No impact |
| Traceability | A valid character vector |
| Efficiency | No impact |
| Safety precaution | `''` |

## See Also

### Related Examples

- "Model Configuration Parameters: Code Generation Data Type Replacement" on page 9-2
- "Control Data Type Names in Generated Code"
- "Data Type Replacement Limitations"
- `Simulink.AliasType`

# Replacement Name: uint16

## Description

Specify names for built-in Simulink data types in generated code.

**Category:** Code Generation > Data Type Replacement

## Settings

**Default:** `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `uint16_T`.

Specify character vectors for the code generator to use as names for built-in Simulink data types.

- A Simulink.AliasType object.
- The **Simulink Name** built-in data type name.

To replace the **Code Generation Name** `uint16_T`:

- For a `Simulink.AliasType` object, set the `BaseType` object property to `uint16`.
- To use the built-in data type name that matches the **Code Generation Name**, specify `uint16` in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

## Dependency

This parameter is enabled by **Replace data type names in the generated code**.

### Command-Line Information

**Parameter:** `ReplacementTypes, replacementName.uint16`
**Type:** character vector
**Value:** The **Simulink Name** or the name of a `Simulink.AliasType` object, where the object exists in the base workspace.
**Default:** `''`

### Recommended Settings

| Application | Setting |
|---|---|
| Debugging | No impact |
| Traceability | A valid character vector |
| Efficiency | No impact |
| Safety precaution | `''` |

## See Also

### Related Examples

- "Model Configuration Parameters: Code Generation Data Type Replacement" on page 9-2
- "Control Data Type Names in Generated Code"
- "Data Type Replacement Limitations"
- `Simulink.AliasType`

# Replacement Name: uint8

## Description

Specify names for built-in Simulink data types in generated code.

**Category:** Code Generation > Data Type Replacement

## Settings

**Default:** `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `uint8_T`.

Specify character vectors for the code generator to use as names for built-in Simulink data types.

- A Simulink.AliasType object.
- The **Simulink Name** built-in data type name.

To replace the **Code Generation Name** `uint8_T`:

- For a `Simulink.AliasType` object, set the `BaseType` object property to `uint8`.
- To use the built-in data type name that matches the **Code Generation Name**, specify `uint8` in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

## Dependency

This parameter is enabled by **Replace data type names in the generated code**.

### Command-Line Information

**Parameter:** `ReplacementTypes, replacementName.uint8`
**Type:** character vector
**Value:** The **Simulink Name** or the name of a `Simulink.AliasType` object, where the object exists in the base workspace.
**Default:** `''`

### Recommended Settings

| Application | Setting |
|---|---|
| Debugging | No impact |
| Traceability | A valid character vector |
| Efficiency | No impact |
| Safety precaution | `''` |

## See Also

### Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 9-2
- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- `Simulink.AliasType`

# Replacement Name: boolean

## Description

Specify names for built-in Simulink data types in generated code.

**Category:** Code Generation > Data Type Replacement

## Settings

**Default:** `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `boolean_T`.

Specify character vectors for the code generator to use as names for built-in Simulink data types.

For ERT S-functions, the replacement data type can be only an 8-bit integer, `int8`, or `uint8`.

Specify the replacement name as one of the following:

- A Simulink.AliasType object.
- A Simulink.NumericType object.
- The **Simulink Name** built-in data type name.

To replace the **Code Generation Name** `boolean_T`:

- For a `Simulink.AliasType` object, set the `BaseType` object property to `boolean`, `uint8` , `int8`, or `int`*n*. *n* is the number of bits set for **Configuration Parameters > Hardware ImplementationNumber of bits: int**. Specify the name of the `Simulink.AliasType` object in the **Replacement Name** column.
- For a `Simulink.NumericType` object, to replace `real_T`, set the`DataTypeMode` object property to`Boolean`. Specify the name of the `Simulink.NumericType` object in the **Replacement Name** column.
- To use the Simulink Name built-in data type name which matches the Code Generation name, in the **Replacement Name** column, specify `uint8`, `int8`, or `int`*n*, where *n* is

the number of bits set for **Configuration Parameters > Hardware ImplementationNumber of bits: int**.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

### Dependency

This parameter is enabled by **Replace data type names in the generated code**.

### Command-Line Information

**Parameter:** `ReplacementTypes, replacementName.boolean`
**Type:** character vector
**Value:** The **Simulink Name** , a `Simulink.AliasType` object, or a `Simulink.NumericType` object, where the object exists in the base workspace.
**Default:** `''`

### Recommended Settings

| Application | Setting |
|---|---|
| Debugging | No impact |
| Traceability | A valid character vector |
| Efficiency | No impact |
| Safety precaution | `''` |

## See Also

### Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 9-2
- “Replace boolean with Specific Integer Data Type”

- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- `Simulink.AliasType`
- `Simulink.NumericType`

# Replacement Name: int

## Description

Specify names for built-in Simulink data types in generated code.

**Category:** Code Generation > Data Type Replacement

## Settings

**Default:** `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `int_T`.

Specify character vectors for the code generator to use as names for built-in Simulink data types.

Specify the replacement name as one of the following:

- A Simulink.AliasType object.
- The **Simulink Name** built-in data type name.

To replace the **Code Generation Name** `int_T`:

- For a `Simulink.AliasType` object

  Set the `BaseType` object property to `int`*n*. Specify the name of the `Simulink.AliasType` object in the **Replacement Name** column.
- To use the **Simulink Name** for int_T, in the **Replacement Name** column, specify `int`*n*.

*n* is the number of bits displayed in the Configuration Parameters dialog box, **Hardware Implementation** pane > **Number of bits: int**.

An error occurs, if

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

### Dependency

This parameter is enabled by **Replace data type names in the generated code**.

### Command-Line Information

**Parameter:** `ReplacementTypes, replacementName.int`
**Type:** character vector
**Value:** The **Simulink Name** or the name of a `Simulink.AliasType`, where the object exists in the base workspace.
**Default:** `''`

### Recommended Settings

| Application | Setting |
| --- | --- |
| Debugging | No impact |
| Traceability | A valid value |
| Efficiency | No impact |
| Safety precaution | `''` |

## See Also

### Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 9-2
- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- `Simulink.AliasType`

# Replacement Name: uint

## Description

Specify names to use for built-in Simulink data types in generated code.

**Category:** Code Generation > Data Type Replacement

## Settings

**Default:** `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `uint_T`.

Specify character vectors for the code generator to use as names for built-in Simulink data types.

Specify the replacement name as one of the following:

- A Simulink.AliasType object.
- The **Simulink Name** built-in data type name.

To replace the **Code Generation Name** `uint_T`:

- For a `Simulink.AliasType` object

  Set the `BaseType` object property to `uint`*n*. Specify the name of the `Simulink.AliasType` object in the **Replacement Name** column.
- To use the **Simulink Name** for `uint_T`, in the **Replacement Name** column, specify `uint`*n*.

*n* is the number of bits displayed in the Configuration Parameters dialog box, **Hardware Implementation** pane > **Number of bits: int**.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

### Dependency

This parameter is enabled by **Replace data type names in the generated code**.

### Command-Line Information

**Parameter:** `ReplacementTypes, replacementName.uint`
**Type:** character vector
**Value:** The **Simulink Name** or the name of a `Simulink.NumericType`, where the object exists in the base workspace.
**Default:** `''`

### Recommended Settings

| Application | Setting |
|---|---|
| Debugging | No impact |
| Traceability | A valid character vector |
| Efficiency | No impact |
| Safety precaution | `''` |

## See Also

### Related Examples

- "Model Configuration Parameters: Code Generation Data Type Replacement" on page 9-2
- "Control Data Type Names in Generated Code"
- "Data Type Replacement Limitations"
- `Simulink.AliasType`

# Replacement Name: char

## Description

Specify names for built-in Simulink data types in generated code.

**Category:** Code Generation > Data Type Replacement

## Settings

**Default:** `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `char_T`.

Specify character vectors for the code generator to use as names for built-in Simulink data types.

To replace the **Code Generation Name** `char_T`, create a `Simulink.AliasType` object in the Command Window.

Set the `BaseType` object property to `int`*n*. Specify the name of the `Simulink.AliasType` object in the **Replacement Name** column. *n* is the number of bits set for **Configuration Parameters > Hardware ImplementationNumber of bits: char**.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

## Dependency

This parameter is enabled by **Replace data type names in the generated code**.

## Command-Line Information

**Parameter:** `ReplacementTypes, replacementName.char`
**Type:** character vector

**Value:** The name of a `Simulink.AliasType` object, where the object exists in the base workspace.
**Default:** `''`

### Recommended Settings

| Application | Setting |
|---|---|
| Debugging | No impact |
| Traceability | A valid character vector |
| Efficiency | No impact |
| Safety precaution | `''` |

## See Also

### Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 9-2
- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- `Simulink.AliasType`

# 10

# Memory Sections Parameters on the Code Generation Pane

# Code Generation: Memory Sections Tab Overview

Insert comments and pragmas into the generated code for data and functions.

## Configuration

This tab appears only if you specify an ERT based system target file (Simulink Coder).

## 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” (Simulink Coder)
- “Control Data and Function Placement in Memory by Inserting Pragmas”

# Package

## Description

Specify a package that contains memory sections you want to apply to model-level functions and internal data.

**Category:** Code Generation

## Settings

Memory section specifications for model-level functions and internal data apply to the top level of the model and to subsystems except atomic subsystems that contain overriding memory section specifications.

**Default:** `---None---`

`---None---`

Suppresses memory sections.

`Simulink`

Applies the built-in Simulink package.

`mpt`

Applies the built-in mpt package.

## Tip

If you have defined packages of your own, click **Refresh package list**. This action adds user-defined packages on your search path to the package list.

## Command-Line Information

**Parameter:** `MemSecPackage`
**Type:** character vector
**Value:** `'--- None ---'` | `'Simulink'` | `'mpt'`
**Default:** `'--- None ---'`

### Recommended Settings

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

## See Also

### Related Examples

- “Model Configuration Parameters: Code Generation” (Simulink Coder)
- “Control Data and Function Placement in Memory by Inserting Pragmas”

# Refresh package list

## Description

Add user-defined packages that are on the search path to list of packages displayed by **Packages**.

**Category:** Code Generation

## Tip

If you have defined packages of your own, click **Refresh package list**. This action adds user-defined packages on your search path to the package list.

# See Also

## Related Examples

- “Model Configuration Parameters: Code Generation” (Simulink Coder)
- “Control Data and Function Placement in Memory by Inserting Pragmas”

# Initialize/Terminate

## Description

Specify whether to apply a memory section to Initialize/Start and Terminate functions.

**Category:** Code Generation

## Settings

Memory section specifications for model-level functions and internal data apply to the top level of the model and to subsystems except atomic subsystems that contain overriding memory section specifications.

**Default:** `Default`

`Default`

Suppresses the use of a memory section for Initialize, Start, and Terminate functions.

*`memory-section-name`*

Applies a memory section to Initialize, Start, and Terminate functions.

## Command-Line Information

**Parameter:** `MemSecFuncInitTerm`
**Type:** character vector
**Value:** `'Default'` | `'MemConst'` | `'MemVolatile'` | `'MemConstVolatile'`
**Default:** `'Default'`

## Recommended Settings

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

# See Also

## Related Examples

- “Model Configuration Parameters: Code Generation” (Simulink Coder)
- “Control Data and Function Placement in Memory by Inserting Pragmas”

# Execution

## Description

Specify whether to apply a memory section to execution functions.

**Category:** Code Generation

## Settings

Memory section specifications for model-level functions and internal data apply to the top level of the model and to subsystems except atomic subsystems that contain overriding memory section specifications.

**Default:** `Default`

`Default`

Suppresses the use of a memory section for Step, Run-time initialization, Derivative, Enable, and Disable functions.

*`memory-section-name`*

Applies a memory section to Step, Run-time initialization, Derivative, Enable, and Disable functions.

## Command-Line Information

**Parameter:** `MemSecFuncExecute`
**Type:** character vector
**Value:** `'Default'` | `'MemConst'` | `'MemVolatile'` | `'MemConstVolatile'`
**Default:** `'Default'`

## Recommended Settings

| Application | Setting |
|---|---|
| Debugging | No impact |
| Traceability | No impact |
| Efficiency | No impact |

| Application | Setting |
| --- | --- |
| Safety precaution | No impact |

# See Also

## Related Examples

- “Model Configuration Parameters: Code Generation” (Simulink Coder)
- “Control Data and Function Placement in Memory by Inserting Pragmas”

# Shared utility

## Description

Specify whether to apply memory sections to shared utility functions.

**Category:** Code Generation

## Settings

Memory section specifications for model-level functions and internal data apply to the top level of the model and to subsystems except atomic subsystems that contain overriding memory section specifications.

**Default:** `Default`

`Default`

Suppresses the use of memory sections for shared utility functions.

*`memory-section-name`*

Applies a memory section to shared utility functions, such as fixed-point functions, lookup table functions, and binary search functions.

## Command-Line Information

**Parameter:** `MemSecFuncSharedUtil`
**Type:** character vector
**Value:** `'Default'` | `'MemConst'` | `'MemVolatile'` | `'MemConstVolatile'`
**Default:** `'Default'`

## Recommended Settings

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

# See Also

## Related Examples

- “Model Configuration Parameters: Code Generation” (Simulink Coder)
- “Control Data and Function Placement in Memory by Inserting Pragmas”

# Constants

## Description

Specify whether to apply a memory section to constants.

**Category:** Code Generation

## Settings

Memory section specifications for model-level functions and internal data apply to the top level of the model and to subsystems except atomic subsystems that contain overriding memory section specifications.

**Default:** `Default`

`Default`

  Suppresses the use of a memory section for constants.

*`memory-section-name`*

  Applies a memory section to constants.

This parameter applies to the generated global data structures that contain:

- Constant parameters
- Constant block I/O

For basic information about the global data structures generated for models, see “Standard Data Structures in the Generated Code” (Simulink Coder).

## Command-Line Information

**Parameter:** `MemSecDataConstants`
**Type:** character vector
**Value:** `'Default'` | `'MemConst'` | `'MemVolatile'` | `'MemConstVolatile'`
**Default:** `'Default'`

## Recommended Settings

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

# See Also

## Related Examples

- "Model Configuration Parameters: Code Generation" (Simulink Coder)
- "Control Data and Function Placement in Memory by Inserting Pragmas"

# Inputs/Outputs

## Description

Specify whether to apply a memory section to root input and output.

**Category:** Code Generation

## Settings

Memory section specifications for model-level functions and internal data apply to the top level of the model and to subsystems except atomic subsystems that contain overriding memory section specifications.

**Default:** `Default`

`Default`

Suppresses the use of a memory section for root-level input and output.

*`memory-section-name`*

Applies a memory section for root-level input and output.

This parameter applies to the generated global data structures that contain:

- Root-level inputs
- Root-level outputs

For basic information about the global data structures generated for models, see “Standard Data Structures in the Generated Code” (Simulink Coder).

## Command-Line Information

**Parameter:** `MemSecDataIO`
**Type:** character vector
**Value:** `'Default'` | `'MemConst'` | `'MemVolatile'` | `'MemConstVolatile'`
**Default:** `'Default'`

### Recommended Settings

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

## See Also

### Related Examples

- “Model Configuration Parameters: Code Generation” (Simulink Coder)
- “Control Data and Function Placement in Memory by Inserting Pragmas”

# Internal data

## Description

Specify whether to apply a memory section to internal data.

**Category:** Code Generation

## Settings

Memory section specifications for model-level functions and internal data apply to the top level of the model and to subsystems except atomic subsystems that contain overriding memory section specifications.

**Default:** `Default`

`Default`

Suppresses the use of a memory section for internal data.

*`memory-section-name`*

Applies a memory section for internal data.

This parameter applies to the generated global data structures that contain:

- Block I/O
- DWork vectors
- Zero-crossings

For basic information about the global data structures generated for models, see "Standard Data Structures in the Generated Code" (Simulink Coder).

## Command-Line Information

**Parameter:** `MemSecDataInternal`
**Type:** character vector
**Value:** `'Default'` | `'MemConst'` | `'MemVolatile'` | `'MemConstVolatile'`
**Default:** `'Default'`

### Recommended Settings

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

## See Also

### Related Examples

- “Model Configuration Parameters: Code Generation” (Simulink Coder)
- “Control Data and Function Placement in Memory by Inserting Pragmas”

# Parameters

## Description

Specify whether to apply a memory section to parameters.

**Category:** Code Generation

## Settings

Memory section specifications for model-level functions and internal data apply to the top level of the model and to subsystems except atomic subsystems that contain overriding memory section specifications.

**Default:** `Default`

`Default`

Suppress the use of a memory section for parameters.

*`memory-section-name`*

Apply memory section for parameters.

This parameter applies to the generated global data structure that contains block parameter data.

For basic information about the global data structures generated for models, see “Standard Data Structures in the Generated Code” (Simulink Coder).

## Command-Line Information

**Parameter:** `MemSecDataParameters`
**Type:** character vector
**Value:** `'Default'` | `'MemConst'` | `'MemVolatile'` | `'MemConstVolatile'`
**Default:** `'Default'`

### Recommended Settings

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

## See Also

### Related Examples

- "Model Configuration Parameters: Code Generation" (Simulink Coder)
- Memory Sections

# Validation results

## Description

Display the results of memory section validation.

**Category:** Code Generation

## Settings

The code generation software checks and reports whether the currently chosen package is on the MATLAB path and that the selected memory sections exist inside the package.

## Recommended Settings

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

# See Also

## Related Examples

- “Model Configuration Parameters: Code Generation” (Simulink Coder)

11

# Code Generation Parameters: Templates

# Model Configuration Parameters: Code Generation Templates

The **Code Generation > Templates** category includes parameters for customizing the organization of your generated code. On the Configuration Parameters dialog box, the following configuration parameters are on the **Code Generation > Templates** pane.

| Parameter | Description |
| --- | --- |
| "Code templates: Source file (*.c) template" on page 11-5 | Specify the code generation template (CGT) file to use when generating a source code file. |
| "Code templates: Header file (*.h) template" on page 11-7 | Specify the code generation template (CGT) file to use when generating a code header file. |
| "Data templates: Source file (*.c) template" on page 11-9 | Specify the code generation template (CGT) file to use when generating a data source file. |
| "Data templates: Header file (*.h) template" on page 11-11 | Specify the code generation template (CGT) file to use when generating a data header file. |
| "File customization template" on page 11-13 | Specify the custom file processing (CFP) template file to use when generating code. |
| "Generate an example main program" on page 11-15 | Control whether to generate an example main program for a model. |
| "Target operating system" on page 11-18 | Specify a target operating system to use when generating model-specific example main program module. |

The following parameters on **Advanced parameters** section are infrequently used and have no other documentation.

| Parameter | Description |
| --- | --- |
| `GenerateFullHeader` | Generate full header including time stamp.<br><br>For GRT targets, this parameter is on the **Code Generation > Interface** pane. |

| Parameter | Description |
|---|---|
| `ERTCustomFileBanners` | If this option is cleared, the configurations for Code and Data templates are ignored. |

# See Also

## More About

- "Model Configuration"

# Code Generation: Templates Tab Overview

Customize the organization of your generated code.

## Configuration

This tab appears only if you specify an ERT based system target file (Simulink Coder).

## To get help on an option

1. Right-click the option text label.
2. From the context menu, select **What's This**.

# See Also

## Related Examples

- "Model Configuration Parameters: Code Generation Templates" on page 11-2

# Code templates: Source file (*.c) template

## Description

Specify the code generation template (CGT) file to use when generating a source code file.

**Category:** Code Generation > Templates

## Settings

**Default:** `ert_code_template.cgt`

You can use a CGT file to define the top-level organization and formatting of generated source code files (`.c` or `.cpp`).

**Note** The CGT file must be located on the MATLAB path.

## Command-Line Information

**Parameter:** `ERTSrcFileBannerTemplate`
**Type:** character vector
**Value:** valid CGT file
**Default:** `'ert_code_template.cgt'`

## Recommended Settings

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

## See Also

### Related Examples

- "Model Configuration Parameters: Code Generation Templates" on page 11-2
- Selecting and Defining Templates
- Custom File Processing

# Code templates: Header file (*.h) template

## Description

Specify the code generation template (CGT) file to use when generating a code header file.

**Category:** Code Generation > Templates

## Settings

**Default:** `ert_code_template.cgt`

You can use a CGT file to define the top-level organization and formatting of generated header files (`.h` ).

---

**Note** The CGT file must be located on the MATLAB path.

---

## Command-Line Information

**Parameter:** `ERTHdrFileBannerTemplate`
**Type:** character vector
**Value:** valid CGT file
**Default:** `'ert_code_template.cgt'`

## Recommended Settings

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

# See Also

## Related Examples

- “Model Configuration Parameters: Code Generation Templates” on page 11-2
- Selecting and Defining Templates
- Custom File Processing

# Data templates: Source file (*.c) template

## Description

Specify the code generation template (CGT) file to use when generating a data source file.

**Category:** Code Generation > Templates

## Settings

**Default:** `ert_code_template.cgt`

You can use a CGT file to define the top-level organization and formatting of generated data source files (`.c` or `.cpp`) that contain definitions of variables of global scope.

**Note** The CGT file must be located on the MATLAB path.

## Command-Line Information

**Parameter:** `ERTDataSrcFileTemplate`
**Type:** character vector
**Value:** valid CGT file
**Default:** `'ert_code_template.cgt'`

## Recommended Settings

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

## See Also

### Related Examples

- “Model Configuration Parameters: Code Generation Templates” on page 11-2
- Selecting and Defining Templates
- Custom File Processing

# Data templates: Header file (*.h) template

## Description

Specify the code generation template (CGT) file to use when generating a data header file.

**Category:** Code Generation > Templates

## Settings

**Default:** `ert_code_template.cgt`

You can use a CGT file to define the top-level organization and formatting of generated data header files (`.h` ) that contain declarations of variables of global scope.

**Note** The CGT file must be located on the MATLAB path.

## Command-Line Information

**Parameter:** `ERTDataHdrFileTemplate`
**Type:** character vector
**Value:** valid CGT file
**Default:** `'ert_code_template.cgt'`

## Recommended Settings

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

# See Also

## Related Examples

- “Model Configuration Parameters: Code Generation Templates” on page 11-2
- Selecting and Defining Templates
- Custom File Processing

# File customization template

## Description

Specify the custom file processing (CFP) template file to use when generating code.

**Category:** Code Generation > Templates

## Settings

**Default:** `'example_file_process.tlc'`

You can use a CFP template file to customize generated code. A CFP template file is a TLC file that organizes types of code (for example, includes, `typedefs`, and functions) into sections. The primary purpose of a CFP template is to assemble code to be generated into buffers, and to call a code template API to emit the buffered code into specified sections of generated source and header files. The CFP template file must be located on the MATLAB path.

## Command-Line Information

**Parameter:** `ERTCustomFileTemplate`
**Type:** character vector
**Value:** valid TLC file
**Default:** `'example_file_process.tlc'`

## Recommended Settings

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

# See Also

## Related Examples

- “Model Configuration Parameters: Code Generation Templates” on page 11-2
- Selecting and Defining Templates
- Custom File Processing

# Generate an example main program

## Description

Control whether to generate an example main program for a model.

**Category:** Code Generation > Templates

## Settings

**Default:** on

On

Generates an example main program, `ert_main.c` (or `.cpp`). The file includes:

- The `main()` function for the generated program
- Task scheduling code that determines how and when block computations execute on each time step of the model

The operation of the main program and the scheduling algorithm employed depend primarily on whether your model is single-rate or multirate, and also on your model's solver mode (`SingleTasking` or `MultiTasking`).

Off

Does not generate an example main program.

---

**Note** The software provides static versions of the main file, *`matlabroot`*`/rtw/c/src/common/rt_main.c` and *`matlabroot`*`/rtw/c/src/common/rt_cppclass_main.cpp`, as a basis for custom modifications. You can use either static main file as a template for developing embedded applications.

---

## Tips

- After you generate and customize the main program, disable this option to prevent regenerating the main module and overwriting your customized version.

- You can use a custom file processing (CFP) template file to override normal main program generation, and generate a main program module customized for your target environment.
- If you disable this option, the code generator produces slightly different rate grouping code to maintain compatibility with an older static main module.

### Dependencies

- This parameter enables **Target operating system**.
- You must enable this parameter and select `VxWorksExample` for **Target operating system** if you use VxWorks[3] library blocks.

### Command-Line Information

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

### Recommended Settings

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

## See Also

### Related Examples

- “Model Configuration Parameters: Code Generation Templates” on page 11-2
- “Deploy Generated Standalone Executable Programs To Target Hardware”

3. VxWorks is a registered trademark of Wind River Systems, Inc.

- Custom File Processing

# Target operating system

## Description

Specify a target operating system to use when generating model-specific example main program module.

**Category:** Code Generation > Templates

## Settings

**Default:** `BareBoardExample`

`BareBoardExample`

Generates a bareboard main program designed to run under control of a real-time clock, without a real-time operating system.

`VxWorksExample`

Generates a fully commented example showing how to deploy the code under the VxWorks real-time operating system.

`NativeThreadsExample`

Generates a fully commented example showing how to deploy the threaded code under the host operating system. This option requires you to configure your model for concurrent execution.

## Dependencies

- This parameter is enabled by **Generate an example main program**.
- This parameter must be the same for top-level and referenced models.

## Command-Line Information

**Parameter:** `TargetOS`
**Type:** character vector
**Value:** `'BareBoardExample'` | `'VxWorksExample'` | `'NativeThreadsExample'`
**Default:** `'BareBoardExample'`

### Recommended Settings

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

## See Also

### Related Examples

- “Model Configuration Parameters: Code Generation Templates” on page 11-2
- “Deploy Generated Standalone Executable Programs To Target Hardware”
- Custom File Processing

# Code Generation Parameters: Verification

# Model Configuration Parameters: Code Generation Verification

The **Code Generation > Verification** category includes code verification and performance analysis parameters for SIL and PIL simulations. These parameters require an Embedded Coder license.

| Parameter | Description |
|---|---|
| "Measure task execution time" on page 12-5 | Measure execution times and generate metrics for tasks in generated code. |
| "Measure function execution times" on page 12-7 | Measure execution times and generate metrics for functions inside generated code. |
| "Workspace variable" on page 12-9 | Specify workspace variable that collects measurements and allows viewing and analysis of execution profiles. |
| "Save options" on page 12-11 | Specify whether to save code profiling measurement and analysis data to base workspace. |
| "Third-party tool" on page 12-13 | Specify a third-party tool for code coverage. |
| "Enable portable word sizes" on page 12-15 | Allow portability across host and target processors that support different word sizes. |
| "Enable source-level debugging for SIL" on page 12-17 | Allow debugging of generated code during a SIL simulation. |

This parameter belongs to the **Advanced parameters** category.

| Parameter | Description |
|---|---|
| "Create block" on page 5-2 | Generate a SIL or PIL block. |

# See Also

## More About

- “Numerical Equivalence Testing”
- “Code Execution Profiling”
- “Code Coverage”
- “Model Configuration”

# Code Generation: Verification Tab Overview

Create SIL block and configure word size portability, code coverage for SIL testing, and code execution profiling

## Configuration

This tab appears only if you specify an ERT–based system target file (Simulink Coder).

## To get help on an option

1. Right-click the option text label.
2. From the context menu, select **What's This**.

# See Also

## Related Examples

- “Model Configuration Parameters: Code Generation Verification” on page 12-2
- “SIL and PIL Simulations”

# Measure task execution time

## Description

Measure execution times and generate metrics for tasks in generated code.

**Category:** Code Generation > Verification

## Settings

**Default:** off

On

During SIL and PIL simulations, collect execution-time measurements for tasks. The software obtains data from instrumentation in the SIL or PIL application.

Off

Do not collect measurements of execution times

## Dependencies

When you use this parameter, you must also specify a workspace variable. The software uses this variable to collect execution-time measurements.

In a model reference hierarchy, the top-model parameter value applies to the whole hierarchy. The software ignores the value of this parameter in referenced models.

## Command-Line Information

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

### Recommended Settings

| Application | Setting |
|---|---|
| Debugging | On |
| Traceability | On |
| Efficiency | Off |
| Safety precaution | No recommendation |

## See Also

### Related Examples

- “Model Configuration Parameters: Code Generation Verification” on page 12-2
- “Code Execution Profiling with SIL and PIL”
- “View and Compare Code Execution Times”
- “Analyze Code Execution Data”

# Measure function execution times

## Description

Measure execution times and generate metrics for functions inside generated code.

**Category:** Code Generation > Verification

## Settings

**Default:** off

On

During SIL and PIL simulations, collect execution-time measurements for functions. The software obtains data from instrumentation inside code generated from atomic subsystems and model reference hierarchies.

Off

Do not collect execution times for functions inside generated code

## Dependencies

To use this parameter, you must also select **Measure task execution time** for the top model of the model reference hierarchy.

For a model in a reference hierarchy, the software does not support simultaneous function execution-time measurement and code coverage.

## Command-Line Information

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

### Recommended Settings

| Application | Setting |
|---|---|
| Debugging | On |
| Traceability | On |
| Efficiency | Off |
| Safety precaution | No recommendation |

## See Also

### Related Examples

- “Model Configuration Parameters: Code Generation Verification” on page 12-2
- “Code Execution Profiling with SIL and PIL”
- “View and Compare Code Execution Times”
- “Analyze Code Execution Data”

# Workspace variable

## Description

Specify workspace variable that collects measurements and allows viewing and analysis of execution profiles.

**Category:** Code Generation > Verification

## Settings

**Default:** `executionProfile`

When you run simulation, software generates specified workspace variable as an `coder.profile.ExecutionTime` object. To view and analyze execution profiles, use methods from the `coder.profile.ExecutionTime` and `coder.profile.ExecutionTimeSection` classes.

## Dependency

You can only specify this parameter if you select the **Measure task execution time** check box. Otherwise the field appears dimmed.

## Command-Line Information

**Parameter:** `CodeExecutionProfileVariable`
**Type:** character vector
**Value:** valid MATLAB variable name
**Default:** none

## Recommended Settings

| Application | Setting |
|---|---|
| Debugging | No impact |
| Traceability | Valid MATLAB variable name |
| Efficiency | No impact |

| Application | Setting |
| --- | --- |
| Safety precaution | No impact |

# See Also

## Related Examples

- “Model Configuration Parameters: Code Generation Verification” on page 12-2
- “Code Execution Profiling with SIL and PIL”
- “View and Compare Code Execution Times”
- “Analyze Code Execution Data”

# Save options

## Description

Specify whether to save code profiling measurement and analysis data to base workspace.

**Category:** Code Generation > Verification

## Settings

**Default:** `Summary data only`

`Summary data only`

Save only code profiling summary data to a `coder.profile.ExecutionTime` object in the base workspace. Use this option to limit the amount of data that the software saves to base workspace. For example, if you are concerned that your computer may not have enough memory to store the time measurements for a long simulation. The software calculates metrics for the code execution report as the simulation proceeds, without saving raw data to memory. To view these metrics, use the `coder.profile.ExecutionTime` `report` method.

Selecting this value disables the streaming of execution times to the Simulation Data Inspector during simulations.

`All data`

Save the code profiling measurement and analysis data to a `coder.profile.ExecutionTime` object in the base workspace. In addition to viewing the code execution report, this option allows you to analyze data using `coder.profile.ExecutionTime` and `coder.profile.ExecutionTimeSection` methods.

## Dependency

You can only specify this parameter if you select the **Measure task execution time** check box. Otherwise the field appears dimmed.

## Command-Line Information

**Parameter:** `CodeProfilingSaveOptions`

**Type:** character vector
**Value:** `'SummaryOnly'` | `'AllData'`
**Default:** `'SummaryOnly'`

### Recommended Settings

| Application | Setting |
| --- | --- |
| Debugging | `All data` |
| Traceability | `All data` |
| Efficiency | `Summary data only` |
| Safety precaution | No impact |

## See Also

### Related Examples

- “Model Configuration Parameters: Code Generation Verification” on page 12-2
- “Code Execution Profiling with SIL and PIL”
- “View and Compare Code Execution Times”
- “Analyze Code Execution Data”

# Third-party tool

## Description

Specify a third-party tool for code coverage.

**Category:** Code Generation > Verification

## Settings

**Default:** `None (use Simulink Coverage)`

`None (use Simulink Coverage)`

No third-party tool specified for code coverage. You can use Simulink Coverage™to analyze code coverage.

`BullseyeCoverage`

Specifies the BullseyeCoverage tool from Bullseye Testing Technology

`LDRA Testbed`

Specifies the LDRA Testbed® tool from LDRA Software Technology

## Dependencies

Code coverage is not supported if the **Create block** configuration parameter is either `SIL` or `PIL`.

If you do not specify a third-party tool, **Configure Coverage** appears dimmed. Otherwise, click **Configure Coverage** to open the Code Coverage Settings dialog box.

## Command-Line Information

**Parameter:** `CoverageTool` field of `CodeCoverageSettings`
**Type:** character vector
**Value:** `'None'` | `'BullseyeCoverage'` | `'LDRA Testbed'`
**Default:** `'None'`

---

**Tip** To access the `CoverageTool` value, type:

```
covSettings = get_param(gcs, 'CodeCoverageSettings');
covSettings.CoverageTool
```

---

## Recommended Settings

| Application | Setting |
|---|---|
| Debugging | `BullseyeCoverage` or `LDRA Testbed` |
| Traceability | `BullseyeCoverage` or `LDRA Testbed` |
| Efficiency | `None` (code coverage off) |
| Safety precaution | No recommendation |

# See Also

## Related Examples

- "Model Configuration Parameters: Code Generation Verification" on page 12-2
- "Configure Code Coverage with Third-Party Tools"
- "Configure Code Coverage Programmatically"

# Enable portable word sizes

## Description

Allow portability across host and target processors that support different word sizes.

You can enable portable word sizes to support SIL testing of your generated code. For a SIL simulation, you use the top-model or Model block SIL simulation mode, or select `SIL` in the **Configuration Parameters** > **Create block** field.

**Category:** Code Generation > Verification

## Settings

**Default:** off

On

Generate conditional processing macros to support compilation of generated code on a processor that supports a different word size than the target processor on which production code is intended to run. This option allows you to use the same generated code for software-in-the-loop (SIL) testing on the host platform and production deployment on the target platform. For example, you can perform SIL testing on a 32-bit host and deploy the code on a 16-bit target.

Off

Does not generate portable code.

## Dependencies

When you use this option, you should select **Test hardware is the same as production hardware** on the **Hardware Implementation** pane.

## Command-Line Information

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

### Recommended Settings

| Application | Setting |
|---|---|
| Debugging | On |
| Traceability | On |
| Efficiency | Off |
| Safety precaution | No impact |

## See Also

### Related Examples

- “Model Configuration Parameters: Code Generation Verification” on page 12-2
- “Configure Hardware Implementation Settings”

# Enable source-level debugging for SIL

## Description

Allow debugging of generated code during a SIL simulation.

**Category:** Code Generation > Verification

## Settings

**Default:** off

☑ On

Source-level debugging is enabled.

☐ Off

Source-level debugging is disabled.

## Command-Line Information

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

## Recommended Settings

| Application | Setting |
|---|---|
| Debugging | On |
| Traceability | On |
| Efficiency | Off |
| Safety precaution | No impact |

## See Also

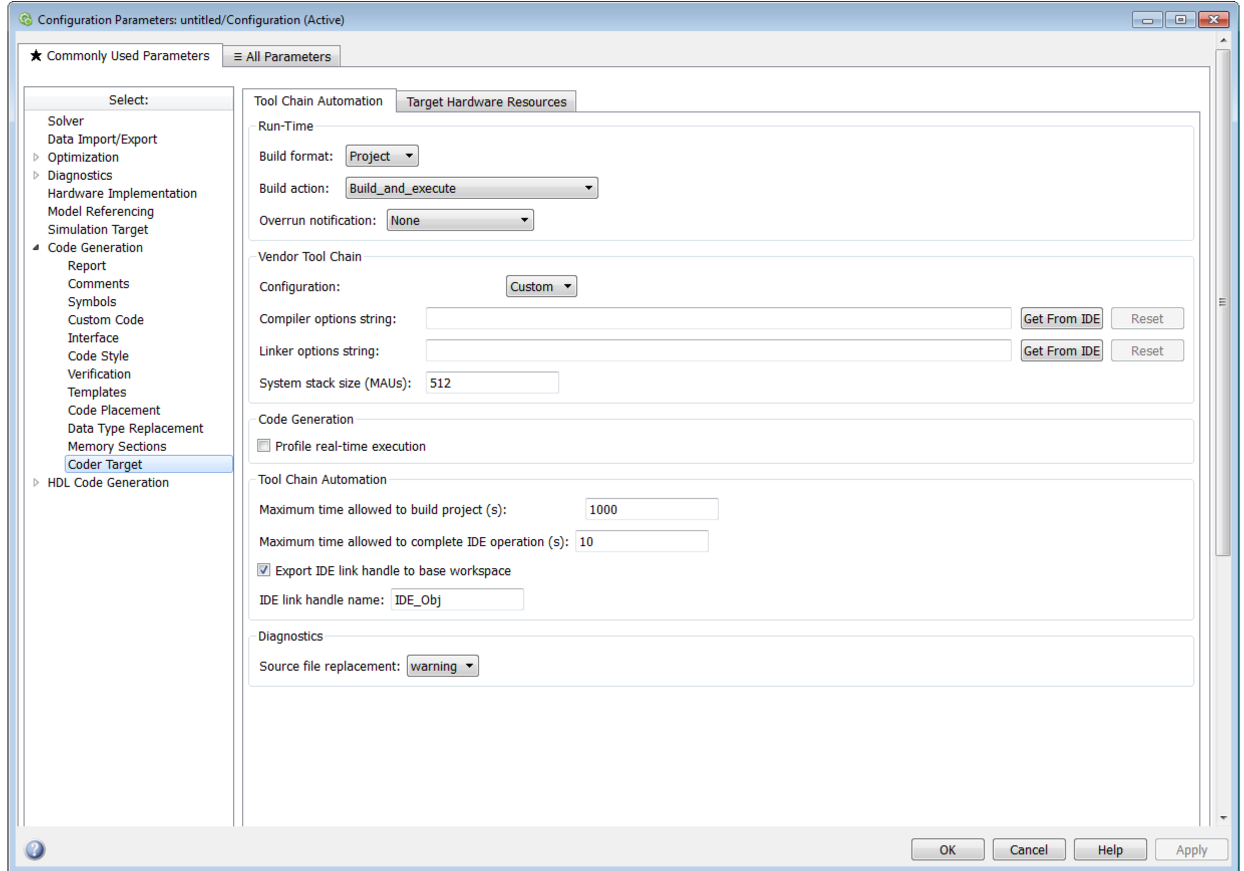
### Related Examples

- "Model Configuration Parameters: Code Generation Verification" on page 12-2
- "Debug Generated Code During SIL Simulation"

# Configuration Parameters


- “Code Generation: Coder Target Pane” on page 13-2
- “Code Generation: Target Hardware Resources Pane” on page 13-27
- “Hardware Implementation Pane: Altera Cyclone V SoC development kit, Arrow SoCKit development board” on page 13-99
- “Hardware Implementation Pane: ARM Cortex-A9 (QEMU)” on page 13-102
- “Hardware Implementation Pane: ARM Cortex-M3 (QEMU)” on page 13-106
- “Hardware Implementation Pane” on page 13-108
- “Hardware Implementation Pane: Freescale FRDM-KL25Z” on page 13-120
- “Hardware Implementation Pane: BeagleBone Black” on page 13-127
- “Hardware Implementation Pane: STMicroelectronics Discovery Boards” on page 13-131
- “Hardware Implementation Pane” on page 13-0
- “Hardware Implementation Pane” on page 13-0
- “Hardware Implementation Pane: Texas Instruments C2000” on page 13-165
- “Hardware Implementation Pane: Texas Instruments Concerto” on page 13-232
- “Hardware Implementation Pane: Xilinx Zynq ZC702/ZC706 Evaluation Kits, ZedBoard” on page 13-289
- “Recommended Settings Summary for Model Configuration Parameters” on page 13-292

# Code Generation: Coder Target Pane




**In this section...**

"Code Generation: Coder Target Pane Overview (previously "IDE Link Tab Overview")" on page 13-3

"Coder Target: Tool Chain Automation Tab Overview" on page 13-4

"Build format" on page 13-5

"Build action" on page 13-6

"Overrun notification" on page 13-9

**In this section...**


"Function name" on page 13-10
"Configuration" on page 13-11
"Compiler options string" on page 13-12
"Linker options string" on page 13-13
"System stack size (MAUs)" on page 13-14
"System heap size (MAUs)" on page 13-16
"Profile real-time execution" on page 13-17
"Profile by" on page 13-18
"Number of profiling samples to collect" on page 13-19
"Maximum time allowed to build project (s)" on page 13-20
"Maximum time allowed to complete IDE operation (s)" on page 13-22
"Export IDE link handle to base workspace" on page 13-23
"IDE link handle name" on page 13-24
"Source file replacement" on page 13-25


## Code Generation: Coder Target Pane Overview (previously "IDE Link Tab Overview")

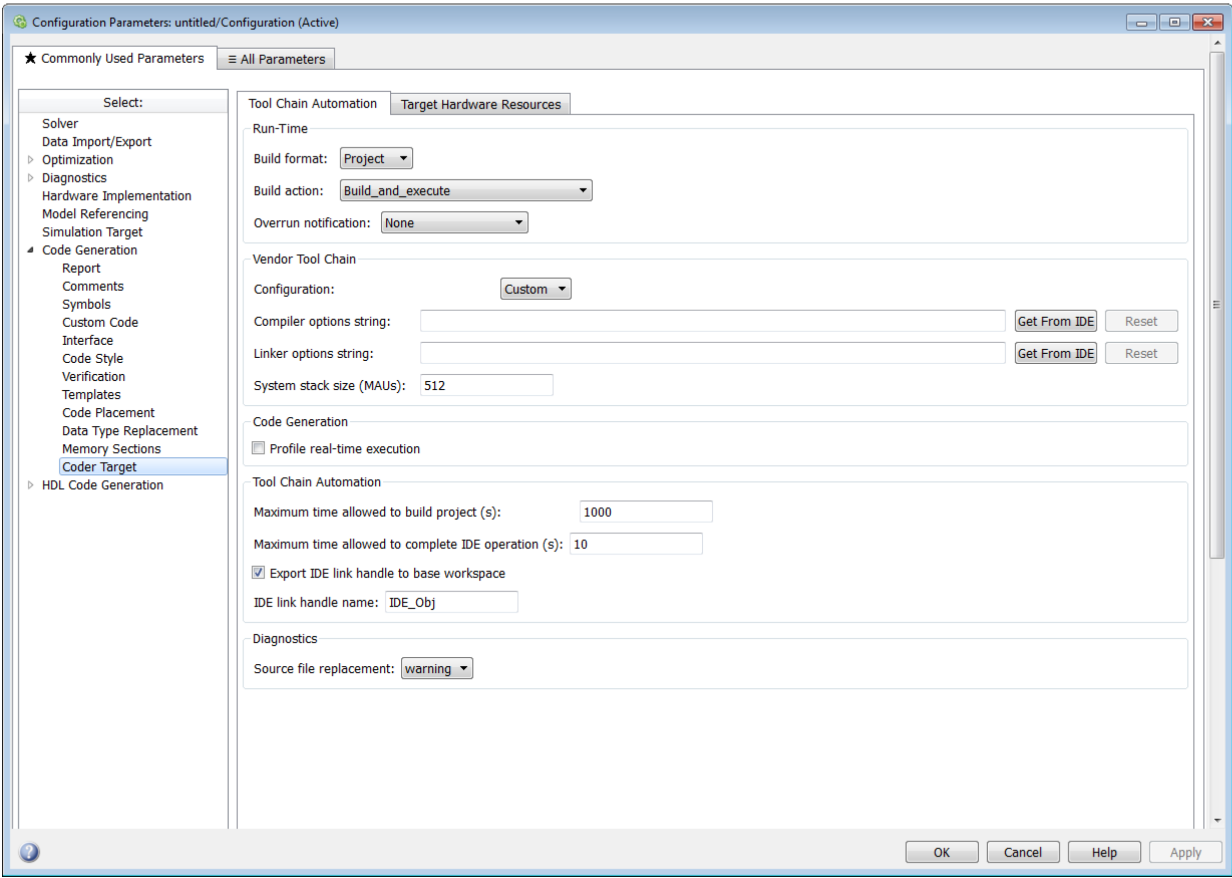
Configure the parameters for:

- Tool Chain Automation — How the code generator interacts with third-party software build toolchains.
- Target Hardware Resources — The IDE toolchain and properties of the physical hardware, such as board, operating system, memory, and peripherals.

### See Also


- "Coder Target: Tool Chain Automation Tab Overview" on page 13-4
- "Coder Target: Target Hardware Resources Tab Overview" on page 13-29

# Coder Target: Tool Chain Automation Tab Overview




The Tool Chain Automation Tab is only visible under the Coder Target pane.

The following table lists the parameters on the Tool Chain Automation Tab.

- “Build format” on page 13-5
- “Build action” on page 13-6
- “Overrun notification” on page 13-9
- “Function name” on page 13-10

- “Configuration” on page 13-11
- “Compiler options string” on page 13-12
- “Linker options string” on page 13-13
- “System stack size (MAUs)” on page 13-14
- “System heap size (MAUs)” on page 13-16
- “Profile real-time execution” on page 13-17
- “Profile by” on page 13-18
- “Number of profiling samples to collect” on page 13-19
- “Maximum time allowed to build project (s)” on page 13-20
- “Maximum time allowed to complete IDE operation (s)” on page 13-22
- “Export IDE link handle to base workspace” on page 13-23
- “IDE link handle name” on page 13-24
- “Source file replacement” on page 13-25


## Build format

Defines how the code generator responds when you press Ctrl+B to build your model.

### Settings

**Default:** `Project`

`Project`

Builds your model as an IDE project.

`Makefile`

Creates a makefile and uses it to build your model.

### Dependencies

Selecting `Makefile` removes the following parameters:

- **Code Generation**
  - **Profile real-time execution**
  - **Profile by**

  - **Number of profiling samples to collect**
- **Link Automation**
  - **Maximum time allowed to build project (s)**
  - **Maximum time allowed to complete IDE operation (s)**
  - **Export IDE link handle to base workspace**
  - **IDE link handle name**

#### Command-Line Information

**Parameter:** `buildFormat`
**Type:** character vector
**Value:** `'Project'` | `'Makefile'`
**Default:** `'Build_and_execute'`

#### Recommended Settings

| Application | Setting |
|---|---|
| Debugging | `Project` |
| Traceability | `Project` |
| Efficiency | No impact |
| Safety precaution | No impact |

#### See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

## Build action

Defines how the code generator responds when you press Ctrl+B to build your model.

#### Settings

**Default:** `Build_and_execute`

If you set **Build format** to `Project`, select one of the following options:

`Build_and_execute`

Builds your model, generates code from the model, and then compiles and links the code. After the software links your compiled code, the build process downloads and runs the executable on the processor.

`Create_project`

Directs the code generator to create a new project in the IDE. The command line equivalent for this setting is `Create`.

`Archive_library`

Invokes the IDE Archiver to build and compile your project, but It does not run the linker to create an executable project. Instead, the result is a library project.

`Build`

Builds a project from your model. Compiles and links the code. Does not download and run the executable on the processor.

`Create_processor_in_the_loop_project`

Directs the code generator to create PIL algorithm object code as part of the project build.

If you set **Build format** to `Makefile`, select one of the following options:

`Create_makefile`

Creates a makefile. For example, ".mk". The command line equivalent for this setting is `Create`.

`Archive_library`

Creates a makefile and an archive library. For example, ".a" or ".lib".

`Build`

Creates a makefile and an executable. For example, ".exe".

`Build_and_execute`

Creates a makefile and an executable. Then it evaluates the execute instruction under the **Execute** tab in the current XMakefile configuration.

### Dependencies

Selecting `Archive_library` removes the following parameters:

- **Overrun notification**

- **Function name**
- **Profile real-time execution**
- **Number of profiling samples to collect**
- **Linker options string**
- **Get from IDE**
- **Reset**
- **Export IDE link handle to base workspace**

Selecting `Create_processor_in_the_loop_project` removes the following parameters:

- **Overrun notification**
- **Function name**
- **Profile real-time execution**
- **Number of profiling samples to collect**
- **Linker options string**
- **Get from IDE**
- **Reset**
- **Export IDE link handle to base workspace** with the option set to export the handle

#### Command-Line Information

**Parameter:** `buildAction`
**Type:** character vector
**Value:** `'Build'` | `'Build_and_execute'` | `'Create'` | `'Archive_library'` | `'Create_processor_in_the_loop_project'`
**Default:** `'Build_and_execute'`

#### Recommended Settings

| Application | Setting |
| --- | --- |
| Debugging | `Build_and_execute` |
| Traceability | `Archive_library` |
| Efficiency | No impact |
| Safety precaution | No impact |

#### See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

For more information about PIL and its uses, refer to the “Verifying Generated Code via Processor-in-the-Loop” topic.

## Overrun notification

Specifies how your program responds to overrun conditions during execution.

#### Settings

**Default:** `None`

`None`

Your program does not notify you when it encounters an overrun condition.

`Print_message`

Your program prints a message to standard output when it encounters an overrun condition.

`Call_custom_function`

When your program encounters an overrun condition, it executes a function that you specify in **Function name**.

#### Tips

- The definition of the standard output depends on your configuration.

#### Dependencies

Selecting `Call_custom_function` enables the **Function name** parameter.

Setting this parameter to `Call_custom_function` enables the **Function name** parameter.

#### Command-Line Information

**Parameter:** `overrunNotificationMethod`
**Type:** character vector
**Value:** `'None'` | `'Print_message'` | `'Call_custom_function'`
**Default:** `'None'`

#### Recommended Settings

| Application | Setting |
| --- | --- |
| Debugging | `Print_message` or `Call_custom_function` |
| Traceability | `Print_message` |
| Efficiency | `None` |
| Safety precaution | No impact |

#### See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

## Function name

Specifies the name of a custom function your code runs when it encounters an overrun condition during execution.

#### Settings

**No Default**

#### Dependencies

This parameter is enabled by setting **Overrun notification** to `Call_custom_function`.

#### Command-Line Information

**Parameter:** `overrunNotificationFcn`
**Type:** character vector
**Value:** no default
**Default:** no default

#### Recommended Settings

| Application | Setting |
| --- | --- |
| Debugging | String |
| Traceability | String |
| Efficiency | No impact |

| Application | Setting |
|---|---|
| Safety precaution | No impact |

#### See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

## Configuration

Sets the Configuration for building your project from the model.

#### Settings

**Default:** `Custom`

`Custom`

Lets the user apply a specialized combination of build and optimization settings.

`Custom` applies the same settings as the Release project configuration in IDE, except:

- The compiler options do not use optimizations.
- The memory configuration specifies a memory model that uses `Far Aggregate` for data and `Far` for functions.

`Debug`

Applies the `Debug` Configuration defined by the IDE to the generated project and code.

`Release`

Applies the `Release` project configuration defined by the IDE to the generated project and code.

#### Dependencies

- Selecting `Custom` disables the reset options for **Compiler options string** and **Linker options string**.
- Selecting `Release` sets the **Compiler options string** to the settings defined by the IDE.
- Selecting `Debug` sets the **Compiler options string** to the settings defined by the IDE.

.

**Command-Line Information**
**Parameter:** `projectOptions`
**Type:** character vector
**Value:** `'Custom'` | `'Debug'` | `'Release'`
**Default:** `'Custom'`

### Recommended Settings

| Application | Setting |
|---|---|
| Debugging | `Custom` or `Debug` |
| Traceability | `Custom`, `Debug`, `Release` |
| Efficiency | `Release` |
| Safety precaution | No impact |

### See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

## Compiler options string

To determine the degree of optimization provided by the optimizing compiler, enter the optimization level to apply to files in your project. For details about the compiler options, refer to your IDE documentation. When you create new projects, the code generator does not set optimization flags.

With Texas Instruments Code Composer Studio v3.3 and Analog Devices VisualDSP++, the user interface displays **Get From IDE** and **Reset** buttons next to this parameter. If you have an active project open in the IDE, you can click **Get From IDE** to import the compiler option setting from the current project in the IDE. To reset the compiler option to the default value, click **Reset**.

### Settings

**Default:** No default

#### Tips

- Use spaces between options.
- Verify that the options are valid. The software does not validate the option string.
- Setting **Configuration** to `Custom` applies the `Custom` compiler options defined by the code generator. `Custom` does not use optimizations.
- Setting **Configuration** to `Debug` applies the debug settings defined by the IDE.
- Setting **Configuration** to `Release` applies the release settings defined by the IDE.

#### Command-Line Information

**Parameter:** `compilerOptionsStr`
**Type:** character vector
**Value:** `'Custom'` | `'Debug'` | `'Release'`
**Default:** `'Custom'`

#### Recommended Settings

| Application | Setting |
|---|---|
| Debugging | `Custom` |
| Traceability | `Custom` |
| Efficiency | No impact |
| Safety precaution | No impact |

#### See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

## Linker options string

To specify the options provided by the linker during link time, you enter the linker options as a string. For details about the linker options, refer to your IDE documentation. When you create new projects, the code generator does not set linker options.

With Texas Instruments Code Composer Studio v3.3 and Analog Devices VisualDSP++, the user interface displays **Get From IDE** and **Reset** buttons next to this parameter. If you have an active project open in the IDE, you can click **Get From IDE** to import the linker options string from the current project in the IDE. To reset the linker options to the default value of no options, click **Reset**.

#### Settings

**Default**: No default

#### Tips

- Use spaces between options.
- Verify that the options are valid. The software does not validate the options string.

#### Dependencies

Setting **Build action** to `Archive_library` removes this parameter.

#### Command-Line Information

**Parameter:** `linkerOptionsStr`
**Type:** character vector
**Value:** valid linker option
**Default:** none

#### Recommended Settings

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

#### See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

## System stack size (MAUs)

Enter the amount of memory that is available for allocating stack data. Block output buffers are placed on the stack until the stack memory is fully allocated. After that, the output buffers go in global memory.

This parameter is used in targets to allocate the stack size for the generated application. For example, with embedded processors that are not running an operating system, this parameter determines the total stack space that can be used for the application. For

operating systems such as Linux or VxWorks, this value specifies the stack space allocated per thread.

This parameter also affects the “Maximum stack size (bytes)” (Simulink Coder) parameter, located in the Optimization pane.

### Settings

**Default:** 8192

**Minimum:** 0

**Maximum:** Available memory

- Enter the stack size in minimum addressable units (MAUs). An MAU is typically 1 byte, but its size can vary by target processor.
- The software does not verify the value you entered is valid.

### Dependencies

Setting **Build action** to `Archive_library` removes this parameter.

When you set the **System target file** parameter on the **Code Generation** pane to `idelink_ert.tlc` or `idelink_grt.tlc`, the software sets the **Maximum stack size** parameter on the **Optimization** pane to `Inherit from target` and makes it non-editable. In that case, the **Maximum stack size** parameter compares the value of (**System stack size**/2) with 200,000 bytes and uses the smaller of the two values.

### Command-Line Information

**Parameter:** `systemStackSize`
**Type:** int
**Default:** 8192

### Recommended Settings

| Application | Setting |
| --- | --- |
| Debugging | int |
| Traceability | int |
| Efficiency | int |
| Safety precaution | No impact |

### See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

## System heap size (MAUs)

Set the default heap size that the target processor reserves for dynamic memory allocation.

The target processor uses this heap for functions like printf() and system services code.

The following IDEs use this parameter:

- Analog Devices VisualDSP++
- Wind River Diab/GCC (makefile generation only)

### Settings

**Default:** 8192

**Minimum:** 0

**Maximum:** Available memory

- Enter the heap size in minimum addressable units (MAUs). An MAU is typically 1 byte, but its size can vary by target processor.
- The software does not verify that your size is valid. Be sure that you enter an acceptable value.

### Dependencies

Setting **Build action** to `Archive_library` removes this parameter.

### Command-Line Information

**Parameter:** `systemHeapSize`
**Type:** int
**Default:** `8192`

### Recommended Settings

| Application | Setting |
|---|---|
| Debugging | int |
| Traceability | int |
| Efficiency | int |
| Safety precaution | No impact |

### See Also

For more information, refer to the "Code Generation Pane: Coder Target" topic.

## Profile real-time execution

Enables real-time execution profiling in the generated code by adding instrumentation for task functions or atomic subsystems.

### Settings

**Default:** Off

On

Adds instrumentation to the generated code to support execution profiling and generate the profiling report.

Off

Does not instrument the generated code to produce the profile report.

### Dependencies

This parameter adds **Number of profiling samples to collect** and **Profile by**.

Selecting this parameter enables **Export IDE link handle to base workspace** and makes it non-editable, since the code generator must create a handle.

Setting **Build action** to `Archive_library` or `Create_processor_in_the_loop project` removes this parameter.

**Command-Line Information**
**Parameter:** `ProfileGenCode`
**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

For more information, refer to the “Code Generation Pane: Coder Target” topic.

For more information about using profiling, refer to the “profile” and “Profiling Code Execution in Real-Time” topics..

## Profile by

Defines which execution profiling technique to use.

### Settings

**Default:** `Task`

`Task`

Profiles model execution by the tasks in the model.

`Atomic subsystem`

Profiles model execution by the atomic subsystems in the model.

### Dependencies

Selecting **Real-time execution profiling** enables this parameter.

**Command-Line Information**
**Parameter:** `profileBy`
**Type:** character vector
**Value:** `Task` | `Atomic subsystem`
**Default:** `Task`

**Recommended Settings**

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

**See Also**

For more information, refer to the “Code Generation Pane: Coder Target” topic.

For more information about PIL and its uses, refer to the “Verifying Generated Code via Processor-in-the-Loop” topic.

For more information about using profiling, refer to the “profile” and “Profiling Code Execution in Real-Time” topics.

## Number of profiling samples to collect

Specify the size of the buffer that holds the profiling samples. Enter a value that is 2 times the number of profiling samples.

Each task or subsystem execution instance represents one profiling sample. Each sample requires two memory locations, one for the start time and one for the end time. Consequently, the size of the buffer is twice the number of samples.

Sample collection begins with the start of code execution and ends when the buffer is full.

The profiling data is held in a statically sited buffer on the target processor.

**Settings**

**Default:** 100

**Minimum:** 2

**Maximum:** Buffer capacity

### Tips

- Data collection stops when the buffer is full, but the application and processor continue running.
- Real-time task execution profiling works with hardware only. Simulators do not support the profiling feature.

### Dependencies

This parameter is enabled by **Profile real-time execution**.

### Command-Line Information

**Parameter:**`ProfileNumSamples`
**Type:** int
**Value:** Positive integer
**Default:** `100`

### Recommended Settings

| Application | Setting |
|---|---|
| Debugging | 100 |
| Traceability | No impact |
| Efficiency | No impact |
| Safety precaution | No impact |

### See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

## Maximum time allowed to build project (s)

Specifies how long, in seconds, the software waits for the project build process to return a completion message.

### Settings

**Default:** 1000

**Minimum:** 1

**Maximum:** No limit

### Tips

- The build process continues even if MATLAB does not receive the completion message in the allotted time.
- This timeout value does not depend on the global timeout value in a `IDE_Obj` object or the **Maximum time allowed to complete IDE operation** timeout value.

### Dependency

This parameter is disabled when you set **Build action** to `Create_project`.

### Command-Line Information

**Parameter:**`ideObjBuildTimeout`
**Type:** int
**Value:** Integer greater than 0
**Default:** `100`

### Recommended Settings

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

### See Also

For more information, refer to the "Code Generation Pane: Coder Target" topic.

## Maximum time allowed to complete IDE operation (s)

specifies how long, in seconds, the software waits for IDE functions, such as `read` or `write`, to return completion messages.

### Settings

**Default:** 10

**Minimum:** 1

**Maximum:** No limit

### Tips

- The IDE operation continues even if MATLAB does not receive the message in the allotted time.
- This timeout value does not depend on the global timeout value in a `IDE_Obj` object or the **Maximum time allowed to build project (s)** timeout value

### Command-Line Information

**Parameter:** `'ideObjTimeout'`
**Type:** int
**Value:**
**Default:** `10`

### Recommended Settings

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

### See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

## Export IDE link handle to base workspace

Directs the software to export the `IDE_Obj` object to your MATLAB workspace.

#### Settings

**Default:** On

On

Directs the build process to export the `IDE_Obj` object created to your MATLAB workspace. The new object appears in the workspace browser. Selecting this option enables the **IDE link handle name** option.

Off

prevents the build process from exporting the `IDE_Obj` object to your MATLAB software workspace.

#### Dependency

Selecting **Profile real-time execution** enables **Export IDE link handle to base workspace** and makes it non-editable, since the code generator must create a handle.

Selecting **Export IDE link handle to base workspace** enables **IDE link handle name**.

#### Command-Line Information
**Parameter:** `exportIDEObj`
**Type:** character vector
**Value:** `'on'` | `'off'`
**Default:** `'on'`

#### Recommended Settings

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

### See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

## IDE link handle name

specifies the name of the `IDE_Obj` object that the build process creates.

### Settings

**Default:** `IDE_Obj`

- Enter a valid C variable name, without spaces.
- The name you use here appears in the MATLAB workspace browser to identify the `IDE_Obj` object.
- The handle name is case sensitive.

### Dependency

This parameter is enabled by **Export IDE link handle to base workspace**.

### Command-Line Information
**Parameter:** `ideObjName`
**Type:** character vector
**Value:**
**Default:** `IDE_Obj`

### Recommended Settings

| Application | Setting |
| --- | --- |
| Debugging | Enter a valid C program variable name, without spaces |
| Traceability | No impact |
| Efficiency | No impact |
| Safety precaution | No impact |

### See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

## Source file replacement

Selects the diagnostic action to take if the code generator detects conflicts that you are replacing source code with custom code.

### Settings

**Default:** `warn`

`none`

Does not generate warnings or errors when it finds conflicts.

`warning`

Displays a warning.

`error`

Terminates the build process and displays an error message that identifies which file has the problem and suggests how to resolve it.

### Tips

- The build operation continues if you select `warning` and the software detects custom code replacement. You see warning messages as the build progresses.
- Select `error` the first time you build your project after you specify custom code to use. The error messages can help you diagnose problems with your custom code replacement files.
- Select `none` when you do not want to see multiple messages during your build.
- The messages apply to code generator **Custom Code** replacement options as well.

### Command-Line Information

**Parameter:** `DiagnosticActions`
**Type:** character vector
**Value:** `none` | `warning` | `error`
**Default:** `warning`

### Recommended Settings

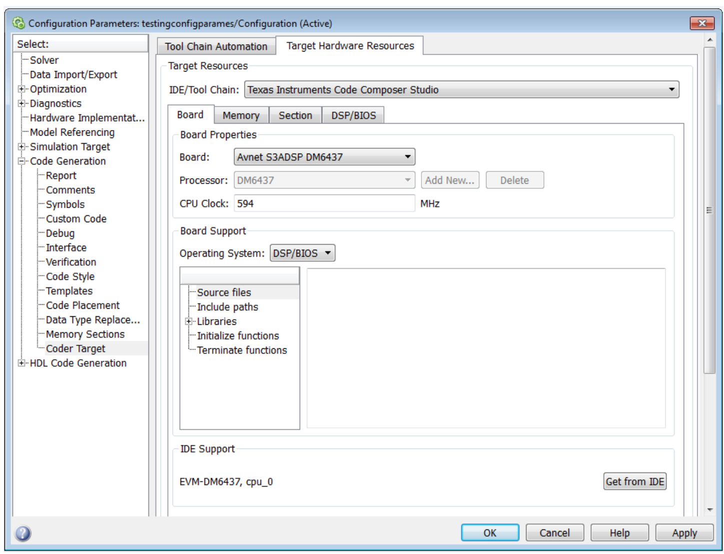
| Application | Setting |
| --- | --- |
| Debugging | `error` |

| Application | Setting |
| --- | --- |
| Traceability | `error` |
| Efficiency | `warning` |
| Safety precaution | `error` |

### See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

# Code Generation: Target Hardware Resources Pane




**In this section...**

"Code Generation: Coder Target Pane Overview" on page 13-28
"(Target Hardware Resources)" on page 13-28
"Coder Target: Target Hardware Resources Tab Overview" on page 13-29
"IDE/Tool Chain" on page 13-29
"Target Hardware Resources: Board Tab" on page 13-30
"Target Hardware Resources: Memory Tab" on page 13-33
"Target Hardware Resources: Section Tab" on page 13-36
"Target Hardware Resources: DSP/BIOS Tab" on page 13-38
"Target Hardware Resources: Peripherals Tab" on page 13-41
"C28x-Clocking" on page 13-44
"C28x-ADC" on page 13-47
"C28-COMP" on page 13-50
"C28x-eCAN_A, C28x-eCAN_B" on page 13-51

| In this section... |
|---|
| “C28x-eCAP” on page 13-54 |
| “C28x-ePWM” on page 13-57 |
| “C28x-I2C” on page 13-60 |
| “C28x-SCI_A, C28x-SCI_B, C28x-SCI_C” on page 13-66 |
| “C28x-SPI_A, C28x-SPI_B, C28x-SPI_C, C28x-SPI_D” on page 13-69 |
| “C28x-eQEP” on page 13-72 |
| “C28x-Watchdog” on page 13-74 |
| “C28x-GPIO” on page 13-76 |
| “C28x-DMA_ch[#]” on page 13-81 |
| “C28x-LIN” on page 13-90 |
| “Add Processor Dialog Box” on page 13-96 |
| “Target Hardware Resources Tab: Linux, VxWorks, or Windows” on page 13-97 |

## Code Generation: Coder Target Pane Overview

Control options for third-party software build toolchains and processors.

### Configuration

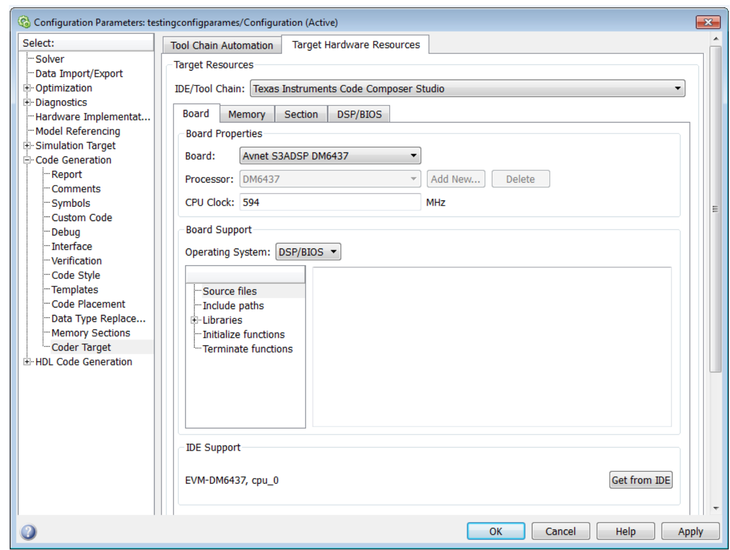
This tab appears only if you specify an idelink_ert or idelink_grt system target file (Simulink Coder).

## (Target Hardware Resources)

Configure the parameters for:

- Tool Chain Automation — How the code generator interacts with third-party software build toolchains.
- Target Hardware Resources — The IDE toolchain and properties of the physical hardware, such as board, operating system, memory, and peripherals.

# Coder Target: Target Hardware Resources Tab Overview



The Target Hardware Resources tab is only visible under the Coder Target pane.

The following table lists the parameters and tabs on the Target Hardware Resources tab.


- “IDE/Tool Chain” on page 13-29
- “Target Hardware Resources: Board Tab” on page 13-30
- “Target Hardware Resources: Memory Tab” on page 13-33
- “Target Hardware Resources: Section Tab” on page 13-36
- “Target Hardware Resources: Peripherals Tab” on page 13-41


## IDE/Tool Chain

Select the IDE or software build tool chain you are using from the list of options. This action applies parameter values for a specific IDE or tool chain.

Located on the Target Hardware Resources tab.

#### Settings

The name of a specific toolchain

The name of a specific toolchain appears after you install an Embedded Coder support package.

Selecting the toolchain configures the Target Hardware Resources parameters to work with a specific toolchain.

To install a support package, select `Get more...` or enter `supportPackageInstaller` in the MATLAB Command Window.

`Get more...`

Launches the Support Package Installer. For more information, see `supportPackageInstaller`.

#### See Also

`supportPackageInstaller`

## Target Hardware Resources: Board Tab

The following options appear on the **Board** pane, which has separate panels for **Board Properties**, **Board Support**, and **IDE Support** labels.

**Board**

Select your target board from the list of options. Selecting a specific board sets the value for the **Processor** parameter. If you select a custom board, also set the **Processor** parameter.

**Processor**

The Board and Processor settings apply default values to many of the parameters, such as those under the **Memory** and **Section** tabs.

If the code generator supports an operating system for the processor, it enables the **Operating system** option.

**Note** Selecting or reselecting a processor resets the solver and some processor-specific parameters to their default values.

### Add New

Clicking **Add new** opens a new dialog box where you specify configuration information for a processor that is not on the Processor list.

For details about the New Processor dialog box, refer to “Add Processor Dialog Box” on page 13-96.

### Delete

Clicking **Delete**, removes a processor that you added to the **Processor** list. You cannot delete the standard processors.

### CPU Clock

Enter the actual clock rate the board uses. This action does not change the rate on the board. Rather, the code generation process requires this information to produce code that runs on the hardware. Setting this value incorrectly causes timing and profiling errors when you run the code on the hardware.

The timer uses the value of **CPU clock** to calculate the time for each interrupt. For example, a model with a sine wave generator block running at 1 kHz uses timer interrupts to generate sine wave samples at the specified rate. For example, using 100 MHz, the timer calculates the sine generator interrupt period as follows:

- Sine block rate = 1 kHz, or 0.001 s/sample
- CPU clock rate = 100 MHz, or 0.000000001 s/sample

To create sine block interrupts at 0.001 s/sample requires:

100,000,000/1000 = 1 Sine block interrupt per 100,000 clock ticks

### Board Support

Select the following parameters and edit their values in the text box on the right:

- `Source files` — Enter the full paths to source code files.
- `Include paths` — Add paths to include files.
- `Libraries` — Identify specific libraries for the processor. Required libraries appear on the list by default. To add more libraries, entering the full path to the library with the library file in the text area.
- `Initialize functions` — If your project requires an initialize function, enter it in this field. By default, this parameter is empty.
- `Terminate functions` — Enter a function to run when a program terminates. The default setting is not to include a specific termination function.

**Note** Invalid or incorrect entries in these fields can cause errors during code generation. When you enter a file path, library, or function, the block does not verify that the path or function exists or is valid.

When entering a path to a file, library, or other custom code, use the following text in the path to refer to the IDE installation folder.

```
$(Install_dir)
```

Enter new paths or files (custom code items) one entry per line. Include the full path to the file for libraries and source code. **Support** options do not support functions that use return arguments or values. These parameters accept only functions of type `void fname void` as valid as entries.

You can also set up environment variables to use as folder path tokens. For example, if you set up an environment called `USER_VAR`, you can use it as a token when you define a path in Coder Target > Target Hardware Resources. For example: `$(USER_VAR)\myinstal\foo.c`.

**Operating System**

Select an operating system or RTOS for your target. If your target platform supports an operating system, the software enables the **Operating system** parameter. Otherwise, the software disables this option.

**Get from IDE**

This button only appears when you are using Texas Instruments Code Composer Studio 3.3 IDE or Analog Devices VisualDSP++ IDE:

- With Texas Instruments Code Composer Studio 3.3 IDE, the **Get from IDE** button imports the current **Board Name** and **Processor Name** from the IDE.
- With Analog Devices VisualDSP++ IDE, the **Get from IDE** button imports the current **Session Name** and **Processor Name** from the IDE.

Use the **Get from IDE** button to update the Coder Target > Target Hardware Resources, the IDE, and the hardware board so they refer to the same processor. Otherwise, during code generation, the software generates a warning similar to the following message:

```
Target Hardware Resources tab specifies that the board named
 '<boardname1>' will be used to run generated code.
However, since only board named '<boardname2>' is found
in your system, that board will be used.
```

**Board Name**

**Board Name** appears after you click **Get from IDE**. Select the board you are using. Match **Board Name** with the **Board** option near the top of the **Board** pane.

**Processor Name**

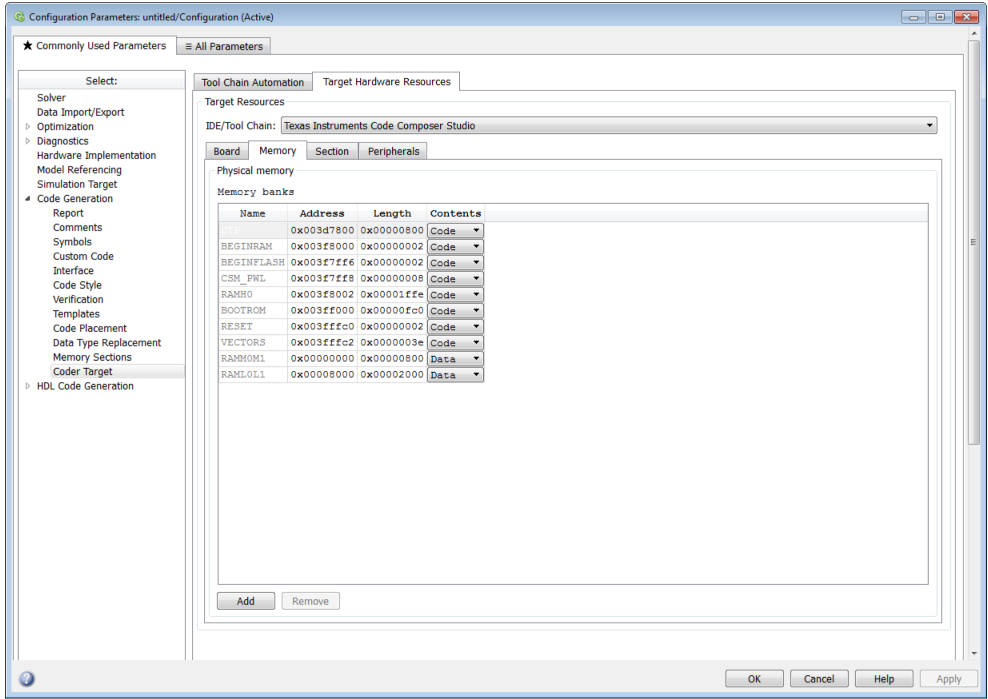
**Processor Name** appears after you click **Get from IDE**. If the board you selected in **Board Name** has multiple processors, select the processor you are using. Match**Processor Name** with the **Processor** option near the top of the **Board** pane.

---

**Note** Click **Apply** to update the board and processor description under **IDE Support**.

---

## Target Hardware Resources: Memory Tab




After selecting a board, specify the layout of the physical memory on your processor and board to determine how to use it for your program.

The **Memory** pane contains memory options for:

- **Physical Memory** — Specifies the processor and board memory map
- **Cache Configuration** — Select a cache configuration where available, such as L2 cache, and select one of the corresponding configuration options, such as `32 kb`.

For more information about memory segments and memory allocation, consult the reference documentation for the IDE or processor.

The **Physical Memory** table shows the memory segments or memory banks available on the board and processor. By default, Target Hardware Resources tab show the memory segments found on the selected processor. In addition, the **Memory** pane on Target Hardware Resources tab shows the memory segments available on the board, but external to the processor. Target Hardware Resources tab set default starting addresses, lengths, and contents of the default memory segments. The default memory segments for each processor and board differ.

Click **Add** to add physical memory segments to the **Memory banks** table.

After you add the segment, you can configure the starting address, length, and contents for the new segment.

**Name**

To change the memory segment name, click the name, and then type the new name. Names are case sensitive. `NewSegment` is not the same as `newsegment` or `newSegment`.

**Note** You cannot rename default processor memory segments (name in gray text).

**Address**

**Address** reports the starting address for the memory segment showing in **Name**. Address entries appear in hexadecimal format and are limited only by the board or processor memory.

**Length**

From the starting address, **Length** sets the length of the memory allocated to the segment in **Name**. As in all memory entries, specify the length in hexadecimal format, in minimum addressable data units (MADUs).

For the C6000 processor family, the MADU requires inputs of 8 bytes, one word.

**Contents**

Configure the segment to store `Code`, `Data`, or `Code & Data`. Changing processors changes the options for each segment.

You can add and use as many segments of each type as you need, within the limits of the memory on your processor. Every processor must have a segment that holds code, and a segment that holds data.

**Add**

Click **Add** to add a new memory segment to the processor memory map. When you click **Add**, a new segment name appears, for example `NEWMEM1`, in **Name** and on the **Memory banks** table. In **Name**, change the temporary name `NEWMEM1` by entering the new segment name. Entering the new name, or clicking **Apply**, updates the temporary name on the table to the name you enter.

**Remove**

This option lets you remove a memory segment from the memory map. Select the segment to remove on the **Memory banks** table, and click **Remove** to delete the segment.

**Cache (Configuration)**

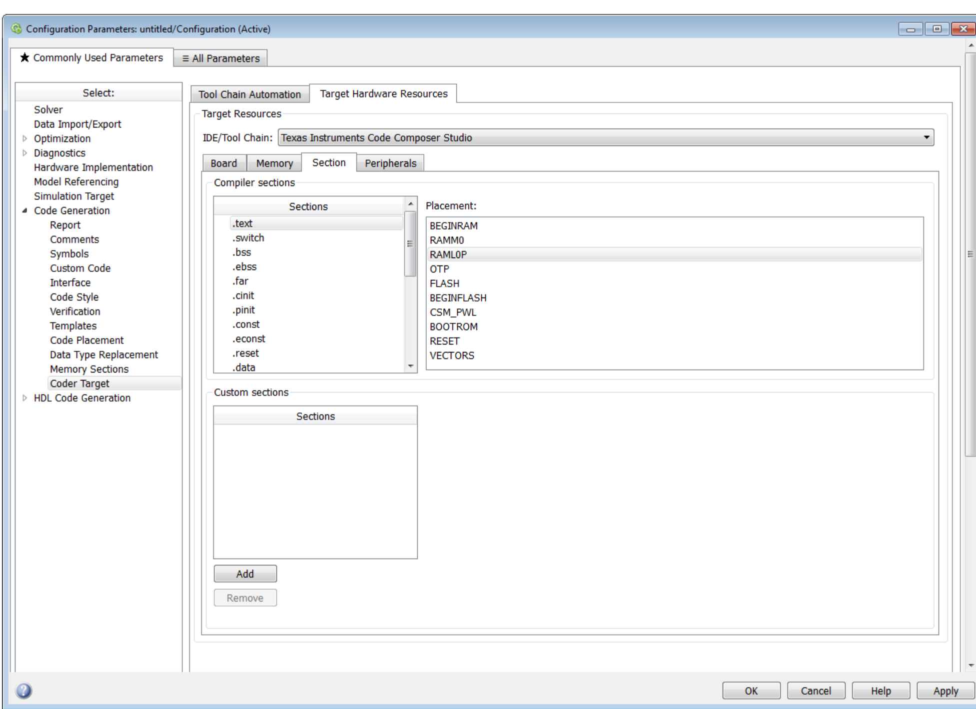
When the **Processor** on the Board pane supports a cache memory structure, the dialog box displays a table of **Cache** parameters. You can use this table to configure the cache as SRAM and partial cache. Both the data memory and the program share this second-level memory.

If your processor supports the two-level memory scheme, this option enables the L2 cache on the processor.

Some processors support code base memory organization. For example, you can configure part of internal memory as code.

Cache level lets you select one of the available cache levels to configure by selecting one of its configurations. For example, you can select L2 cache level, and choose one of its configurations, such as `32 kb`.

## Target Hardware Resources: Section Tab




Options on this pane specify where program sections appear in memory. Program sections differ from memory segments—sections comprise portions of the executable code stored in contiguous memory locations. Commonly used sections include `.text`, `.bss`, `.data`, and `.stack`. Some sections relate to the compiler, and some can be custom sections.

For more information about program sections and objects, refer to the online help for your IDE.

Within the Section pane, you configure the allocation of sections for **Compiler** and **Custom** needs.

This table provides brief definitions of the kinds of sections in the **Compiler sections** and **Custom sections** lists in the pane. all sections do not appear on all lists.

| String | Section List | Description of the Section Contents |
|---|---|---|
| `.bss` | Compiler | Static and global C variables in the code |
| `.cinit` | Compiler | Tables for initializing global and static variables and constants |
| `.cio` | Compiler | Standard I/O buffer for C programs |
| `.const` | Compiler | Data defined with the C qualifier and string constants |
| `.data` | Compiler | Program data for execution |
| `.far` | Compiler | Variables, both static and global, defined as far variables |
| `.pinit` | Compiler | Load allocation of the table of global object constructors section |
| `.stack` | Compiler | The global stack |
| `.switch` | Compiler | Jump tables for switch statements in the executable code |
| `.sysmem` | Compiler | Dynamically allocated object in the code containing the heap |
| `.text` | Compiler | Load allocation for the literal strings, executable code, and compiler generated constants |

You can learn more about memory sections and objects in the online help for your IDE.

**Default Sections**

When you highlight a section on the list, **Description** show a brief description of the section. Also, **Placement** shows you the memory allocation of the section.

**Description**

Provides a brief explanation of the contents of the selected entry on the **Compiler sections** list.

**Placement**

Shows the allocation of the selected **Compiler sections** entry in memory. You change the memory allocation by selecting a different location from the **Placement** list. The list contains the memory segments as defined in the physical memory map on the **Memory** pane. Select one of the listed memory segments to allocate the highlighted compiler section to the segment.

To see a description of the placement item, hover your mouse pointer over the item for a few moments.

**Custom Sections**

If your program uses code or data sections that are not in the **Compiler sections**, add the new sections to **Custom sections**.

**Sections**

This window lists data sections that are not in the **Compiler sections**.

**Placement**

With your new section added to the **Name** list, select the memory segment to which to add your new section. Within the restrictions imposed by the hardware and compiler, you can select a segment that appears on the list.

**Add**

Clicking **Add** lets you configure a new entry to the list of custom sections. When you click **Add**, the block provides a new temporary name in **Name**. Enter the new section name to add the section to the **Custom sections** list. After typing the new name, click **Apply** to add the new section to the list. You can also click **OK** to add the section to the list and close the dialog box.

**Name**

Enter the name of the new section here. To add a new section, click **Add**. Then, replace the temporary name with the name to use. Although the temporary name includes a period at the beginning you do not need to include the period in your new name. Names are case sensitive. `NewSection` is not the same as `newsection`, or `newSection`.

**Contents**

Identify whether the contents of the new section are `Code`, `Data`, or `Any`.

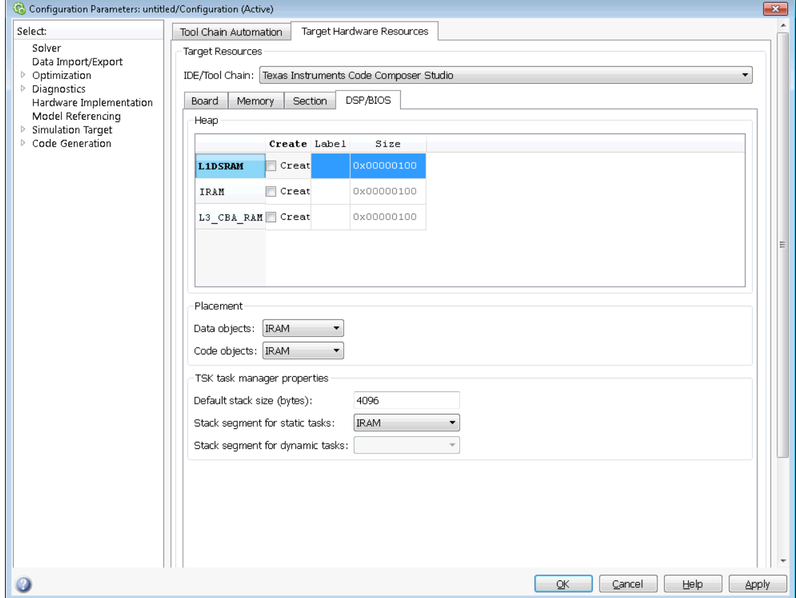
**Remove**

To remove a section from the **Custom sections** list, select the section and click **Remove**.

## Target Hardware Resources: DSP/BIOS Tab

The DSP/BIOS pane is available if the two following conditions are true:

- You are using Texas Instruments CCS IDE.
- You set the Target Hardware Resources tab **Processor** option to a C6000 processor that supports DSP/BIOS.




Selecting `DSP/BIOS` for **Operating system** on the Board pane enables this pane.

Use the **Heap**, **Placement**, and **TSK task manager properties** sections of this pane to configure various modules of DSP/BIOS.

For more information about tasks, refer to the Code Composer Studio online help.

---

**Note** To enable the **Heap** option, select DSP/BIOS for **Operating system** on the **Board** pane.

---

### Heap

The heap section contains the **Create**, **Label**, and **Size** options to manage the heap.

### Create

If your processor supports using a heap, selecting this option enables creating the heap. Define the heap using the **Label** and **Size** options. **Create** becomes unavailable for processors that do not provide a heap or do not allow you to configure the heap.

The location of the heap in the memory segment is not under your control. The only way to control the location of the heap in a segment is to make the segment and the

heap the same size. Otherwise, the compiler determines the location of the heap in the segment.

**Size**

After you select **Create**, this option lets you specify the size of the heap in words. Enter the number of words in decimal format. When you enter the heap size in decimal words, the system converts the decimal value to hexadecimal format. You can enter the value directly in hexadecimal format as well. Processors can support different maximum heap sizes.

**Label**

Selecting **Create** enables this option. Enter your label for the heap in the **Heap** option.

---

**Note** When you enter a label, the block does not verify that the label is valid. An invalid label in this field can cause errors during code generation.

---

**Placement**

Use the **Data object** and **Code object** options in **Placement** to configure the memory allocation of the selected **Heap** list entry.

**Data object**

Specify where to place new data objects in memory.

**Code object**

Specify where to place new code objects in memory.

**TSK task manager properties**

Use the **Default stack size (bytes)**, **Stack segment for static tasks**, and **Stack segment for dynamic tasks** options in **TSK task manager properties** to configure the task manager properties.

**Default stack size (bytes)**

DSP/BIOS uses a stack to save and restore variables and CPU context during thread preemption for task threads. This option sets the size of the DSP/BIOS stack in bytes allocated for each task. The software sets the default value to 4096 bytes. The maximum value is determined by the processor. Set the stack size so that tasks do not use more memory than you allocate. Exceeding the stack memory size can cause the task to write into other memory or data areas, causing unpredictable behavior.

**Stack segment for static tasks**

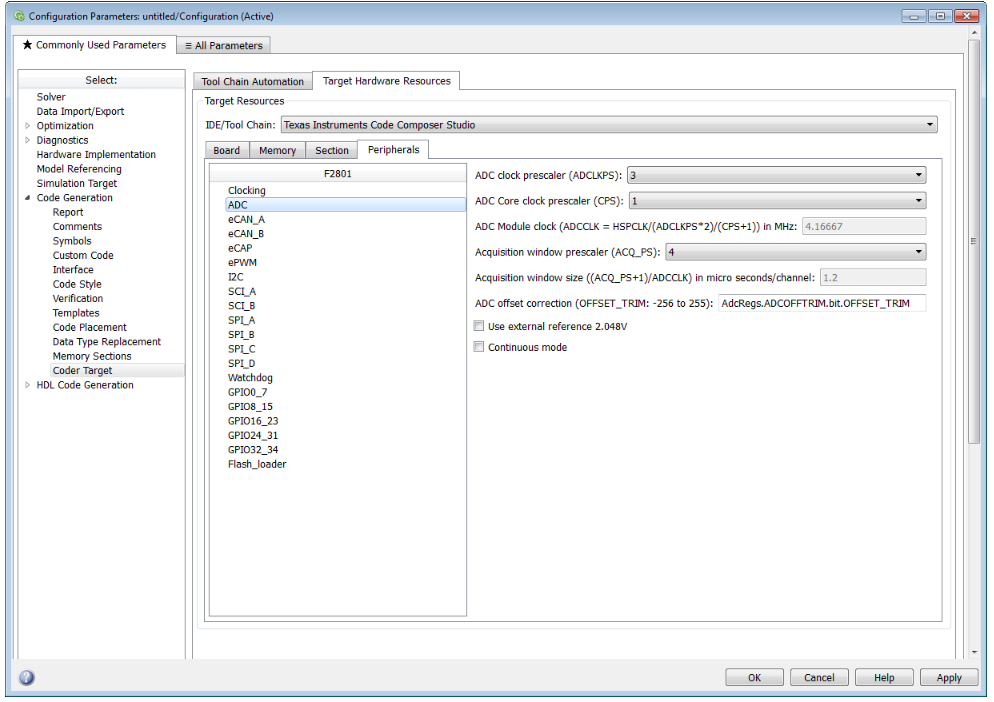
Use this option to specify where to allocate the stack for static tasks. Tasks that your program uses often are good candidates for static tasks. Infrequently used tasks usually work best as dynamic tasks.

The list offers `IDRAM` for locating the stack in memory. The Memory pane provides more options for the physical memory on the processor.

**Stack segment for dynamic tasks**

Like static tasks, dynamic tasks use a stack as well. Setting this option specifies where to locate the stack for dynamic tasks. In this case, `MEM_NULL` is the only valid stack location in memory. Allocate system heap storage to use this option. Specify the system heap configuration on the "Target Hardware Resources: Memory Tab" on page 13-33.

## Target Hardware Resources: Peripherals Tab



The Peripherals pane is only visible under the Target Hardware Resources tab, when **Board** and **Processor** parameters are configured for a C2000 processors.

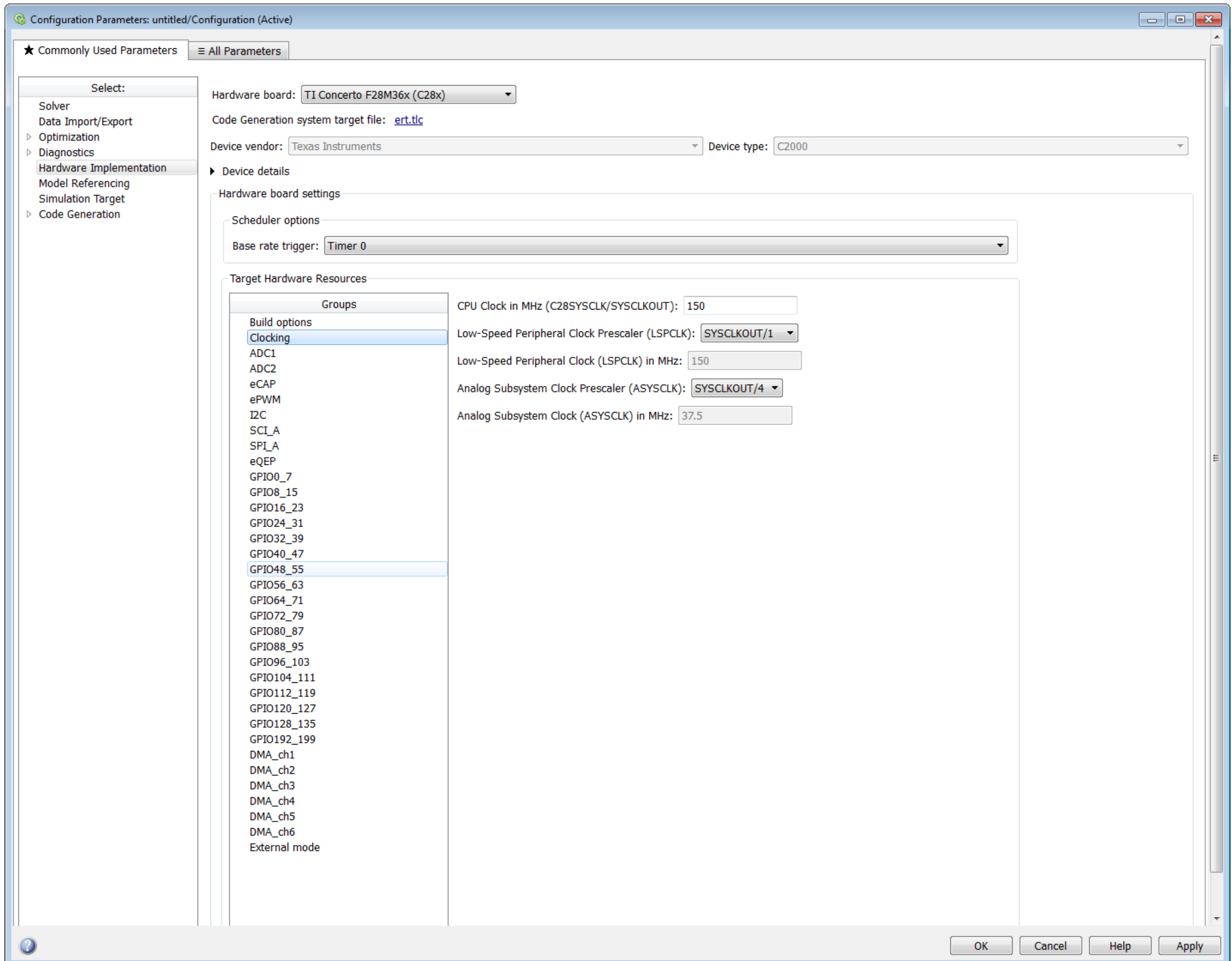
To set the attributes for a peripheral, select the peripheral from the **Peripherals** list and then set the attribute options on the right side.

The following table describes all the peripherals provided on the **Peripherals** list. Some peripherals are not available on some C2000 processors.

| Peripheral Name | Description |
|---|---|
| "C28x-Clocking" on page 13-44 | Clocking parameters to adjust clock settings and match custom oscillator frequencies |
| "C28x-ADC" on page 13-47 | Analog-to-Digital Converter (ADC) parameters |
| "C28-COMP" on page 13-50 | Parameters to assign COMP pins to GPIO pins. |
| "C28x-eCAN_A, C28x-eCAN_B" on page 13-51 | Enhanced Controller Area Network (eCAN) parameters for modules A or B |
| "C28x-eCAP" on page 13-54 | Enhanced Capture (eCAP) parameters for pin mapping to GPIO |
| "C28x-ePWM" on page 13-57 | Enhanced Pulse Width Modulation (ePWM) parameters for pin mapping to GPIO |
| "C28x-I2C" on page 13-60 | Inter-Integrated Circuit (I2C) parameters for communications |
| "C28x-SCI_A, C28x-SCI_B, C28x-SCI_C" on page 13-66 | Serial Communications Interface (SCI) parameters for communications with modules A, B, or C |
| "C28x-SPI_A, C28x-SPI_B, C28x-SPI_C, C28x-SPI_D" on page 13-69 | Serial Peripheral Interface (SPI) parameters for communications with module A, B, C, or D |

| Peripheral Name | Description |
|---|---|
| "C28x-eQEP" on page 13-72 | Enhanced Quadrature Encoder Pulse (eQEP) parameters for pin mapping to GPIO |
| "C28x-Watchdog" on page 13-74 | Watchdog enable/disable and timing |
| "C28x-GPIO" on page 13-76 | General Purpose Input Output (GPIO) parameters for input qualification types |
| "C28x-Flash_loader" on page 13-213 | Flash memory loader/programmer |
| "C28x-DMA_ch[#]" on page 13-81 | Direct Memory Access (DMA) parameters for channels 1 to N |
| "C28x-LIN" on page 13-90 | Local Interconnect Network (LIN) parameters for communications |

## C28x-Clocking




Use the clocking options to help you achieve the CPU Clock rate specified on the board. The default clocking values run the CPU clock (CLKIN) at its maximum frequency. The parameters use the external oscillator frequency on the board (OSCCLK) that is recommended by the processor vendor.

You can get feedback on the closest achievable SYSCLKOUT value with the specified Oscillator clock frequency by selecting the **Auto set PLL based on OSCCLK and CPU**

**clock** check box. Alternatively, you can manually specify the PLL value for the SYSCLKOUT value calculation.

Change the clocking values if:

- You want to change the CPU frequency.
- The external oscillator frequency differs from the value recommended by the manufacturer.

To determine the CPU frequency (CLKIN), use the following equation:

CLKIN = (OSCCLK * PLLCR) / (DIVSEL or CLKINDIV)

- CLKIN is the frequency at which the CPU operates, also known as the CPU clock.
- OSCCLK is the frequency of the oscillator.
- **PLLCR** is the PLL Control Register value.
- **CLKINDIV** is the Clock in Divider.
- **DIVSEL** is the Divider Select.

The availability of the DIVSEL or CLKINDIV parameters changes depending on the processor that you select. If neither parameter is available, use the following equation:

CLKIN = (OSCCLK * PLLCR) / 2

In case of Concerto C28x processor, make sure to match the Achievable C28x SYSCLK in MHz with the value entered here.

In the **CPU clock** parameter, enter the resulting CPU clock frequency (CLKIN).

For more information, see the “PLL-Based Clock Module” section in the Texas Instruments *Reference Guide* for your processor.

**Use internal oscillator**

Use the internal zero pin oscillator on the CPU. This parameter is enabled by default.

**Oscillator clock (OSCCLK) frequency in MHz**

The oscillator frequency that is used in the processor. This parameter is not available for TI Concerto F28M35x/ F28M36x processors.

**Auto set PLL based on OSCCLK and CPU clock**

The option that helps you set the PLL control register value automatically. When you select this check box, the values in the PLLCR, DIVSEL , and the Closest achievable

SYSCLKOUT in MHz parameters are automatically calculated based on the **CPU Clock** value entered on the Board. This parameter is not available for TI Concerto F28M35x/ F28M36x processors.

**PLL control register (PLLCR)**

If you select the **Auto set PLL based on OSCCLK and CPU clock** check box, the auto calculated control register value achieves the specified CPU Clock value, based on the Oscillator clock frequency. Otherwise, you can select a value for PLL control register. This parameter is not available for TI Concerto F28M35x/ F28M36x processors.

**Clock divider (DIVSEL)**

If you select the **Auto set PLL based on OSCCLK and CPU clock** check box, the auto calculated clock divider value achieves the specified CPU Clock value based on the Oscillator clock frequency. Otherwise, you can select a value for Clock divider (DIVSEL). This parameter is not available for TI Concerto F28M35x/ F28M36x processors.

**Closest achievable SYSCLKOUT in MHz = (OSCCLK*PLLCR)/DIVSELClosest achievable SYSCLKOUT in MHz = (OSCCLK*PLLCR)/CLKINDIV**

The auto calculated feedback value that matches most closely to the desired CPU Clock value on the board, based on the values of OSCCLK, PLLCR, and the DIVSEL. This parameter is not available for TI Concerto F28M35x/ F28M36x processors.

**Low-Speed Peripheral Clock Prescaler (LSPCLK)**

The value by which to scale the LSPCLK. This value is based on the SYSCLKOUT.

**Low-Speed Peripheral Clock (LSPCLK) in MHz**

This value is calculated based on LSPCLK Prescaler. Example: SPI uses a LSPCLK.

**High-Speed Peripheral Clock Prescaler (HSPCLK)**

The value by which to scale the HSPCLK. This value is based on the SYSCLKOUT.

**High-Speed Peripheral Clock (HSPCLK) in MHZ**

This value is calculated based on HSPCLK Prescaler. Example: ADC uses a HSPCLK.

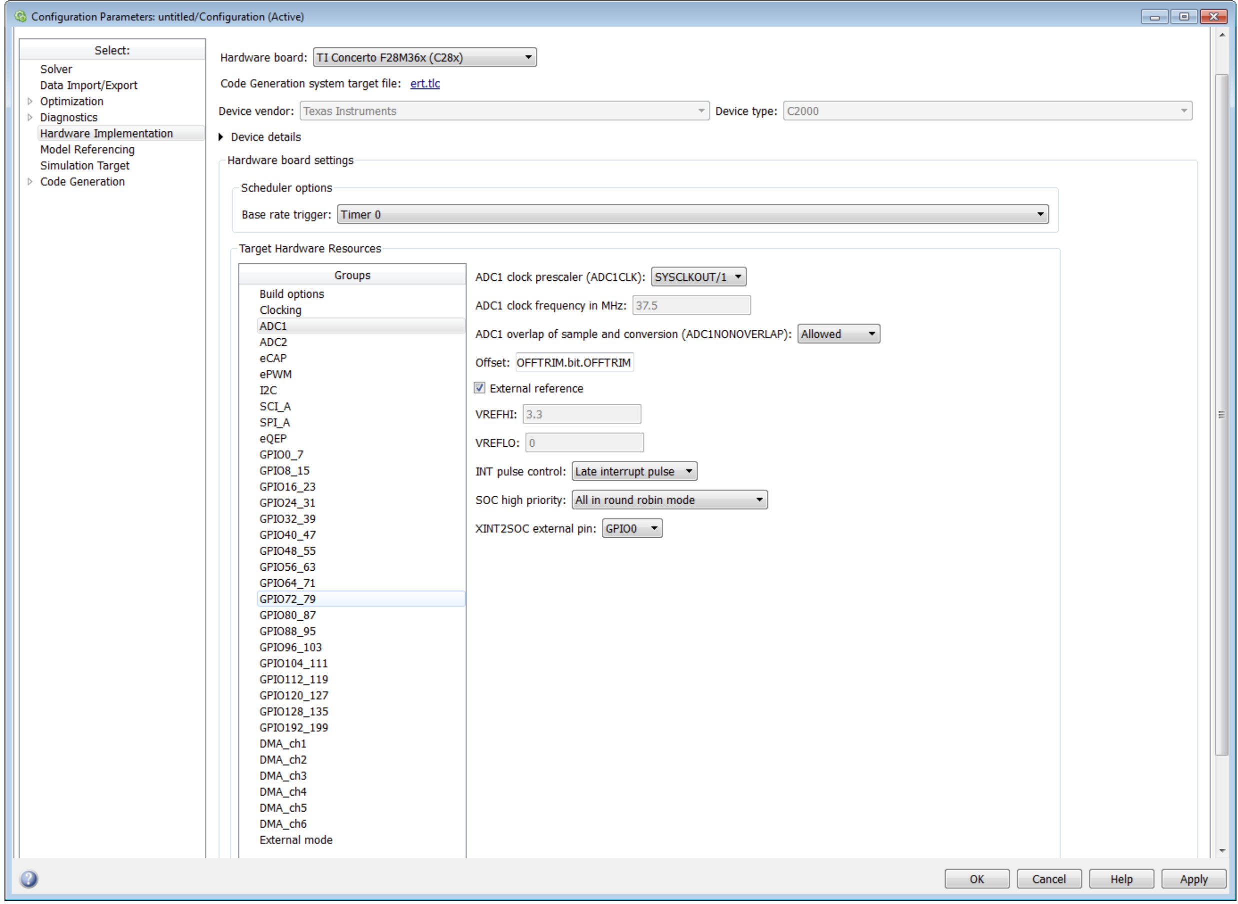
**Analog Common Interface Bus Clock (ACIB)**

The value by which to scale the bus clock. This option is available only for TI Concerto F28M35x/ F28M36x processors.

**Analog Common Interface Bus Clock (ACIB) in MHz**

This value is calculated based on the ACIB value. This option is available only for TI Concerto F28M35x/ F28M36x processors.

## C28x-ADC




The high-speed peripheral clock (HSPCLK) controls the internal timing of the ADC module. The ADC derives the operating clock speed from the HSPCLK speed in several prescaler stages. For more information about configuring these scalers, refer to “Configuring ADC Parameters for Acquisition Window Width”.

You can set the following parameters for the ADC clock prescaler:

**Select the CPU core which controls ADC_x module**

Select the CPU core to control the ADC module.

**ADC clock prescaler (ADCCLK)**

The option to select the ADCCLK divider for processors c2802x, c2803x, c2806x, F28M3x, F2807x, or F2837x.

**ADC clock frequency in MHz**

The clock frequency for ADC. This is a read-only field and the value in this field is based on the value you select in **ADC clock prescaler (ADCCLK)**.

**ADC overlap of sample and conversion (ADC#NONOVERLAP)**

The option to enable or disable overlap of sample and conversion.

**ADC clock prescaler (ADCLKPS)**

The HSPCLK speed is divided by this 4-bit value as the first step in deriving the core clock speed of the ADC. The default value is 3.

**ADC Core clock prescaler (CPS)**

After dividing the HSPCLK speed by the **ADC clock prescaler (ADCLKPS)** value, setting the **ADC clock prescaler (ADCLKPS)** parameter to 1, the default value, divides the result by 2.

**ADC Module clock (ADCCLK = HSPCLK/ADCLKPS*2)/(CPS+1)) in MHz**

The clock to the ADC module and indicates the ADC operating clock speed.

**Acquisition window prescaler (ACQ_PS)**

This value does not directly alter the core clock speed of the ADC. It serves to determine the width of the sampling or acquisition period. The higher the value, the wider is the sampling period. The default value is 4.

**Acquisition window size ((ACQ_PS+1)/ADCCLK) in micro seconds/channel**

Acquisition window size determines for what time duration the sampling switch is closed. The width of SOC pulse is ADCTRL1[11:8] + 1 times the ADCLK period.

**Offset**

Enter the offset value.

**Use external reference 2.048VExternal reference**

By default, an internally generated band gap voltage reference supplies the ADC logic. However, depending on application requirements, you can enable the external reference so the ADC logic uses an external voltage reference instead. Select the check box to use a 2.048V external voltage reference.

**Use external reference**

By default, an internally generated band gap voltage reference supplies the ADC logic. However, depending on application requirements, you can enable the external

reference so the ADC logic uses an external voltage reference instead. Select the check box to use an external voltage reference.

**Continuous mode**

When the ADC generates an end of conversion (EOC) signal, generate an ADCINT# interrupt whether the previous interrupt flag has been acknowledged or not.

**ADC offset correction (OFFSET_TRIM: -256 to 255)**

The 280x ADC supports offset correction via a 9-bit value that it adds or subtracts before the results are available in the ADC result registers. Timing for results is not affected. The default value is `0`.

**VREFHIVREFLO**

When you disable the **Use external reference 2.048V** or **External reference** option, the ADC logic uses a fixed 0-volt to 3.3-volt input range and the software disables **VREFHI** and **VREFLO**. To interpret the ADC input as a ratiometric signal, select the **External reference** option. Then set values for the high voltage reference (**VREFHI**) and the low voltage reference (**VREFLO**). **VREFHI** uses the external ADCINA0 pin, and **VREFLO** uses the internal GND.

**INT pulse control**

Use this option to configure when the ADC sets ADCINTFLG ADCINTx relative to the SOC and EOC Pulses. Select `Late interrupt pulse` or `Early interrupt pulse`.

**SOC high priority**

Use this option to enable and configure **SOC high priority mode** . In `all in round robin mode`, the default selection, the ADC services each SOC interrupt in a numerical sequence.

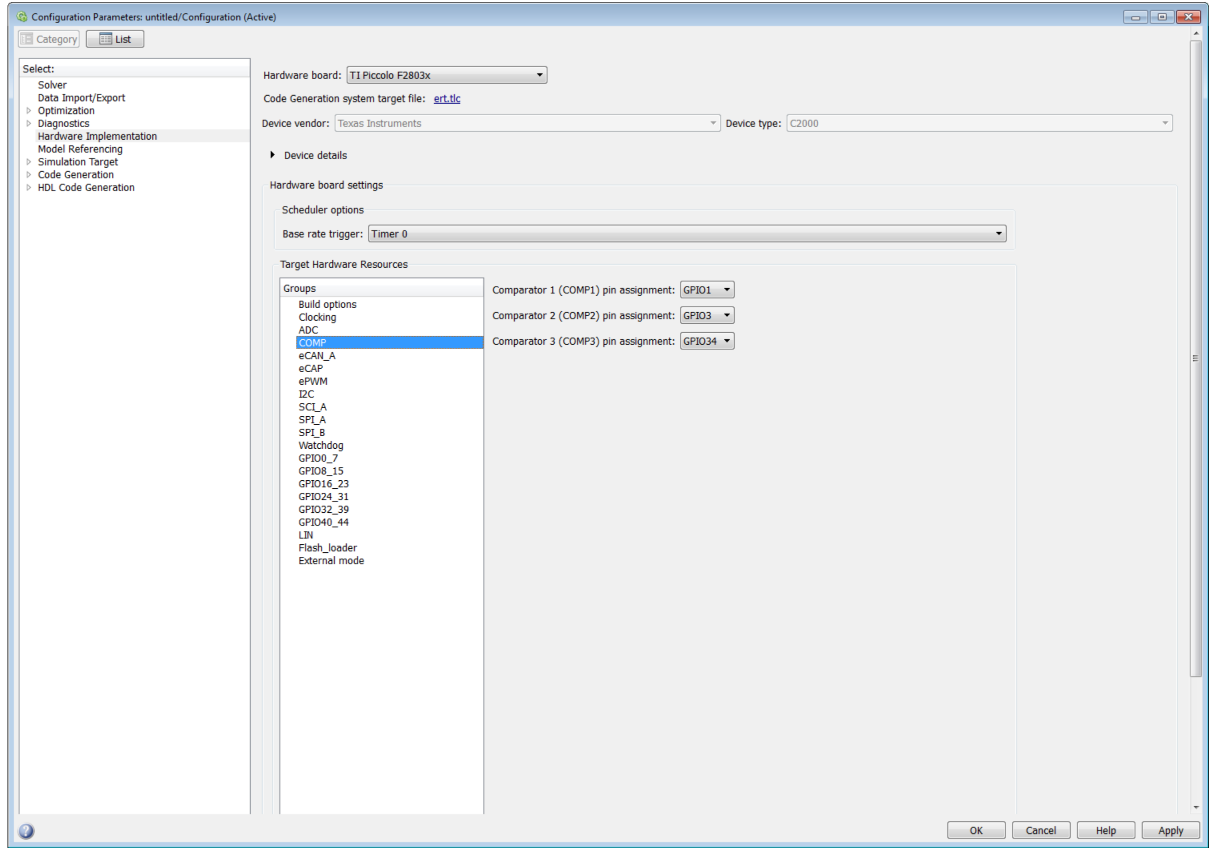
Choose one of the `high priority` selections to assign high priority to one or more of the SOCs. In this mode, the ADC operates in round robin mode until it receives a high priority SOC interrupt. The ADC finishes servicing the current SOC, services the high priority SOCs, and then returns to the next SOC in the round robin sequence.

For example, the ADC is servicing SOC8 when it receives a high priority interrupt on SOC1. The ADC completes servicing SOC8, services SOC1, and then services SOC9.

**XINT2SOC external pin**

Select the pin to which the ADC sends the XINT2SOC pulse.

## C28-COMP




Assigns COMP pins to GPIO pins.

### Comparator 1 (COMP1) pin assignment

Select an option from the list — None,GPIO1, GPIO20, GPIO42.

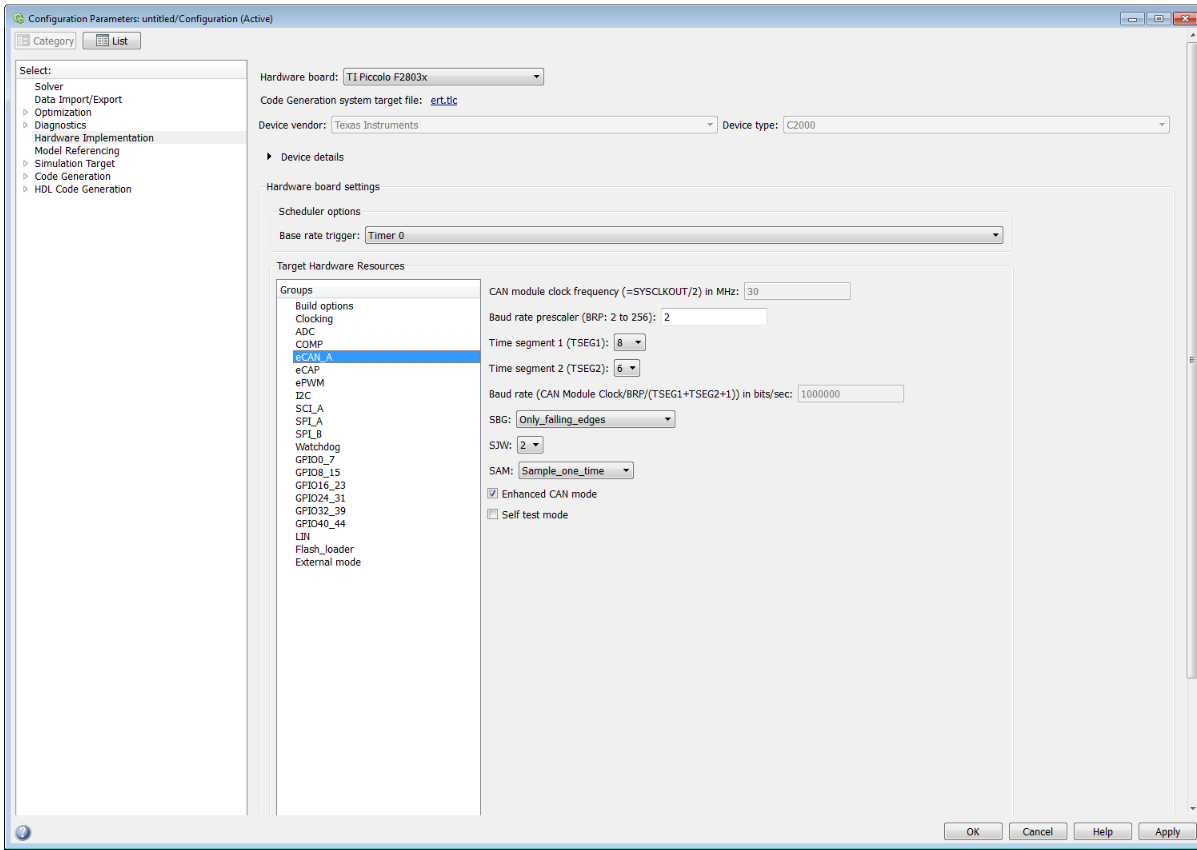
### Comparator 2 (COMP2) pin assignment

Select an option from the list — None,GPIO3, GPIO21, GPIO34, GPIO43.

### Comparator 3 (COMP3) pin assignment

Select an option from the list — None,GPIO34.

## C28x-eCAN_A, C28x-eCAN_B




For more help on setting the timing parameters for the eCAN modules, refer to "Configuring Timing Parameters for CAN Blocks" (Embedded Coder Support Package for Texas Instruments C2000 Processors). You can set the following parameters for the eCAN module:

**CAN module clock frequency (= SYSCLKOUT) in MHz:**

The clock to the enhanced CAN module. The CAN module clock frequency is equal SYSCLKOUT for processors such as c280x, c281x, c28044.

**CAN module clock frequency (=SYSCLKOUT/2) in MHz**

The clock to the enhanced CAN module. The CAN module clock frequency is equal to SYSCLKOUT/2 for processors such as piccolo, c2834x, c28x3x.

**Baud rate prescaler (BRP: 2 to 256)/Baud rate prescaler (BRP: 1 to 1024):**

Value by which to scale the bit rate.

**Time segment 1 (TSEG1):**

Sets the value of time segment 1, which, with **TSEG2** and **Baud rate prescaler**, determines the length of a bit on the eCAN bus. Valid values for **TSEG1** are from `1` through `16`.

**Time segment 2 (TSEG2):**

Sets the value of time segment 2, which, with **TSEG1** and **Baud rate prescaler**, determines the length of a bit on the eCAN bus. Valid values for **TSEG2** are from `1` through `8`.

**Baud rate (CAN Module Clock/BRP/(TSEG1 + TSEG2 +1)) in bits/sec:**

CAN module communication speed represented in bits/sec.

**SBG**

Sets the message resynchronization triggering. Options are `Only_falling_edges` and `Both_falling_and_rising_edges`.

**SJW**

Sets the synchronization jump width, which determines how many units of TQ a bit can be shortened or lengthened when resynchronizing.

**SAM**

Number of samples used by the CAN module to determine the CAN bus level. Selecting `Sample_one_time` samples once at the sampling point. Selecting `Sample_three_times` samples once at the sampling point and twice before at a distance of TQ/2. The CAN module makes a majority decision from the three points.

**Enhanced CAN Mode**

To enable time-stamping and to use **Mailbox Numbers** 16 through 31 in the C2000 eCAN blocks, enable this parameter. Texas Instruments documentation refers to this “HECC mode”.

**Self test mode**

If you set this parameter to `True`, the eCAN module goes to loopback mode. Loopback mode sends a “dummy” acknowledge message back without needing an acknowledge bit. The default is `False`.

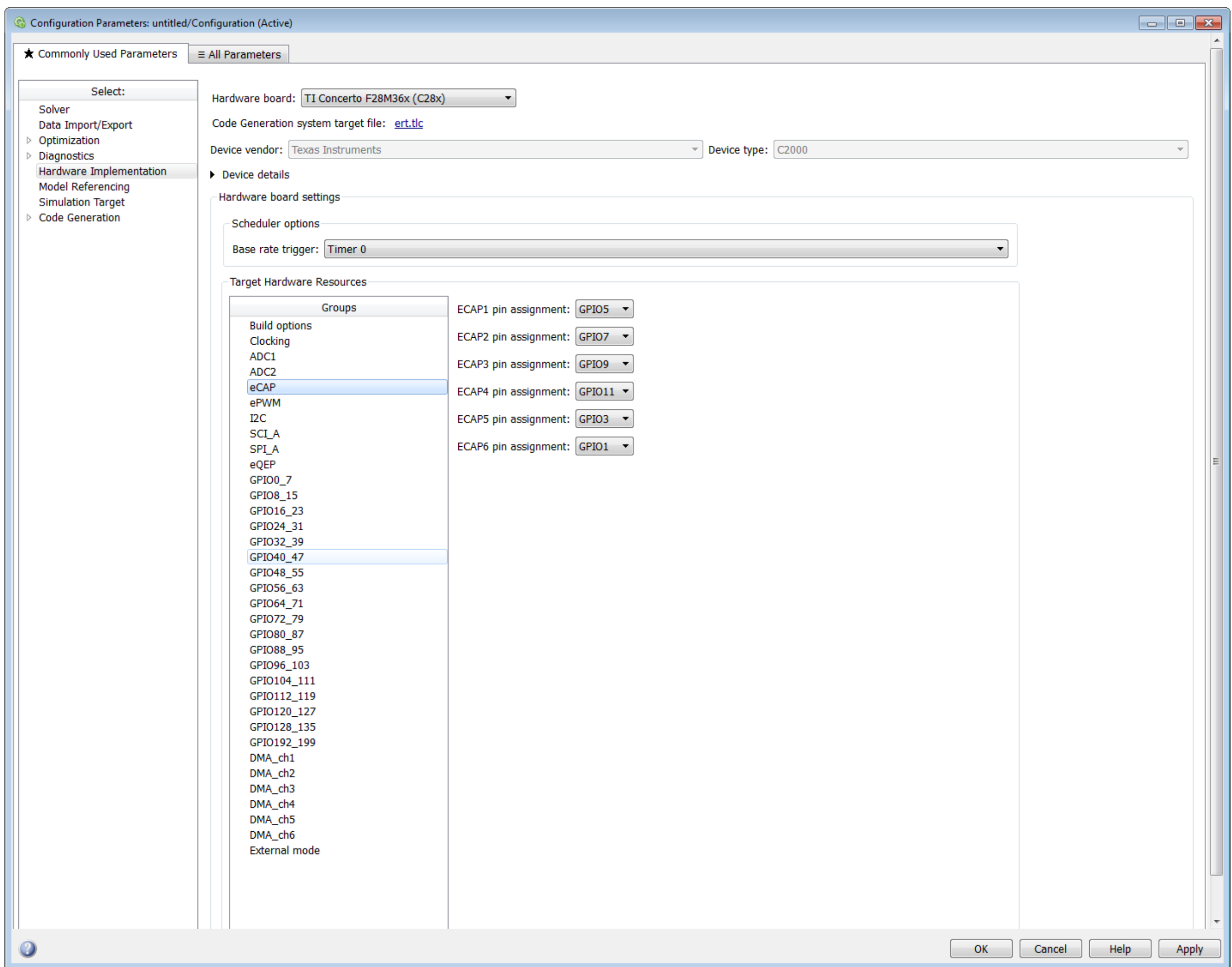
**Pin assignment (Tx)**

Assigns the CAN transmit pin to use with the eCAN_B module. Possible values are GPIO8, GPIO12, GPIO16, and GPIO20.

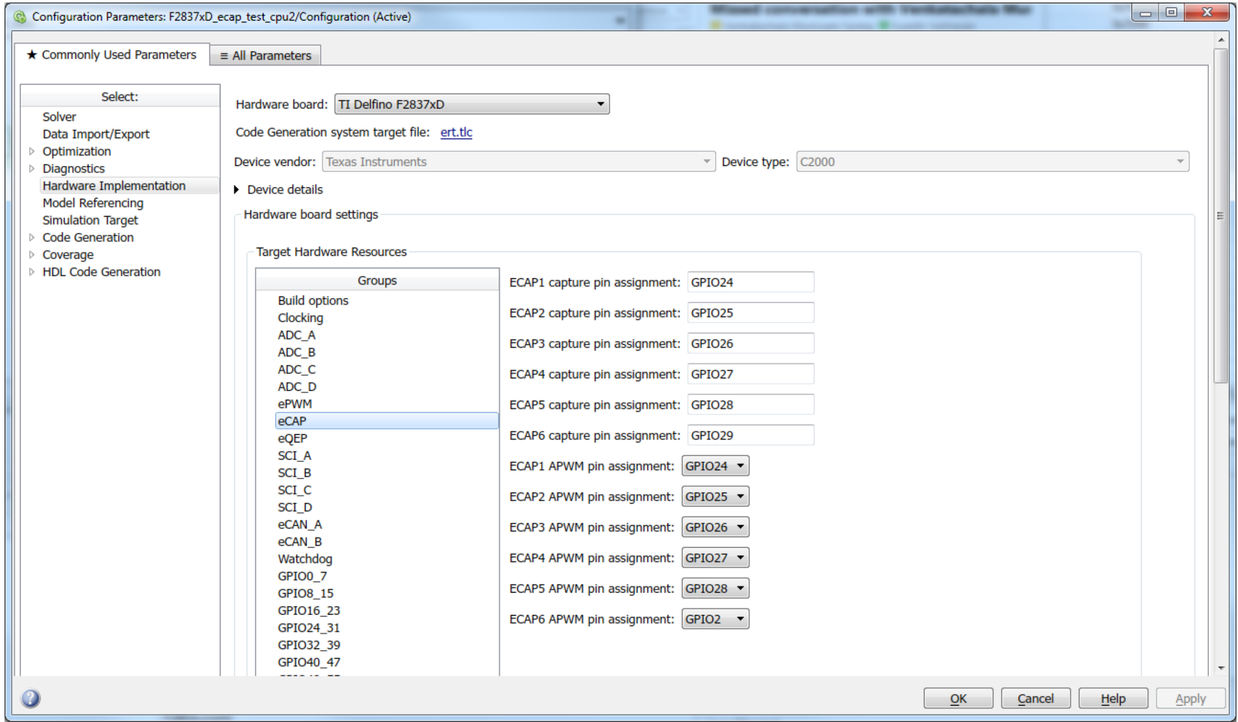
**Pin assignment (Rx)**

Assigns the CAN receive pin to use with the eCAN_B module. Possible values are GPIO10, GPIO13, GPIO17, and GPIO21.

## C28x-eCAP




The following screen shows the eCAP parameters for F2837xS, F2837xD and F2807x processors.




Assigns eCAP pins to GPIO pins.

**ECAP1 pin assignment**

Select a GPIO pin for ECAP1 module.

**ECAP2 pin assignment**

Select a GPIO pin for ECAP2 module.

**ECAP3 pin assignment**

Select a GPIO pin for ECAP3 module.

**ECAP4 pin assignment**

Select a GPIO pin for ECAP4 module.

**ECAP5 pin assignment**

Select a GPIO pin for ECAP5 module.

**ECAP6 pin assignment**

Select a GPIO pin for ECAP6 module.

The parameters described below are only for F2837xS, F2837xD, and F2807x processors.

**ECAP1 capture pin assignment**

Select GPIO pin for ECAP1 module when using in eCAP (capture) operating mode.

**ECAP2 capture pin assignment**

Select a GPIO pin for ECAP2 module.

**ECAP3 capture pin assignment**

Select GPIO pin for ECAP3 module when using in eCAP (capture) operating mode.

**ECAP4 capture pin assignment**

Select GPIO pin for ECAP4 module when using in eCAP (capture) operating mode.

**ECAP5 capture pin assignment**

Select GPIO pin for ECAP5 module when using in eCAP (capture) operating mode.

**ECAP6 capture pin assignment**

Select GPIO pin for ECAP6 module when using in eCAP (capture) operating mode.

**ECAP1 APWM pin assignment**

Select GPIO pin for ECAP1 module when using in APWM operating mode.

**ECAP2 APWM pin assignment**

Select GPIO pin for ECAP2 module when using in APWM operating mode.

**ECAP3 APWM pin assignment**

Select GPIO pin for ECAP3 module when using in APWM operating mode.

**ECAP4 APWM pin assignment**

Select GPIO pin for ECAP4 module when using in APWM operating mode.

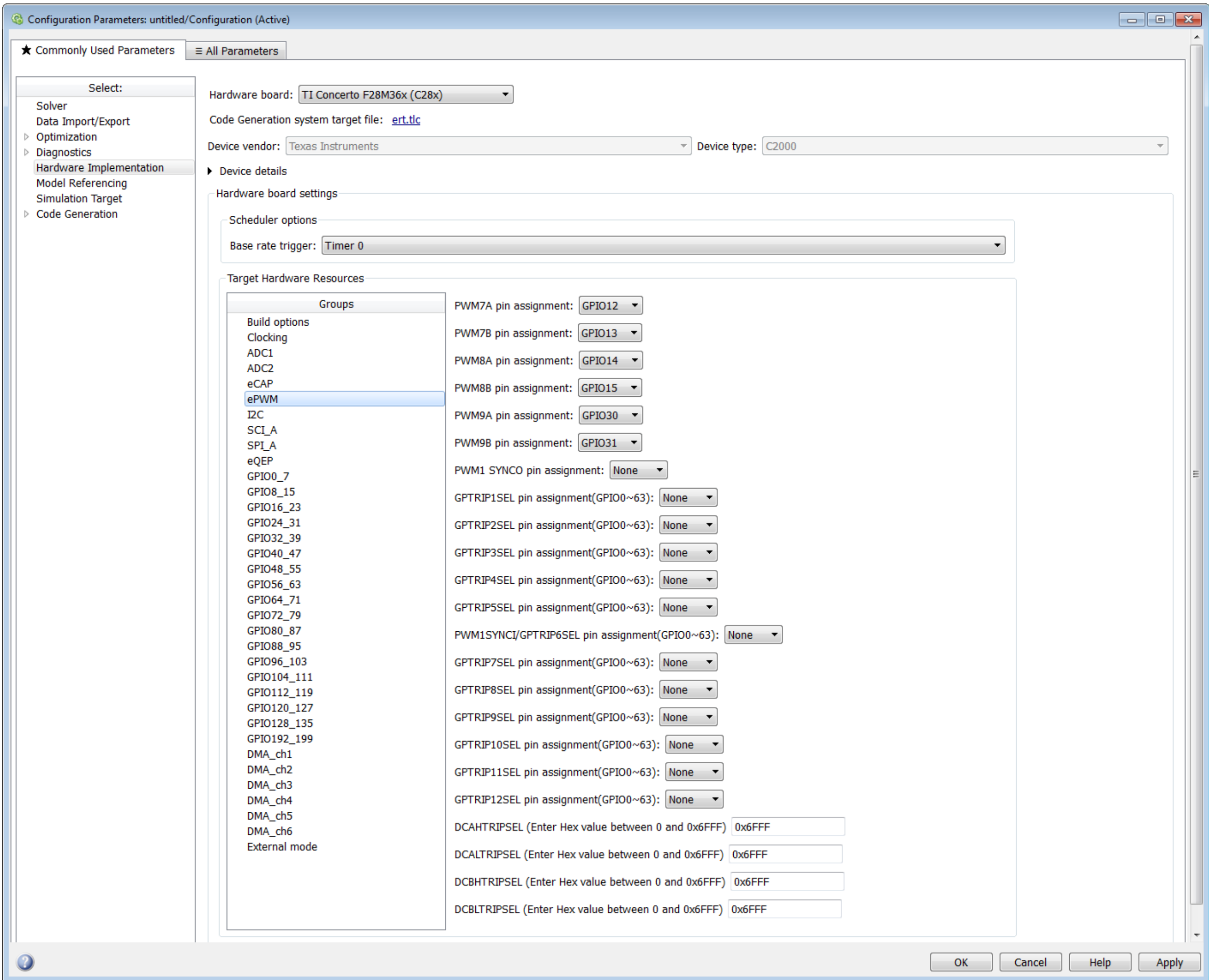
**ECAP5 APWM pin assignment**

Select GPIO pin for ECAP5 module when using in APWM operating mode.

**ECAP6 APWM pin assignment**

Select GPIO pin for ECAP6 module when using in APWM operating mode.

## C28x-ePWM




Assigns ePWM signals to GPIO pins.

### EPWM clock divider (EPWMCLKDIV)

This parameter appears only for F2807x and F2837x processors. This parameter allows you to select the ePWM clock divider.

**TZ1 pin assignment**

Assigns the trip-zone input 1 (TZ1) to a GPIO pin. Choices are `None` (the default), `GPIO12`, and `GPIO15`.

**TZ2 pin assignment**

Assigns the trip-zone input 2 (TZ2) to a GPIO pin. Choices are `None` (the default), `GPIO16`, and `GPIO28`.

**TZ3 pin assignment**

Assigns the trip-zone input 3 (TZ3) to a GPIO pin. Choices are `None` (the default), `GPIO17`, and `GPIO29`.

**TZ4 pin assignment**

Assigns the trip-zone input 4 (TZ4) to a GPIO pin. Choices are `None` (the default), `GPIO17`, and `GPIO28`.

**TZ5 pin assignment**

Assigns the trip-zone input 5 (TZ5) to a GPIO pin. Choices are `None` (the default), `GPIO16`, and `GPIO28`.

**TZ6 pin assignment**

Assigns the trip-zone input 6 (TZ6) to a GPIO pin. Choices are `None` (the default), `GPIO17`, and `GPIO29`.

**Note** The TZ# pin assignments are available only for TI F280x processors.

**SYNCI pin assignment**

Assigns the ePWM external sync pulse input (SYNCI) to a GPIO pin. Choices are `None` (the default), `GPIO6`, and `GPIO32`.

**SYNCO pin assignment**

**Note** SYNCI and SYNCO pin assignments are available for TI F28044, TI F280x, TI Delfino F2833x, TI Delfino F2834x, TI Piccolo F2802x, TI Piccolo F2803x, TI Piccolo F2806 processors.

Assigns the ePWM external sync pulse output (SYNCO) to a GPIO pin. Choices are `None` (the default), `GPIO6`, and `GPIO33`.

**PWM#A, PWM#B, PWM#C pin assignment**

The PWM # A, PWM # B, PWM # C pin assignment.

**GPTRIP#SEL pin assignment(GPIO0~63)**

Assigns the ePWM trip-zone input to a GPIO pin.

---

**Note** The GPTRIP#SEL pin assignment is available only for TI Concerto F28M35x/F28M36x processors.

---

**PWM1SYNCI/ GPTRIP6SEL pin assignment**

Assigns the ePWM sync pulse input (SYNCI) to a GPIO pin.

---

**Note** The PWM1SYNCI/GPTRIP#SEL pin assignments are available only for TI Concerto F28M35x/F28M36x processors.

---

**DCAHTRIPSEL (Enter Hex value between 0 and 0x6FFF) / DCBHTRIPSEL (Enter Hex value between 0 and 0x6FFF)**

Assigns the Digital Compare A High Trip Input to a GPIO pin.

---

**Note** DCAHTRIPSel pin assignment is available only for TI Concerto F28M35x/F28M36x processors.

---

**DCALTRIPSEL (Enter Hex value between 0 and 0x6FFF) / DCBLTRIPSEL (Enter Hex value between 0 and 0x6FFF)**

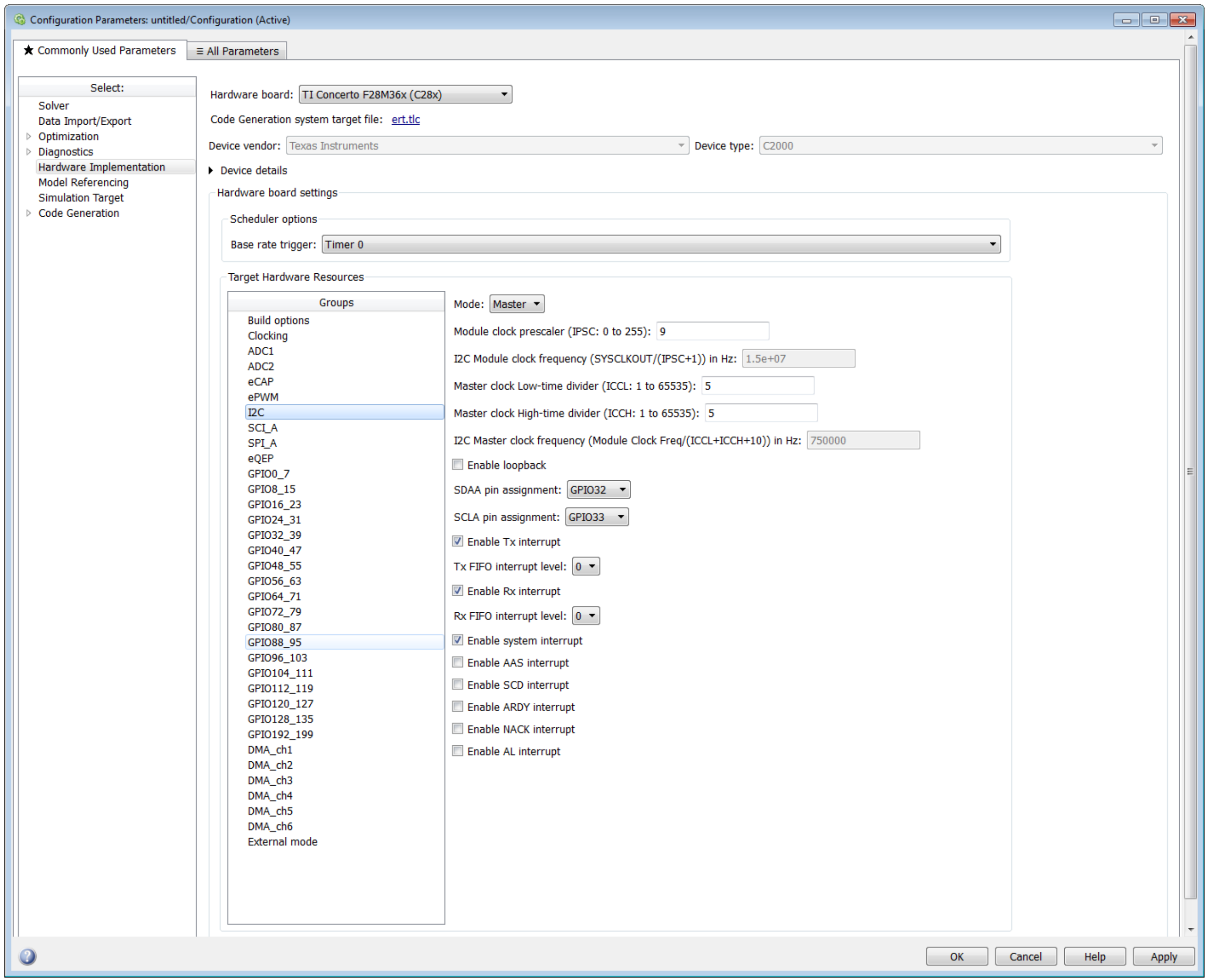
Assigns the Digital Compare A High Trip Input to a GPIO pin.

---

**Note** The DCALTRIPSEL pin assignment is available only for TI Concerto F28M35x/F28M36x processors.

---

## C28x-I2C




Report or set Inter-Integrated Circuit parameters. For more information, consult the *TMS320x280x/ TMS320F28M35x/ TMS320F28M36x Inter-Integrated Circuit Module Reference Guide*, Literature Number: SPRU721A/ SPRUH22F/ SPRUHE8B available on the Texas Instruments Web site.

**Mode**

Configure the I2C module as `Master` or `Slave`.

If a module is an I2C master, it:

Initiates communication with slave nodes by sending the slave address and requesting data transfer to or from the slave.

Outputs the **Master clock frequency** on the serial clock line (SCL) line.

If a module is an I2C slave, it:

- Synchronizes itself with the serial clock line (SCL) line.
- Responds to communication requests from the master.

When **Mode** is `Slave`, you can configure the **Addressing format**, **Address register**, and **Bit count** parameters.

The **Mode** parameter corresponds to bit 10 (MST) of the I2C Mode Register (I2CMDR).

**Addressing format**

If **Mode** is `Slave`, determine the addressing format of the I2C master, and set the I2C module to the same mode:

- `7-Bit Addressing`, the normal address mode.
- `10-Bit Addressing`, the expanded address mode.
- `Free Data Format`, a mode that does not use addresses. (If you **Enable loopback**, the `Free data format` is not supported.)

The **Addressing format** parameter corresponds to bit 3 (FDF) and bit 8 (XA) of the I2C Mode Register (I2CMDR).

**Own address register**

If **Mode** is `Slave`, enter the 7-bit (0–127) or 10-bit (0–1023) address this I2C module uses while it is a slave.

This parameter corresponds to bits 9–0 (OAR) of the I2C Own Address Register (I2COAR).

**Bit count**

If **Mode** is `Slave`, set the number of bits in each data *byte* the I2C module transmits and receives. This value must match that of the I2C master.

This parameter corresponds to bits 2–0 (BC) of the I2C Mode Register (I2CMDR).

**Module clock prescaler (IPSC: 0 to 255):**

If **Mode** is `Master`, configure the module clock frequency by entering a value 0–255.

*Module clock frequency* = *I2C input clock frequency* / (*Module clock prescaler* + 1)

The I2C specifications require a module clock frequency between 7 MHz and 12 MHz.

The *I2C input clock frequency* depends on the DSP input clock frequency and the value of the PLL Control Register divider (PLLCR). For more information on setting the PLLCR, consult the documentation for your specific Digital Signal Controller.

This **Module clock prescaler (IPSC: 0 to 255):** corresponds to bits 7–0 (IPSC) of the I2C Prescaler Register (I2CPSC).

**I2C Module clock frequency (SYSCLKOUT / (IPSC+1)) in Hz:**

This field displays the frequency the I2C module uses internally. To set this value, change the **Module clock prescaler**.

For more information about this value, consult the “Formula for the Master Clock Period” section in the *TMS320x280x Inter-Integrated Circuit Module Reference Guide*, Literature Number: SPRU721, on the Texas Instruments Web site.

**I2C Master clock frequency (Module Clock Freq/(ICCL+ICCH+10)) in Hz:**

This field displays the master clock frequency.

For more information about this value, consult the “Clock Generation” section in the *TMS320x280x/ TMS320F28M35x/ TMS320F28M36x Inter-Integrated Circuit Module Reference Guide*, Literature Number: SPRU721/ SPRUH22F/ SPRUHE8B, available on the Texas Instruments Web site.

**Master clock Low-time divider (ICCL: 1 to 65535):**

When **Mode** is `Master`, this divider determines the duration of the low state of the SCL line on the I2C-bus.

The low-time duration of the master clock = Tmod x (ICCL + d).

For more information, consult the “Formula for the Master Clock Period” section in the *TMS320x280x/ TMS320F28M35x/ TMS320F28M36x Inter-Integrated Circuit*

*Module Reference Guide*, Literature Number: SPRU721A/ SPRUH22F/ SPRUHE8B, available on the Texas Instruments Web site.

This parameter corresponds to bits 15–0 (ICCL) of the Clock Low-Time Divider Register (I2CCLKL).

**Master clock High-time divider (ICCH: 1 to 65535):**

When **Mode** is `Master`, this divider determines the duration of the high state on the serial clock pin (SCL) of the I2C-bus.

The high-time duration of the master clock = Tmod x (ICCL + d).

For more information about this value, consult the “Formula for the Master Clock Period” section in the *TMS320x280x/ TMS320F28M35x/ TMS320F28M36x Inter-Integrated Circuit Module Reference Guide*, Literature Number: SPRU721A, SPRUH22f, SPRUHE8B, available on the Texas Instruments Web site.

This parameter corresponds to bits 15–0 (ICCH) of the Clock High-Time Divider Register (I2CCLKH).

**Enable loopback**

When **Mode** is `Master`, enable or disable digital loopback mode. In digital loopback mode, I2CDXR transmits data over an internal path to I2CDRR, which receives the data after a configurable delay.

The delay, measured in DSP cycles, equals (I2C input clock frequency/module clock frequency) x 8.

While **Enable loopback** is enabled, free data format addressing is not supported.

This parameter corresponds to bit 6 (DLB) of the I2C Mode Register (I2CMDR).

**SDAA pin assignment**

Select a GPIO pin as I2C data bidirectional port.

This parameter is available only for TI Concerto F28M36x (C28x) processors.

**SCLA pin assignment**

Select the GPIO pin as I2C clock bidirectional port.

This parameter is available only for TI Concerto F28M36x (C28x) processors.

### Enable Tx interrupt

This parameter corresponds to bit 5 (TXFFIENA) of the I2C Transmit FIFO Register (I2CFFTX).

### Tx FIFO interrupt level

This parameter corresponds to bits 4–0 (TXFFIL4-0) of the I2C Transmit FIFO Register (I2CFFTX).

### Enable Rx interrupt

This parameter corresponds to bit 5 (RXFFIENA) of the I2C Receive FIFO Register (I2CFFRX).

### Rx FIFO interrupt level

This parameter corresponds to bit 4–0 (RXFFIL4-0) of the I2C Receive FIFO Register (I2CFFRX).

### Enable system interrupt

Select this parameter to display and individually configure the following five Basic I2C Interrupt Request parameters in the Interrupt Enable Register (I2CIER):

- Enable AAS interrupt
- Enable SCD interrupt
- Enable ARDY interrupt
- Enable NACK interrupt
- Enable AL interrupt

### Enable AAS interrupt

Enable the addressed-as-slave interrupt.

When enabled, the I2C module generates an interrupt (AAS bit = 1) upon receiving one of the following:

- Its **Own address register**
- A general call (all zeros)
- A data byte is in free data format

When enabled, the I2C module clears the interrupt (AAS = 0) upon receiving one of the following:

- Multiple START conditions (7-bit addressing mode only)

- A slave address that is different from **Own address register** (10-bit addressing mode only)
- A NACK or a STOP condition

This parameter corresponds to bit 6 (AAS) of the Interrupt Enable Register (I2CIER).

**Enable SCD interrupt**

Enable stop condition detected interrupt.

When enabled, the I2C module generates an interrupt (SCD bit = 1) when the CPU detects a stop condition on the I2C bus.

When enabled, the I2C module clears the interrupt (SCD = 0) upon one of the following events:

- The CPU reads the I2CISRC while it indicates a stop condition
- A reset of the I2C module
- Someone manually clears the interrupt

This parameter corresponds to bit 5 (SCD) of the Interrupt Enable Register (I2CIER).

**Enable ARDY interrupt**

Enable register-access-ready interrupt enable bit.

When enabled, the I2C module generates an interrupt (ARDY bit = 1) when the previous address, data, and command values in the I2C module registers have been used and new values can be written to the I2C module registers.

This parameter corresponds to bit 2 (ARDY) of the Interrupt Enable Register (I2CIER).

**Enable NACK interrupt**

Enable no acknowledgment interrupt enable bit.

When enabled, the I2C module generates an interrupt (NACK bit = 1) when the module is a transmitter in master or slave mode and it receives a NACK condition.

This parameter corresponds to bit 1 (NACK) of the Interrupt Enable Register (I2CIER).

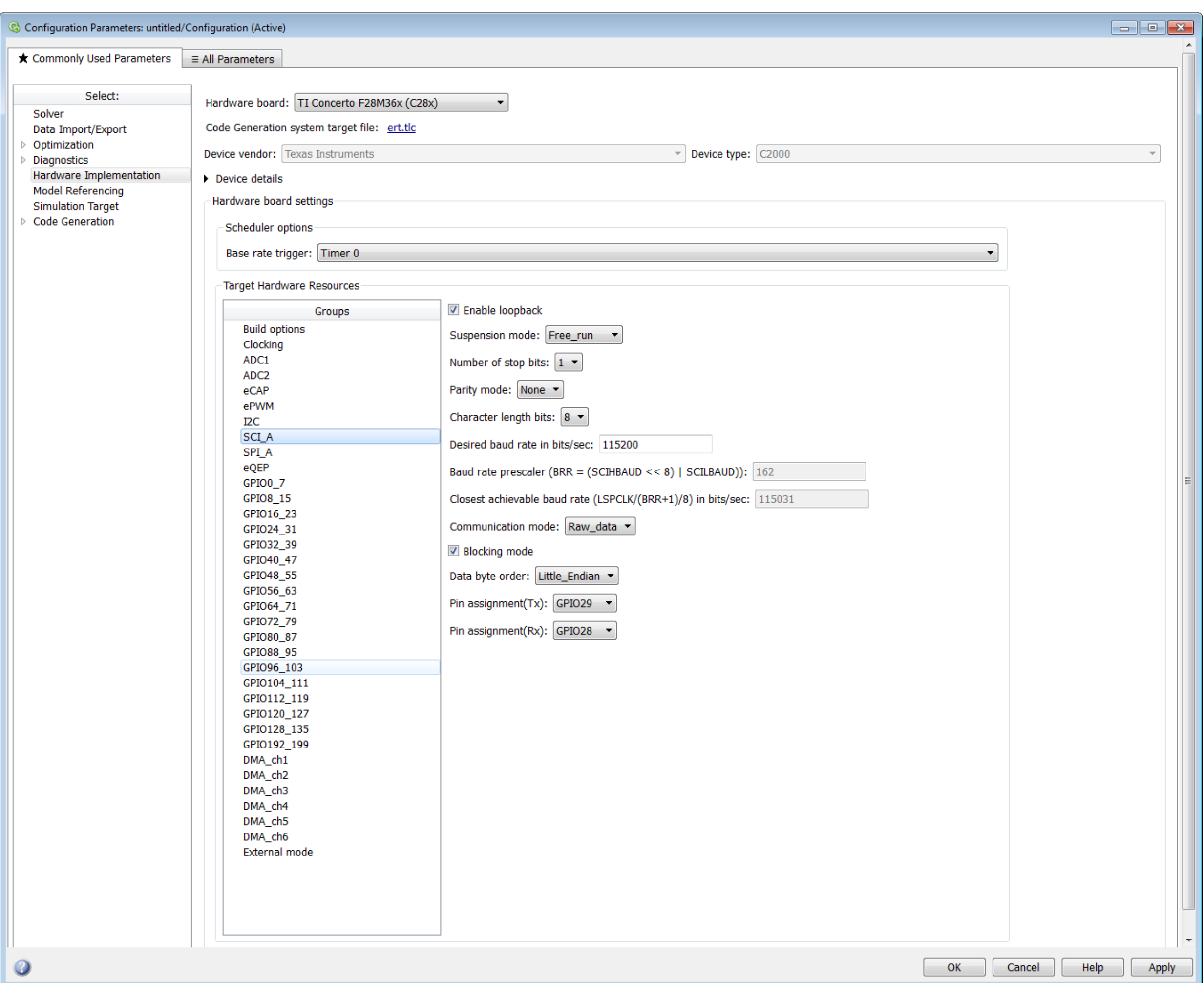
**Enable AL interrupt**

Enable arbitration-lost interrupt.

When enabled, the I2C module generates an interrupt (AL bit = 1) when the I2C module is operating as a master transmitter and looses an arbitration contest with another master transmitter.

This parameter corresponds to bit 0 (AL) of the Interrupt Enable Register (I2CIER).

## C28x-SCI_A, C28x-SCI_B, C28x-SCI_C



The serial communications interface parameters you can set for module A. These parameters are:

**Enable loopback**

Select this parameter to enable the loopback function for self-test and diagnostic purposes only. When this function is enabled, a C28x DSP Tx pin is internally connected to its Rx pin and can transmit data from its output port to its input port to check the integrity of the transmission.

**Baud rate**

Baud rate for transmitting and receiving data. Select from `115200` (the default), `57600`, `38400`, `19200`, `9600`, `4800`, `2400`, `1200`, `300`, and `110`.

**Suspension mode**

Type of suspension to use when debugging your program with Code Composer Studio. When your program encounters a breakpoint, the suspension mode determines whether to perform the program instruction. Available options are `Hard_abort`, `Soft_abort`, and `Free_run`. `Hard_abort` stops the program immediately. `Soft_abort` stops when the current receive/transmit sequence is complete. `Free_run` continues running regardless of the breakpoint.

**Number of stop bits**

Select whether to use `1` or `2` stop bits.

**Parity mode**

Type of parity to use. Available selections are `None`, `Odd` parity, or `Even` parity. `None` disables parity. `Odd` sets the parity bit to one if you have an odd number of ones in your bytes, such as 00110010. `Even` sets the parity bit to one if you have an even number of ones in your bytes, such as 00110011.

**Character length bits**

Length in bits of each transmitted or received character, set to 8 bits.

**Desired baud rate in bits/sec**

The desired baud rate specified by the user.

**Baud rate prescaler (BRR = (SCIHBAUD << 8) | SCILBAUD))**

The baud rate prescaler.

**Closest achievable baud rate (LSPCLK/(BRR+1)/8) in bits/sec**

The closest achievable baud rate calculated based on LSPCLK and BRR.

**Communication mode**

Select `Raw_data` or `Protocol` mode. Raw data is unformatted and sent whenever the transmitting side is ready to send, whether the receiving side is ready or not. Without a wait state, deadlock conditions do not occur. Data transmission is asynchronous. With this mode, it is possible the receiving side could miss data, but if the data is noncritical, using raw data mode can avoid blocking processes.

When you select protocol mode, some handshaking between host and processor occurs. The transmitting side sends $SND to indicate it is ready to transmit. The receiving side sends back $RDY to indicate it is ready to receive. The transmitting side then sends data and, when the transmission is completed, it sends a checksum.

Advantages to using protocol mode include:

- Avoids deadlock
- Determines whether data is received without errors (checksum)
- Determines whether data is received by processor
- Determines time consistency; each side waits for its turn to send or receive

**Note** Deadlocks can occur if one SCI Transmit block tries to communicate with more than one SCI Receive block on different COM ports when both are blocking (using protocol mode). Deadlocks cannot occur on the same COM port.

**Blocking mode**

If this option is set to True, system waits until data is available to read (when data length is reached). If this option is set to False, system checks FIFO periodically (in polling mode) for data to read. If data is present, the block reads and outputs the contents. When data is not present, the block outputs the last value and continues.

**Data byte order**

Select `Little Endian` or `Big Endian`, to match the endianness of the data being moved.

**Pin assignment (Tx)**

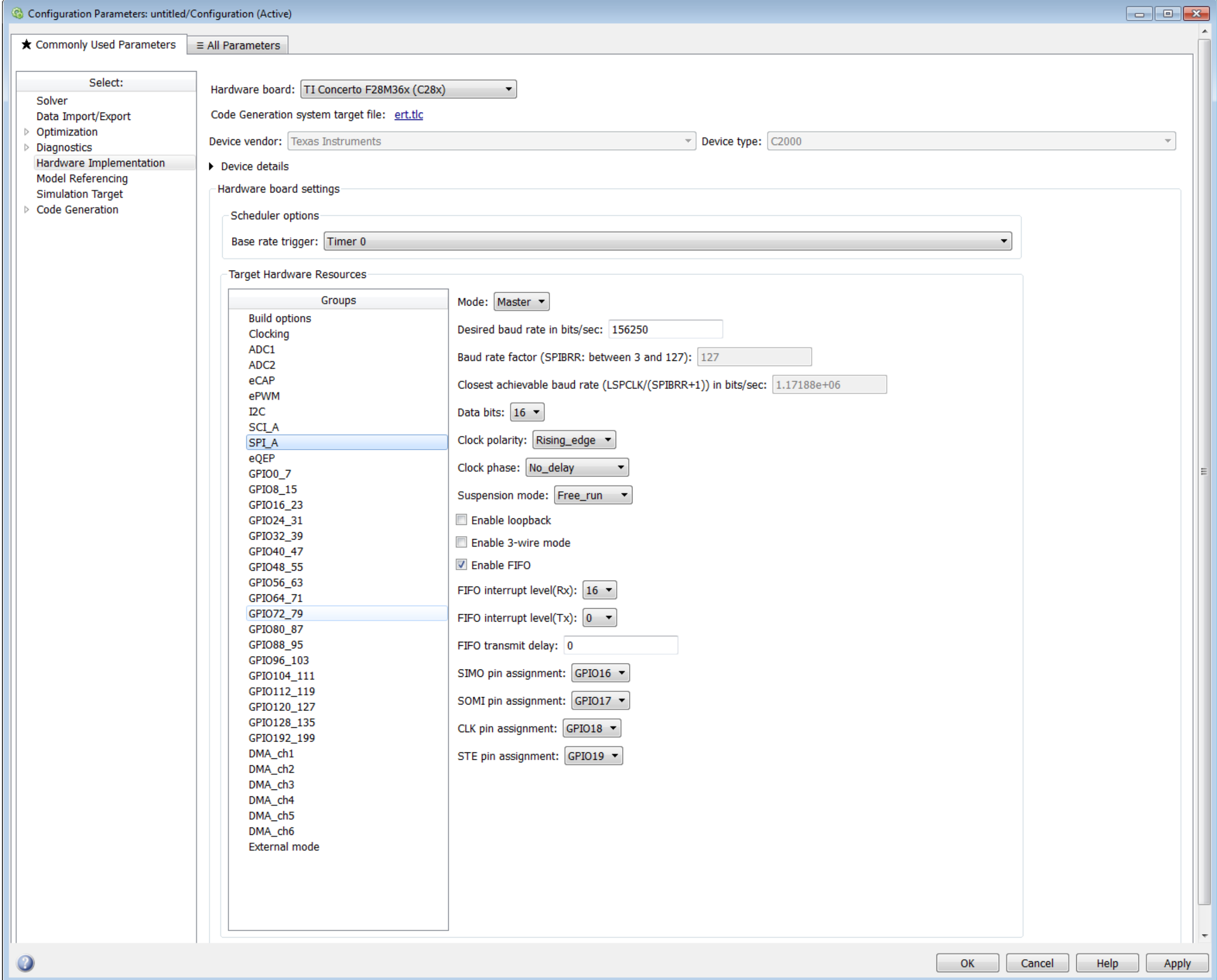
Assigns the SCI transmit pin to use with the SCI module.

**Pin assignment (Rx)**

Assigns the SCI receive pin to use with the SCI module.

**Note** The SCI_B and SCI_C are available only for TI F280x processors.

## C28x-SPI_A, C28x-SPI_B, C28x-SPI_C, C28x-SPI_D




The serial peripheral interface parameters you can set for the A module. These parameters are:

**Mode**

Set to `Master` or `Slave`.

**Desired baud rate in bits/sec**

The desired baud rate specified by the user.

**Baud rate factor (SPIBRR: between 3 and 127)**

To set the **Baud rate factor**, search for “Baud Rate Determination” and “SPI Baud Rate Register (SPIBRR) Bit Descriptions” in *TMS320x28xx, 28xxx DSP Serial Peripheral Interface (SPI) Reference Guide*, Literature Number: SPRU059, available on the Texas Instruments Web Site.

**Closest achievable baud rate (LSPCLK/(SPIBRR+1)) in bits/sec**

The closest achievable baud rate calculated based on LSPCLK and SPIBRR.

**Data bits**

Length in bits from `1` to `16` of each transmitted or received character. For example, if you select `8`, the maximum data that can be transmitted using SPI is $2^{8-1}$. If you send data greater than this value, the buffer overflows.

**Clock polarity**

Select `Rising_edge` or `Falling_edge`.

**Clock phase**

Select `No_delay` or `Delay_half_cycle`.

**Suspension mode**

Type of suspension to use when debugging your program with Code Composer Studio. When your program encounters a breakpoint, the selected suspension mode determines whether to perform the program instruction. Available options are `Hard_abort`, `Soft_abort`, and `Free_run`. `Hard_abort` stops the program immediately. `Soft_abort` stops when the current receive or transmit sequence is complete. `Free_run` continues running regardless of the breakpoint.

**Enable loopback**

Select this option to enable the loopback function for self-test and diagnostic purposes only. When this function is enabled, the Tx pin on a C28x DSP is internally connected to its Rx pin and can transmit data from its output port to its input port to check the integrity of the transmission.

**Enable 3-wire mode**

Enable SPI communication over three pins instead of the normal four pins.

**Enable FIFO**

Set true or false.

**FIFO interrupt level (Rx)**

Set level for receive FIFO interrupt.

**FIFO interrupt level (Tx)**

Set level for transmit FIFO interrupt.

**FIFO transmit delay**

Enter FIFO transmit delay (in processor clock cycles) to pause between data transmissions. Enter an integer.

**CLK pin assignment**

Assigns the SPI something (CLK) to a GPIO pin. Choices are `None` (default), `GPIO14`, or `GPIO26`.

CLK pin assignment is not available for TI Concerto F28M35x/F28M36x processors.

**SOMI pin assignment**

Assigns the SPI value (SOMI) to a GPIO pin. Choices are `None` (default), `GPIO13`, or `GPIO25`.

**STE pin assignment**

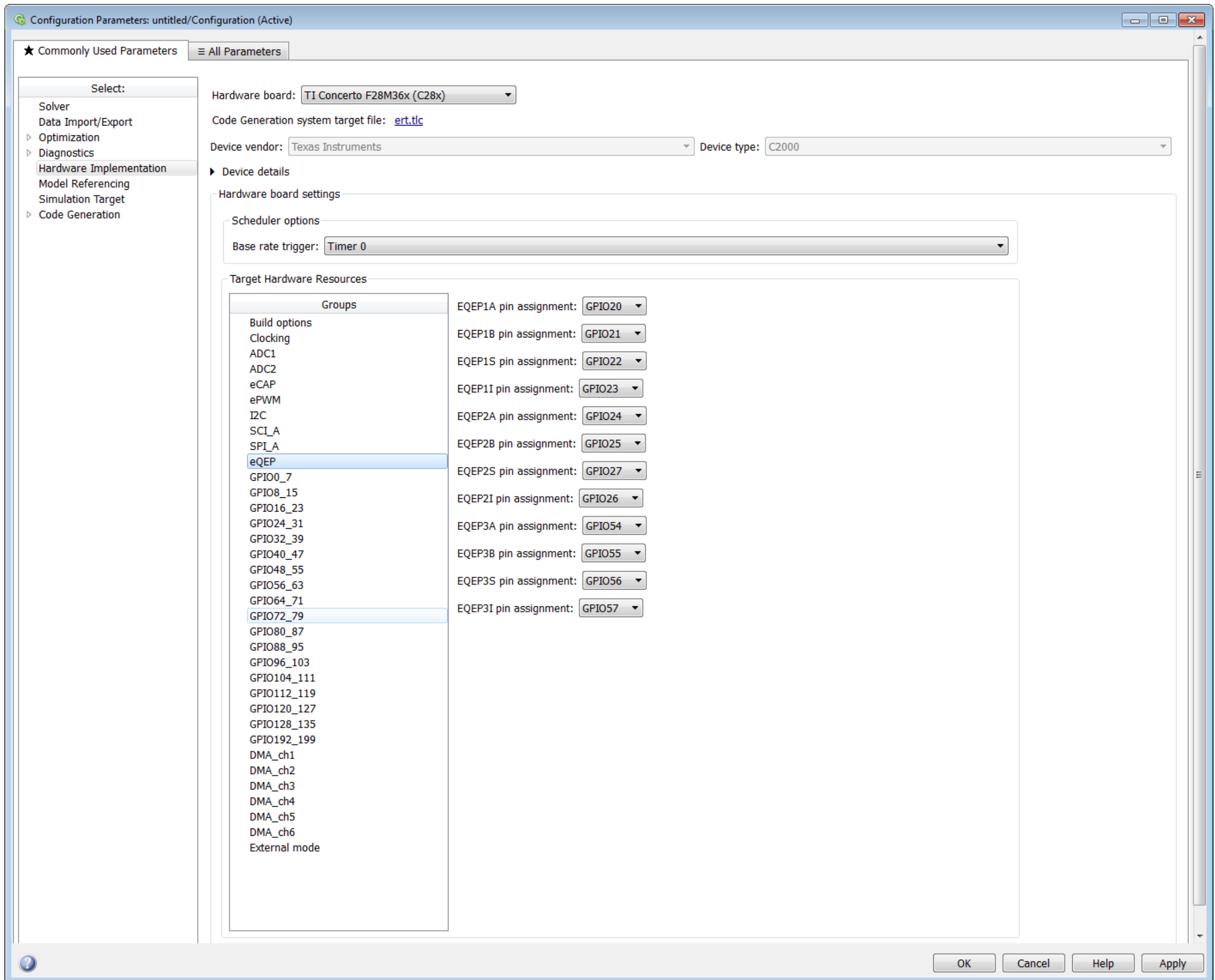
Assigns the SPI value (STE) to a GPIO pin. Choices are `None` (default), `GPIO15`, or `GPIO27`.

STE pin assignment is not available for TI Concerto F28M35x/ F28M36x processors.

**SIMO pin assignment**

Assigns the SPI something (SIMO) to a GPIO pin. Choices are `None` (default), `GPIO12`, or `GPIO24`.

## C28x-eQEP




Assigns eQEP pins to GPIO pins.

### EQEP1A pin assignment

Select an option from the list—`None`, `GPIO20`, `GPIO50`, `GPIO64`, or `GPIO106`.

**EQEP1B pin assignment**

Select an option from the list—None, GPIO21, GPIO51, GPIO65, or GPIO107.

**EQEP1S pin assignment**

Select an option from the list—None, GPIO22, GPIO52, GPIO66, or GPIO108.

**EQEP1I pin assignment**

Select an option from the list—None, GPIO23, GPIO53, GPIO67, or GPIO109.

**EQEP2A pin assignment**

Select an option from the list—None, GPIO24, GPIO54, GPIO66, or GPIO110. The pin numbers shown in the list vary based on the processor selected.

**EQEP2B pin assignment**

Select an option from the list—None, GPIO25, GPIO55, GPIO67, or GPIO111. The pin numbers shown in the list vary based on the processors selected.

**EQEP2S pin assignment**

Select an option from the list—None, GPIO27, GPIO31, GPIO57, GPIO67, or GPIO113.

**EQEP2I pin assignment**

Select an option from the list—None, GPIO26, GPIO30, GPIO56, GPIO64, or GPIO112.

**EQEP3A pin assignment**

Select an option from the list—None, GPIO54, or GPIO112. This parameter is available only for TI Concerto F28M36x (C28x) processors.

**EQEP3B pin assignment**

Select an option from the list—None, GPIO55, or GPIO113. This parameter is available only for TI Concerto F28M36x (C28x) processors.

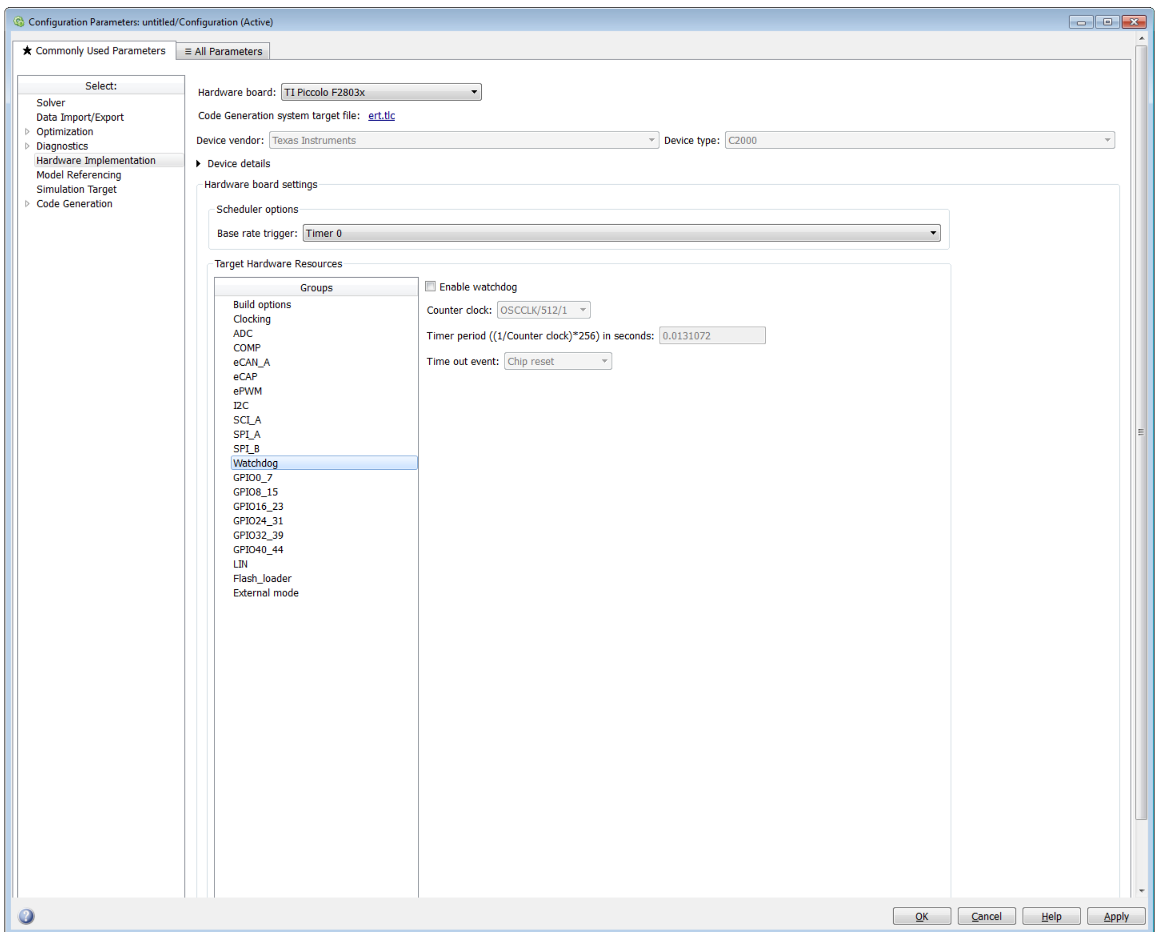
**EQEP3S pin assignment**

Select an option from the list—None, GPIO56, or GPI110.

**EQEP3I pin assignment**

Select an option from the list—None, GPIO57, or GPIO111.

# C28x-Watchdog




When enabled, if the software fails to reset the watchdog counter within a specified interval, the watchdog resets the processor or generates an interrupt. This feature enables the processor to recover from some fault conditions.

For more information, locate the *Data Manual* or *System Control and Interrupts Reference Guide* for your processor on the Texas Instruments Web site.

**Enable watchdog**

Enable the watchdog timer module.

This parameter corresponds to bit 6 (WDDIS) of the Watchdog Control Register (WDCR) and bit 0 (WDOVERRIDE) of the System Control and Status Register (SCSR).

**Counter clock**

Set the watchdog timer period relative to OSCCLK/512.

This parameter corresponds to bits 2–0 (WDPS) of the Watchdog Control Register (WDCR).

**Timer period ((1/Counter clock)*256) in seconds**

This field displays the timer period in seconds. This value automatically updates when you change the **Counter clock** parameter.

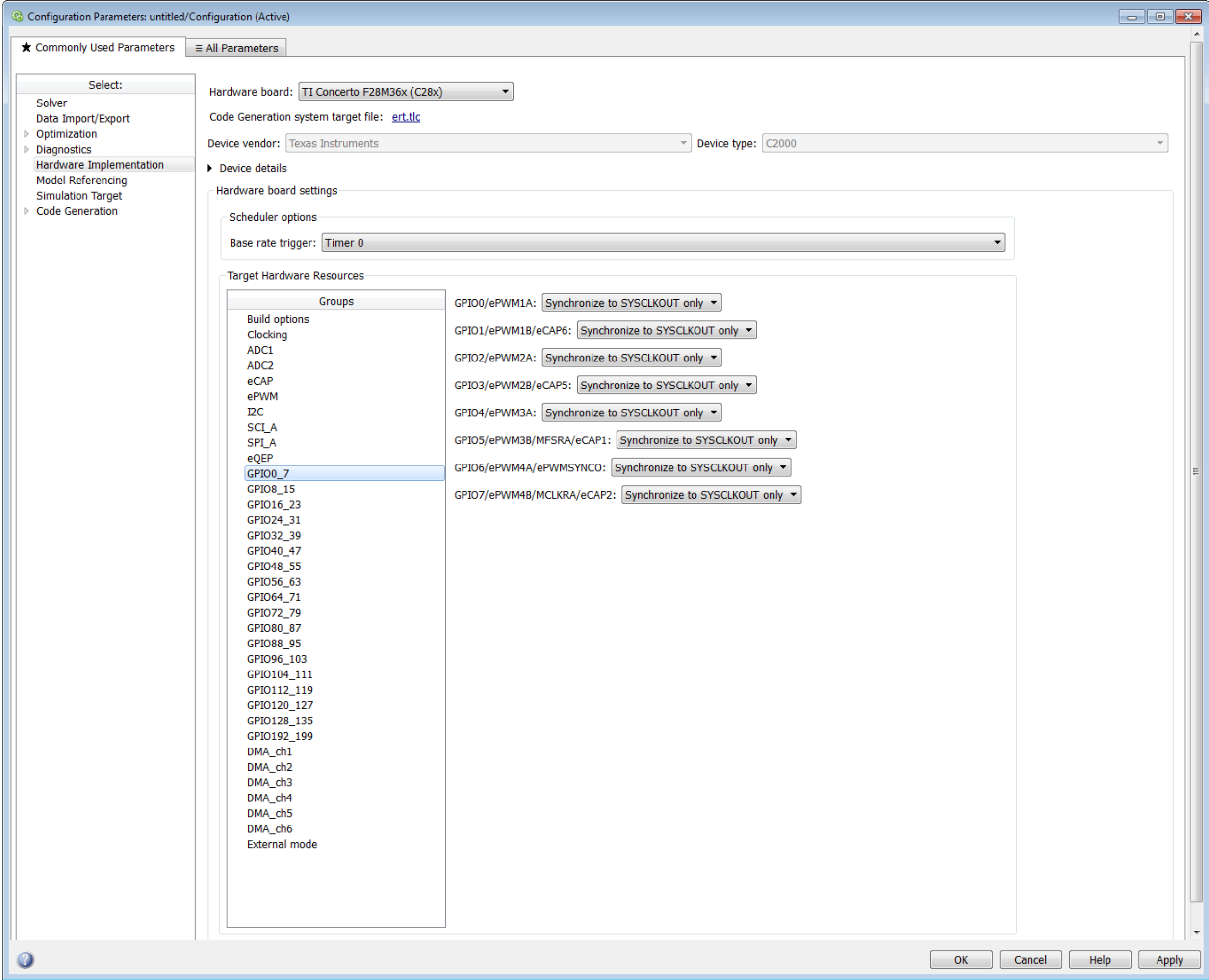
**Time out event**

Configure the watchdog to reset the processor or generate an interrupt when the software fails to reset the watchdog counter:

- Select `Chip reset` to generate a signal that resets the processor (WDRST signal) and disable the watchdog interrupt signal (WDINT signal).
- Select `Raise WD Interrupt` to generate a watchdog interrupt signal (WDINT signal) and disable the reset processor signal (WDRST signal). This signal can be used to wake the device from an IDLE or STANDBY low-power mode.

This parameter corresponds to bit 1 (WDENINT) of the System Control and Status Register (SCSR).

## C28x-GPIO



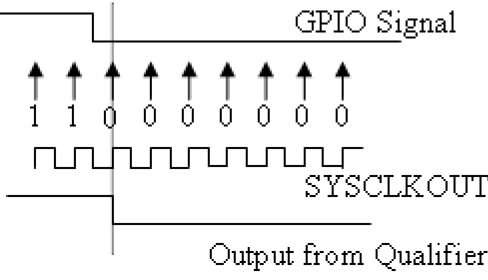

**GPIO**Use the GPIO pins for digital input or output by connecting to one of the three peripheral I/O ports.

The range of GPIO pins for different processors is given below:

| Processors | GPIO Pin Values |
|---|---|
| C281x | GPIOA, GPIOB, GPIOD, GPIOE, GPIOF, and GPIOG |
| F2803x | GPIO0–7, GPIO8–15, GPIO16–23, GPIO24–31, GPIO32–39, GPIO40–44 |
| F2806x | GPIO0–7, GPIO8–15, GPIO16–23, GPIO24–31, GPIO32–39, GPIO40–44, GPIO50–55, GPIO56–58 |
| F2823x, F2833x, and C2834x | GPIO0–7, GPIO8–15, GPIO16–23, GPIO24–31, GPIO32–39, GPIO40–47, GPIO48–55, GPIO56–63 |
| C2801x, F2802x, F28044, F280x | GPIO0–7, GPIO8–15, GPIO16–23, GPIO24–31, GPIO32–34 |
| F28M35x (C28x) | GPIO0–7, GPIO8–15, GPIO16–23, GPIO24–31, GPIO32–39, GPIO40–47, GPIO48_55, GPIO56–63, GPIO68–71, GPIO128–135 |
| F28M36x (C28x) | GPIO0–7, GPIO8–15, GPIO16–23, GPIO24–31, GPIO40–47, GPIO48–55, GPIO56–63, GPIO64–71, GPIO72–79, GPIO80–87, GPIO88–95, GPIO96–103, GPIO104–111, GPIO112–119, GPIO120–127, GPIO128–135, GPIO192–199. |

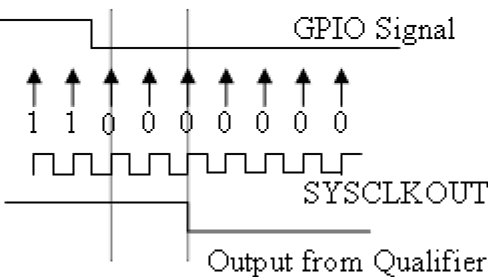
Each pin selected for input offers four signal qualification types:

- `Sync to SYSCLKOUT only` — This setting is the default for all pins at reset. Using this qualification type, the input signal is synchronized to the system clock SYSCLKOUT. The following figure shows the input signal measured on each tick of the system clock, and the resulting output from the qualifier.

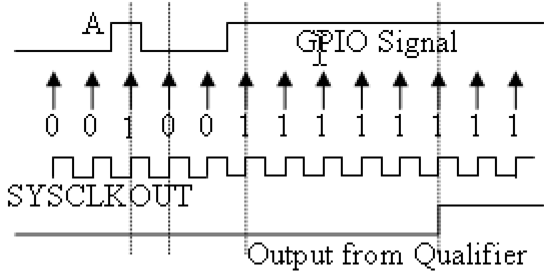



- `Qualification using 3 samples` — This setting requires three consecutive cycles of the same value for the output value to change. The following figure shows that, in the third cycle, the GPIO value changes to `0`, but the qualifier output is still `1`

because it waits for three consecutive cycles of the same GPIO value. The next three cycles all have a value of `0`, and the output from the qualifier changes to `0` immediately after the third consecutive value is received.



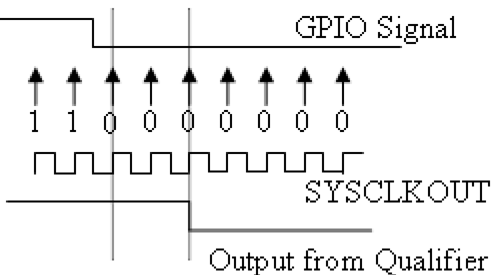

- `Qualification using 6 samples` — This setting requires six consecutive cycles of the same GPIO input value for the output from the qualifier to change. In the following figure, the glitch **A** does not alter the output signal. When the glitch occurs, the counting begins, but the next measurement is low again, so the count is ignored. The output signal does not change until six consecutive samples of the high signal are measured.



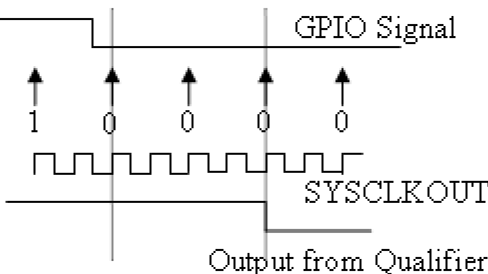

**Qualification sampling period prescaler**

Visible only when a setting for **Qualification type for GPIO [pin#]** is selected. The qualification sampling period prescaler, with possible values of `0` to `255`, calculates the frequency of the qualification samples or the number of system clock ticks per sample. The formula for calculating the qualification sampling frequency is SYSCLKOUT/(2 * Prescaler), except for zero. When **Qualification sampling period prescaler=**`0`, a sample is taken every SYSCLKOUT clock tick. For example, a prescale setting of `0` means that a sample is taken on each SYSCLKOUT tick.

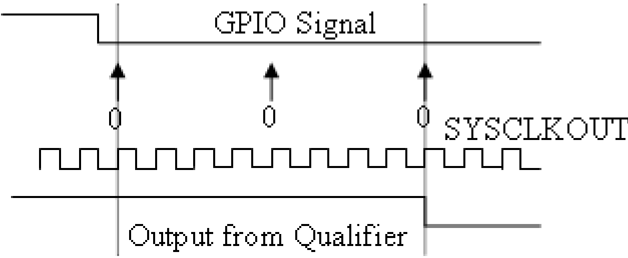
The following figure shows the SYSCLKOUT ticks, a sample taken every clock tick, and the **Qualification type** set to `Qualification using 3 samples`. In this case, the **Qualification sampling period prescaler=**`0`:




In the next figure **Qualification sampling period prescaler=**`1`. A sample is taken every two clock ticks, and the **Qualification type** is set to `Qualification using 3 samples`. The output signal changes much later than if **Qualification sampling period prescaler=**`0`.




In the following figure, **Qualification sampling period prescaler=**`2`. Thus , a sample is taken every four clock ticks, and the **Qualification type** is set to `Qualification using 3 samples`.




- `Asynchronous`

  Using this qualification type, the signal is synchronized to an asynchronous event initiated by software (CPU) via control register bits.

**Qualification sampling period**

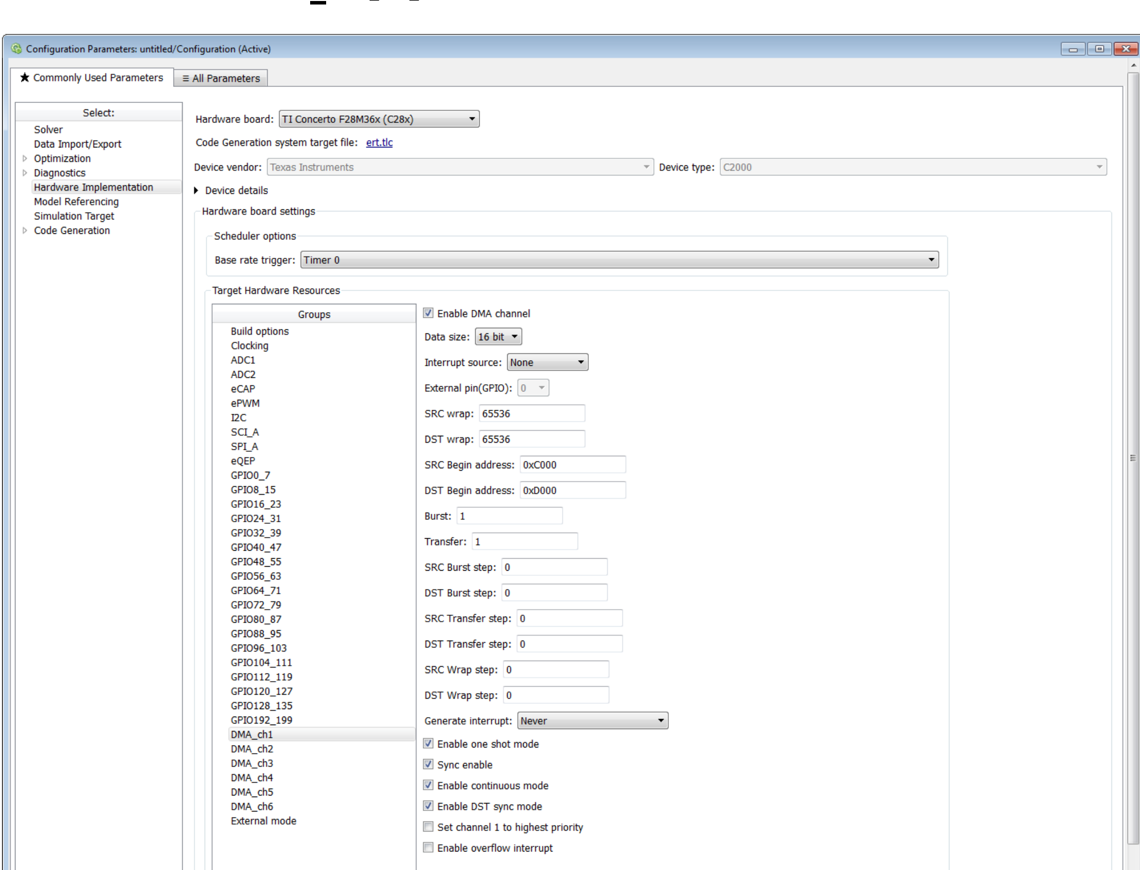
Enter the qualification sampling period.

**GPIOA, GPIOB, GPIOD, GPIOE input qualification sampling period**

**GPIO# Pull Up Disabled**

Select this check box to disable the GPIO pull up register. This option is available only for TI Concerto F28M35x/F28M36x processors.

## C28x-DMA_ch[#]




The Direct Memory Access module transfers data directly between peripherals and memory using a dedicated bus, increasing overall system performance.

You can individually enable and configure each DMA channel.

The DMA module services are event driven. Using the Interrupt source and **External pin (GPIO)** parameters, you can configure a wide range of peripheral interrupt event triggers.

For more information, consult the *TMS320x2833x, 2823x/ TMS320F28M35x/ TMS320F28M36x Direct Memory Access (DMA) Module Reference Guide*, Literature Number: SPRUFB8A/ SPRUH22F/ SPRUHE8B.

Also consult the *Increasing Data Throughput using the TMS320F2833x DSC DMA* training presentation (requires login), both available from the TI Web site.

**Enable DMA channel**

Enable this parameter to edit the configuration of a specific DMA channel.

This parameter does not have a corresponding bit or register.

**Data size**

Select the size of the data bit transfer: `16 bit` or `32 bit`.

The DMA read/write data buses are 32 bits wide. 32-bit transfers have twice the data throughput of a 16-bit transfer.

When providing DMA service to McBSP, set **Data size** to `16 bit`.

The following parameters are based on a 16-bit word size. If you set **Data size** to `32 bit`, double the value of the following parameters:

- Size: Burst
- Source: Burst step
- Source: Transfer step
- Source: Wrap step
- Destination: Burst step
- Destination: Transfer step
- Destination: Wrap step

**Data size** corresponds to bit 14 (DATASIZE) in the Mode Register (MODE).

**Interrupt source**

Select the peripheral interrupt that triggers a DMA burst for the specified channel.

Select SEQ1INT or SEQ2INT to configure the ADC interrupt as interrupt source.

Select XINT1, XINT2, or XINT13 to configure GPIO pin 0 to 31 as an external interrupt source. Select XINT3 to XINT7 to configure GPIO pin 32 to 63 as an external interrupt source.

For more information about configuring XINT, consult the following references:

- *TMS320x2833x, 2823x External Interface (XINTF) User's Guide*, Literature Number: SPRU949, available on the TI Web site.
- *TMS320x2833x System Control and Interrupts*, Literature Number: SPRUFB0, available on the TI Web site.
- *TMS320F28M35x/ TMS320F28M36x*, Literature Number: SPRUH22F/ SPRUHE8B available on the TI Web site.
- The C280x/C2802x/C2803x/C2805x/C2806x/C2833x/C2834x/F28M3x/F2807x/F2837xD/F2837xS/F28004x GPIO Digital Input and C280x/C2802x/C2803x/C2805x/C2806x/C2833x/C2834x/F28M3x/F2807x/F2837xD/F2837xS/F28004x GPIO Digital Output block reference sections.

Drop-down menu items from TINT0 to MREVTB may require manual configuration.

Select ePWM1SOCA through ePWM6SOCB to configure the ePWM interrupt as an interrupt source. Note that not all revisions of the TMS320F2833x silicon provide ePWM interrupts as sources for DMA transfers. For more information about silicon revisions consult the following reference:

*TMS320x2833x, 2823x Silicon Errata/ TMS320F28M35x/ TMS320F28M36x*, Literature Number: SPRZ272/ SPRUH22F/ SPRUHE8B, available on the TI Web site.

The **Interrupt source** parameter corresponds to bit 4-0 (PERINTSEL) in the Mode Register (MODE).

**External pin(GPIO)**

When you set **Interrupt source** is set to an external interface (XINT[#]), specify the GPIO pin number from which the interrupt originates.

This parameter corresponds to the GPIO XINTn, XNMI Interrupt Select (GPIOXINTnSEL, GPIOXNMISEL) Registers.

For more information, consult the *TMS320x2833x / TMS320F28M35x/ TMS320F28M36x System Control and Interrupts Reference Guide*, Literature Number SPRUFB0/ SPRUH22F/ SPRUHE8B available from the TI Web site.

**SRC wrap**

Specify the number of bursts before returning the current source address pointer to the **Source Begin Address** value. To disable wrapping, enter a value for **SRC wrap** that is greater than the **Transfer** value.

This parameter corresponds to bits 15-0 (SRC_WRAP_SIZE) in the Source Wrap Size Register (SRC_WRAP_SIZE).

**DST wrap**

Specify the number of bursts before returning the current destination address pointer to the **Destination Begin Address** value. To disable wrapping, enter a value for **DST wrap** that is greater than the **Transfer** value.

This parameter corresponds to bits 15-0 (DST_WRAP_SIZE) in the Destination Wrap Size Register (DST_WRAP_SIZE).

**SRC Begin address**

Set the starting address for the current source address pointer. The DMA module points to this address at the beginning of a transfer and returns to it as specified by the **SRC wrap** parameter.

This parameter corresponds to bits 21-0 (BEGADDR) in the Active Source Begin Register (SRC_BEG_ADDR).

**DST Begin address**

Set the starting address for the current destination address pointer. The DMA module points to this address at the beginning of a transfer and returns to it as specified by the **DST wrap** parameter.

This parameter corresponds to bits 21-0 (BEGADDR) in the Active Destination Begin Register (DST_BEG_ADDR).

**Burst**

Specify the number of 16-bit words in a burst, from 1 to 32. The DMA module must complete a burst before it can service the next channel.

Set the **Burst** value for the peripheral the DMA module is servicing. For the ADC, the value equals the number of ADC registers used, up to 16. For multichannel buffered serial ports (McBSP), which lack FIFOs, the value is 1.

For RAM, the value can range from 1 to 32.

This parameter corresponds to bits 4-0 (BURSTSIZE) in the Burst Size Register (BURST_SIZE).

**Note** This parameter is based on a 16-bit word size. If you set **Data size** to `32 bit`, double the value of this parameter.

**Transfer**

Specify the number of bursts in a transfer, from 1 to 65536.

This parameter corresponds to bits 15-0 (TRANSFERSIZE) in the Transfer Size Register (TRANSFER_SIZE).

**SRC Burst step**

Set the number of 16-bit words by which to increment or decrement the current address pointer before the next burst. Enter a value from –4096 (decrement) to 4095 (increment).

To disable incrementing or decrementing the address pointer, set **Burst step** to 0. For example, because McBSP does not use FIFO, configure DMA to maintain the sequence of the McBSP data by moving each word of the data individually.

Accordingly, when you use DMA to transmit or receive McBSP data, set **Burst size** to 1 word and **Burst step** to 0.

This parameter corresponds to bits 15-0 (SRCBURSTSTEP) in the Source Burst Step Size Register (SRC_BURST_STEP).

---

**Note** This parameter is based on a 16-bit word size. If you set **Data size** to `32 bit`, double the value of this parameter.

---

**DST Burst step**

Set the number of 16-bit words by which to increment or decrement the current address pointer before the next burst. Enter a value from –4096 (decrement) to 4095 (increment).

To disable incrementing or decrementing the address pointer, set **Burst step** to 0. For example, because McBSP does not use FIFO, configure DMA to maintain the sequence of the McBSP data by moving each word of the data individually. Accordingly, when you use DMA to transmit or receive McBSP data, set **Burst size** to 1 word and **Burst step** to 0.

This parameter corresponds to bits 15-0 (DSTBURSTSTEP) in the Destination Burst Step Size Register (DST_BURST_STEP).

**Note** This parameter is based on a 16-bit word size. If you set **Data size** to `32 bit`, double the value of this parameter.

### SRC Transfer step

Set the number of 16-bit words by which to increment or decrement the current address pointer before the next transfer. Enter a value from –4096 (decrement) to 4095 (increment).

To disable incrementing or decrementing the address pointer, set **Transfer step** to 0.

This parameter corresponds to bits 15-0 (SRCTRANSFERSTEP) Source Transfer Step Size Register (SRC_TRANSFER_STEP).

If DMA is configured to perform memory wrapping (if **SRC wrap** is enabled) the corresponding source **Transfer step** does not alter the results.

**Note** This parameter is based on a 16-bit word size. If you set **Data size** to `32 bit`, double the value of this parameter.

### DST Transfer step

Set the number of 16-bit words by which to increment or decrement the current address pointer before the next transfer. Enter a value from –4096 (decrement) to 4095 (increment).

To disable incrementing or decrementing the address pointer, set **Transfer step** to 0.

This parameter corresponds to bits 15-0 (DSTTRANSFERSTEP) Destination Transfer Step Size Register (DST_TRANSFER_STEP).

If DMA is configured to perform memory wrapping (if **DST wrap** is enabled) the corresponding destination **Transfer step** does not alter the results.

**Note** This parameter is based on a 16-bit word size. If you set **Data size** to `32 bit`, double the value of this parameter.

### SRC Wrap step

Set the number of 16-bit words by which to increment or decrement the SRC_BEG_ADDR address pointer when a wrap event occurs. Enter a value from –4096 (decrement) to 4095 (increment).

This parameter corresponds to bits 15-0 (WRAPSTEP) in the Source Wrap Step Size Registers (SRC_WRAP_STEP).

---

**Note** This parameter is based on a 16-bit word size. If you set **Data size** to `32 bit`, double the value of this parameter.

---

### DST Wrap step

Set the number of 16-bit words by which to increment or decrement the DST_BEG_ADDR address pointer when a wrap event occurs. Enter a value from –4096 (decrement) to 4095 (increment).

This parameter corresponds to bits 15-0 (WRAPSTEP) in the Destination Wrap Step Size Registers (DST_WRAP_STEP).

---

**Note** This parameter is based on a 16-bit word size. If you set **Data size** to `32 bit`, double the value of this parameter.

---

### Generate interrupt

Enable this parameter to have the DMA channel send an interrupt to the CPU via the PIE at the beginning or end of a data transfer.

This parameter corresponds to bit 15 (CHINTE) and bit 9 (CHINTMODE) in the Mode Register (MODE).

### Enable one shot mode

Enable this parameter to have the DMA channel complete an entire *transfer* in response to an interrupt event trigger.

This option allows a single DMA channel and peripheral to dominate resources, and may streamline processing, but it also creates the potential for resource conflicts and delays.

Disable this parameter to have DMA complete one *burst* per channel per interrupt.

### Sync enable

When **Interrupt source** is set to SEQ1INT, enable this parameter to reset the DMA wrap counter when it receives the ADCSYNC signal from SEQ1INT. This way, the wrap counter and the ADC channels remain synchronized with each other.

If **Interrupt source** is not set to SEQ1INT, **Sync enable** does not alter the results.

This parameter corresponds to bit 12 (SYNCE) of the Mode Register (MODE).

### Enable continuous mode

Select this parameter to leave the DMA channel enabled upon completing a transfer. The channel will wait for the next interrupt event trigger.

Clear this parameter to disable the DMA channel upon completing a transfer. The DMA module disables the DMA channel by clearing the RUNSTS bit in the CONTROL register when it completes the transfer. To use the channel again, first reset the RUN bit in the CONTROL register.

### Enable DST sync mode

When **Sync enable** is enabled, enabling this parameter resets the destination wrap counter (DST_WRAP_COUNT) when the DMA module receives the SEQ1INT interrupt/ADCSYNC signal.

Disabling this parameter resets the source wrap counter (SCR_WRAP_COUNT) when the DMA module receives the SEQ1INT interrupt/ADCSYNC signal.

This parameter is associated with bit 13 (SYNCSEL) in the Mode Register (MODE).

### Set channel 1 to highest priority

This parameter is only available for DMA_ch1.

Enable this setting when DMA channel 1 is configured to handle high-bandwidth data, such as ADC data, and the other DMA channels are configured to handle lower-priority data.

When enabled, the DMA module services each enabled channel sequentially until it receives a trigger from channel 1.

Upon receiving the trigger, DMA interrupts its service to the current channel at the end of the current word, services the channel 1 burst that generated the trigger, and then continues servicing the current channel at the beginning of the next word.

Disable this channel to give each DMA channel equal priority, or if DMA channel 1 is the only enabled channel.

When disabled, the DMA module services each enabled channel sequentially.

This parameter corresponds to bit 0 (CH1PRIORITY) in the Priority Control Register 1 (PRIORITYCTRL1).

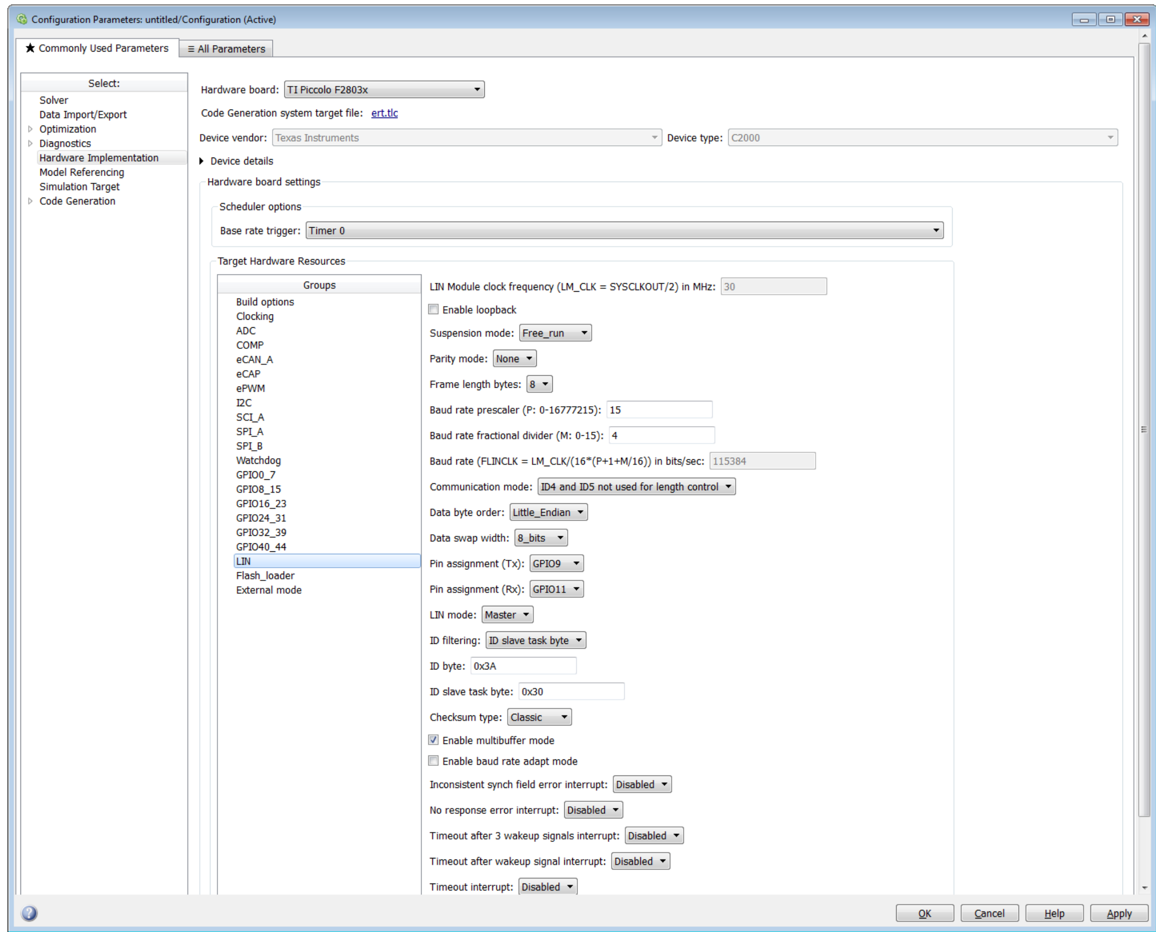
**Enable overflow interrupt**

Enable this parameter to have the DMA channel send an interrupt to the CPU via PIE if the DMA module receives a peripheral interrupt while a previous interrupt from the same peripheral is waiting to be serviced.

This parameter is typically used for debugging during the development phase of a project.

The **Enable overflow interrupt** parameter corresponds to bit 7 (OVRINTE) of the Mode Register (MODE), and involves the Overflow Flag Bit (OVRFLG) and Peripheral Interrupt Trigger Flag Bit (PERINTFLG).

# C28x-LIN




For detailed information on the LIN module, see *TMS320F2803x Piccolo Local Interconnect Network (LIN) Module*, Literature Number SPRUGE2, available at the Texas Instruments Web site.

The following options configure all LIN Transmit and LIN Receive blocks within a model.

### LIN Module clock frequency (LM_CLK = SYSCLKOUT/2) in MHz

Displays the frequency of the LIN module clock in MHz.

**Enable loopback**

To enable LIN loopback testing, select this option. While this option is enabled, the LIN module does the following:

- Internally redirects the LINTX output to the LINRX input.
- Puts the external LINTX pin into high state.
- Puts the external LINRX pin into a high impedance state.

The default is disabled (unchecked).

**Suspension mode**

Use this option to configure how the LIN state machine behaves while you debug the program on an emulator. If you select `Hard_abort`, entering LIN debug mode halts the transmissions and counters.

The transmissions and counters resume when you exit LIN debug mode. If you select `Free_run`, entering LIN debug mode allows the current transmit and receive functions to complete.

The default is `Free_run`.

**Parity mode**

Use this option to configure parity checking:

- To disable parity checking, select `None`.
- To enable odd parity checking, select `Odd`.
- To enable even parity checking, select `Even`.

The default is `None`.

In order for **ID parity error interrupt** in the LIN Receive block to generate interrupts, also enable **Parity mode**.

**Frame length bytes**

Set the number of data bytes in the response field, from 1 to 8 bytes.

The default is `8` bytes.

**Baud rate prescaler (P: 0-16777215)**

To set the LIN baud rate manually, enter a prescaler value, from 0 to 16777215. Click **Apply** to update the **Baud rate** display.

The default is 15.

For more information, consult the “Baud Rate” topic in the TI document, *TMS320F2803x Piccolo Local Interconnect Network (LIN) Module*, Literature Number SPRUGE2.

**Baud rate fractional divider (M: 0–15)**

To set the LIN baud rate manually, enter a fractional divider value, from 0 to 15. Click **Apply** to update the **Baud rate** display.

The default is 4.

For more information, consult the “Baud Rate” topic in the TI document, *TMS320F2803x Piccolo Local Interconnect Network (LIN) Module*, Literature Number SPRUGE2.

**Baud rate (FLINCLK = LM_CLK/(16*(P+1+M/16)) in bits/sec**

This field displays the baud rate. For more information, see “Setting the LIN baud rate”.

**Communication mode**

Enable or disable the LIN module from using the ID-field bits ID4 and ID5 for length control.

The default is `ID4 and ID5 not used for length control`

**Data byte order**

Set the “endianness” of the LIN message data bytes to `Little_Endian` or `Big_Endian`.

The default is `Little_Endian`.

**Data swap width**

Select `8_bits` or `16_bits`. If you set **Data byte order** to `Big_Endian`, the only available option for **Data swap width** is `8_bits`.

**Pin assignment (Tx)**

Map the LINTX output to a specific GPIO pin.

The default is `GPIO9`.

**Pin assignment (Rx)**

Map the LINRX input to a specific GPIO pin.

The default is `GPIO11`.

**LIN mode**

Put the LIN module in `Master` or `Slave` mode. The default is `Slave`.

In master mode, the LIN node can transmit queries and commands to slaves. In slave mode, the LIN module responds to queries or commands from a master node.

This option corresponds to the CLK_MASTER field in the SCI Global Control Register (SCIGCR1).

**ID filtering**

Select which type of mask filtering comparison the LIN module performs, `ID byte` or `ID slave task byte`.

If you select `ID byte`, the module uses the RECID and ID-BYTE fields in the LINID register to detect a match. If you select this option and enter 0xFF for LINMASK, the LIN module does not report matches.

I you select `ID slave task`. the module uses the RECID an ID-SlaveTask byte to detect a match. If you select this option and enter 0xFF for LINMASK, the LIN module reports matches.

The default is `ID slave task byte`.

**ID byte**

If you set **ID filtering** to `ID byte`, use this option to set the ID BYTE, also known as the “LIN mode message ID”.

In master mode, the CPU writes this value to initiate a header transmission. In slave mode, the LIN module uses this value to perform message filtering.

The default is `0x3A`.

**ID slave task byte**

If you set **ID filtering** to `ID slave task byte`, use this option to set the ID-SlaveTask BYTE. The LIN node compares this byte with the Received ID and determines whether to send a transmit or receive response.

The default is `0x30`.

**Checksum type**

Use this option to select the type of checksum. If you select `Classic`, the LIN node generates the checksum field from the data fields in the response.

If you select `Enhance`, the LIN node generates the checksum field from both the ID field in the header and data fields in the response. LIN 1.3 supports classic checksums only. LIN 2.0 supports both classic and enhanced checksums.

The default is `Classic`.

**Enable multibuffer mode**

When you enable (select) this check box, the LIN node uses transmit and receive buffers instead of just one register. This setting affects various other LIN registers, such as: checksums, framing errors, transmitter empty flags, receiver ready flags, transmitter ready flags.

The default is enabled (checked).

**Enable baud rate adapt mode**

The dialog box displays this option when you set **LIN mode** to `Slave`.

If you enable this option, the slave node automatically adjusts its baud rate to match that of the master node. For this feature to work, first set the **Baud rate prescaler** and **Baud rate fractional divider**.

If you disable this option, the LIN module sets a static baud rate based on the **Baud rate prescaler** and **Baud rate fractional divider**.

The default is disabled (unchecked).

**Inconsistent synch field error interrupt**

The dialog box displays this option when you set **LIN mode** to `Slave`.

If you enable this option, the slave node generates interrupts when it detects irregularities in the synch field. This option is only relevant if you enable **Enable adapt mode**.

The default is `Disabled`.

**No response error interrupt**

The dialog box displays this option when you set **LIN mode** to `Slave`.

If you enable this option, the LIN module generates an interrupt if it does not receive a complete response from the master node within a timeout period.

The default is `Disabled`.

**Timeout after 3 wakeup signals interrupt**

The dialog box displays this option when you set **LIN mode** to `Slave`.

When enabled, the slave node generates an interrupt when it sends three wakeup signals to the master node and does not receive a header in response. (The slave waits 1.5 seconds before sending another series of wakeup signals.)

This interrupt typically indicates the master node is having a problem recovering from low-power or sleep mode.

The default is `Disabled`.

**Timeout after wakeup signal interrupt**

The dialog box displays this option when you set **LIN mode** to `Slave`.

When enabled, the slave node generates an interrupt when it sends a wakeup signal to the master node and does not receive a header in response. (The slave waits 150 milliseconds before sending another series of wakeup signals.)

This interrupt typically indicates the master node is delayed recovering from low-power or sleep mode.

The default is `Disabled`.

**Timeout interrupt**

The dialog box displays this option when you set **LIN mode** to `Slave`.

When enabled, the slave node generates an interrupt after 4 seconds of inactivity on the LIN bus.

The default is `Disabled`.

**Wakeup interrupt**

The dialog box displays this option when you set **LIN mode** to `Slave`.

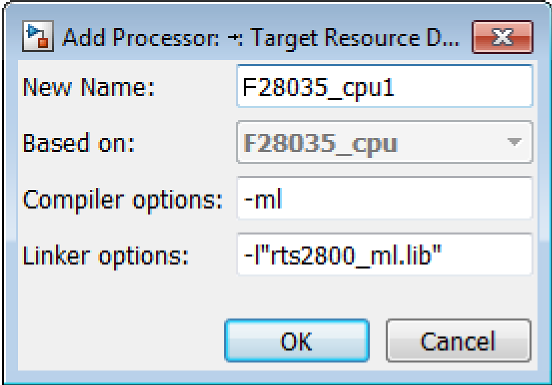
When you enable this option:

- In low-power mode, a LIN slave node generates a wakeup interrupt when it detects the falling edge of a wake-up pulse or a low level on the LINRX pin.
- A LIN slave node that is “awake” generates a wakeup interrupt if it receives a request to enter low-power mode while it is receiving.

- A LIN slave node that is “awake” does not generate a wakeup interrupt if it receives a wakeup pulse.

The default is `Disabled`.

## Add Processor Dialog Box




To add a new processor to the drop down list for the **Processors** option, click the **Add new** button on the **Board** pane. The software opens the Add Processor dialog box.

---

**Note** You can use this feature to create duplicates of existing processors with minor changes to the compiler and linker options. Avoid using this feature to create profiles for processors that are not already supported.

---

**New Name**

Provide a name to identify your new processor. Use a valid C string. The name you enter in this field appears on the list of processors after you add the new processor.

If you do not provide an entry for each parameter, the code generator returns an error message without creating a processor entry.

**Based On**

When you add a processor, the dialog box uses the settings from the currently selected processor as the basis for the new one. This parameter displays the currently selected processor.

**Compiler options**

Identifies the processor family of the new processor to the compiler. The string depends on the processor family or class.

For example, to set the compiler switch for a new C5509 processor, enter `-ml`. The following table shows the compiler switch string for supported processor families.

| Processor Family | Compiler Switch String |
|---|---|
| C62xx | |
| C64xx | |
| C67xx | |
| DM64x and DM64xx | |
| C55xx | `-ml` |
| C28xx, F28xx, R28xx, F28xxx | `-ml` |

**Linker options**

You can use this parameter to specify linker command options. The IDE uses these options to modify how it links project files when you build a project. To get information about specific linker options you can enter here, consult the documentation for your IDE.

## Target Hardware Resources Tab: Linux, VxWorks, or Windows

This tab appears when both of the following conditions are true:

- The **IDE/Tool Chain** parameter is set to a toolchain that builds generated code to run on Windows, Linux, or VxWorks (requires an Embedded Coder license).
- The **Operating System** parameter on the Board tab is set to `Windows`, `Linux`, or `VxWorks`

**Scheduling Mode**

When you select `free-running`, the model generates multi-threaded free-running code. Each rate in the model maps to a separate thread in the generated code. Multi-threaded code can potentially run faster than single threaded code.

When you select `real-time`, the model generates multi-threaded real-time code: Each rate in the Simulink model runs at the rate specified in the model. For example,

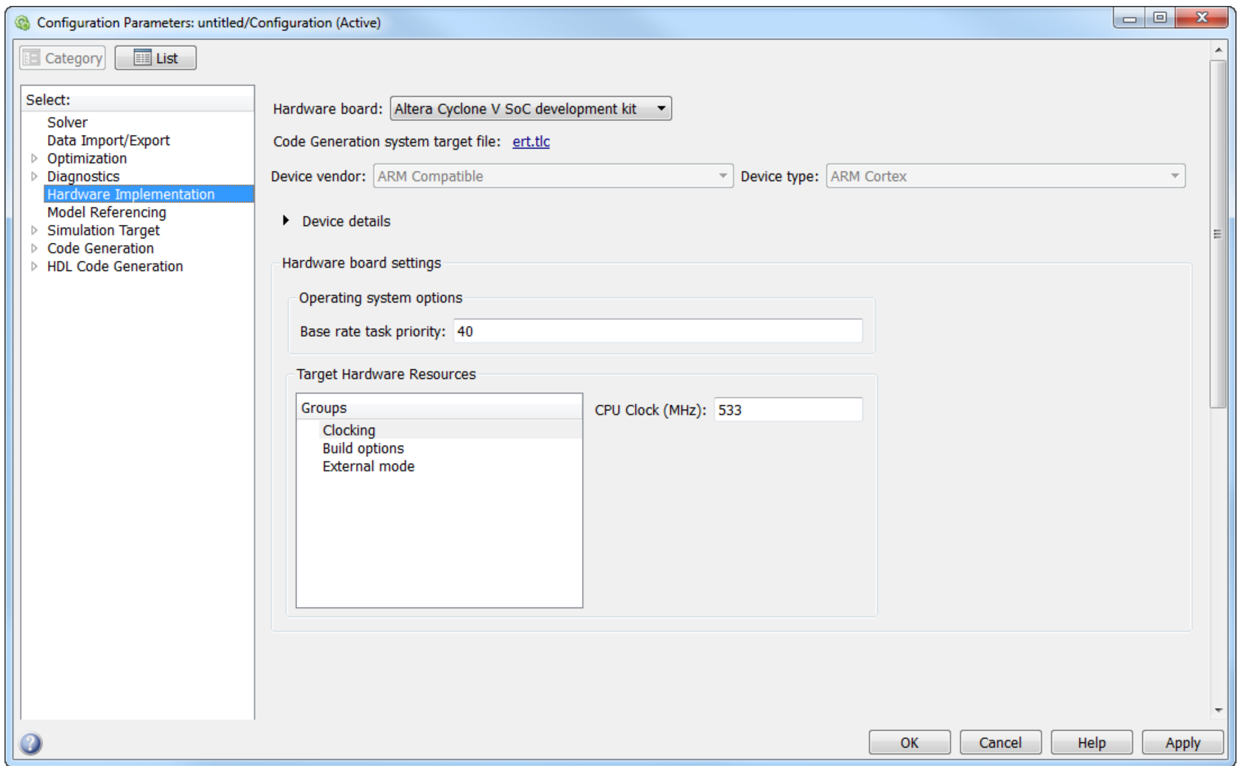
a 1-second rate runs at exactly 1-second intervals. The timing is provided by using a real-time clock.

**Base rate task priority**

The base rate in the model maps to a thread and runs as fast as possible. You can use the value of the base rate priority to set a static priority for the base rate task. By default, this rate is 40.

This parameter is not available for Windows.

# Hardware Implementation Pane: Altera Cyclone V SoC development kit, Arrow SoCKit development board




**In this section...**

"Hardware Implementation Pane Overview" on page 13-100
"Operating system options" on page 13-100
"Clocking" on page 13-100
"Build Options" on page 13-100
"External mode" on page 13-101

## Hardware Implementation Pane Overview

Configure the parameters for properties of the physical hardware, such as peripherals.

## Operating system options

**Base Rate Task Priority**

This parameter sets the static priority of the base rate task. By default, the priority is `40`.

## Clocking

**CPU Clock (MHz)**

Specify the CPU clock frequency of the real ARM Cortex processor on the target hardware.

## Build Options

**Build action**

Defines how the code generator responds when you press Ctrl+B to build your model.

**Settings**

**Default:** `Build, load and run`

`Build, load and run`

With this option, pressing **Ctrl+B** or clicking **Build Model**:

1. Generates code from the model.
2. Compiles and links the code into an executable with libraries.
3. Loads the executable and libraries into the target hardware.
4. Runs the executable in the target hardware.

`Build`

With this option, pressing **Ctrl+B** or clicking **Build Model**:

1. Generates code from the model.

2 Compiles and links the code into an executable with libraries.

This option does not load and run the executable in the target hardware.

## External mode

**Communication interface**

Select the transport layer external mode uses to exchange data between the host computer and the target hardware.

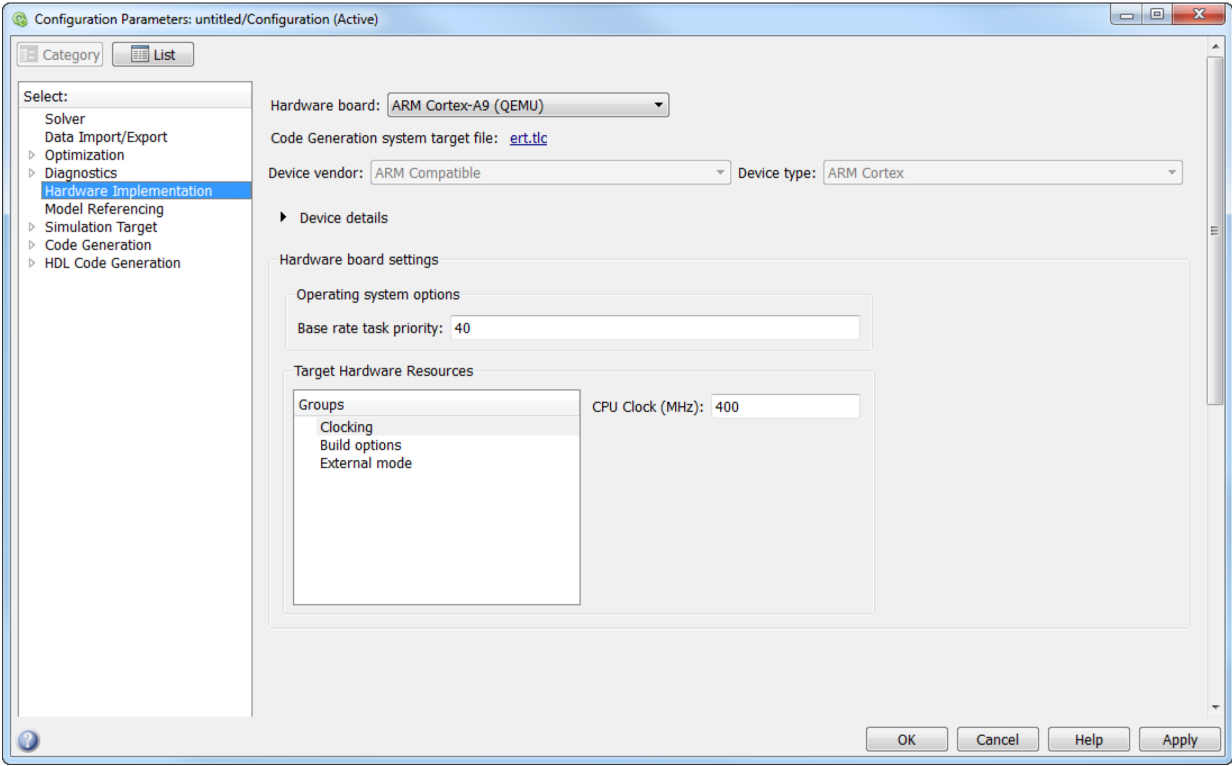
**IP address**

Specify the IP address of the target hardware. By default, this value is an environment variable, `$(SOCFPGA_IPADDRESS)`, which reuses the IP address from the most recent connection to the target hardware.

**Verbose**

Display verbose messages in the MATLAB Command Window.

# Hardware Implementation Pane: ARM Cortex-A9 (QEMU)




| In this section... |
| --- |
| "Hardware Implementation Pane Overview" on page 13-102 |
| "Operating system options" on page 13-103 |
| "Clocking" on page 13-103 |
| "Build Options" on page 13-103 |
| "External mode" on page 13-104 |

## Hardware Implementation Pane Overview

Configure the parameters for properties of the physical hardware, such as peripherals.

#### See Also

"Coder Target: Target Hardware Resources Tab Overview" on page 13-29

## Operating system options

**Base Rate Task Priority**

This parameter sets the static priority of the base rate task. By default, the priority is `40`.

## Clocking

**CPU Clock (MHz)**

Enter the actual CPU clock frequency in MHz. This value does not set the processor frequency.

The software uses this value to calculate the speed of the processor.

## Build Options

**Build action**

Defines how the code generator responds when you press Ctrl+B to build your model.

#### Settings

**Default:** `Build, load, and run`

`Build, load, and run`

With this option, pressing Ctrl+B or clicking Build Model:

1. Generates code from the model.
2. Compiles and links the code into an executable with libraries.
3. Loads the executable and libraries into the QEMU emulator.
4. Runs the executable in the QEMU emulator.

`Build`

With this option, pressing Ctrl+B or clicking Build Model:

1. Generates code from the model.
2. Compiles and links the code into an executable with libraries.

This option does not load and run the executable in the QEMU emulator.

#### Command-Line Information

**Parameter:** `buildAction`
**Type:** character vector
**Value:** `Build` | `Build_load_and_run` |
**Default:** `Build_load_and_run`

#### Recommended Settings

| Application | Setting |
| --- | --- |
| Debugging | `Build_load_and_run` |
| Traceability | No impact |
| Efficiency | No impact |
| Safety precaution | No impact |

#### See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

For more information about PIL and its uses, refer to the “Verifying Generated Code via Processor-in-the-Loop” topic.

## External mode

**Communication interface**

Select the transport layer external mode uses to exchange data between the host computer and the target hardware.

**Run external mode in a background thread**

Force the external mode engine in the generated code to execute in a background task.

This option is not recommended for models that use a very small time step, or which may encounter task overruns. These situations can cause Simulink to become unresponsive.

**Port**

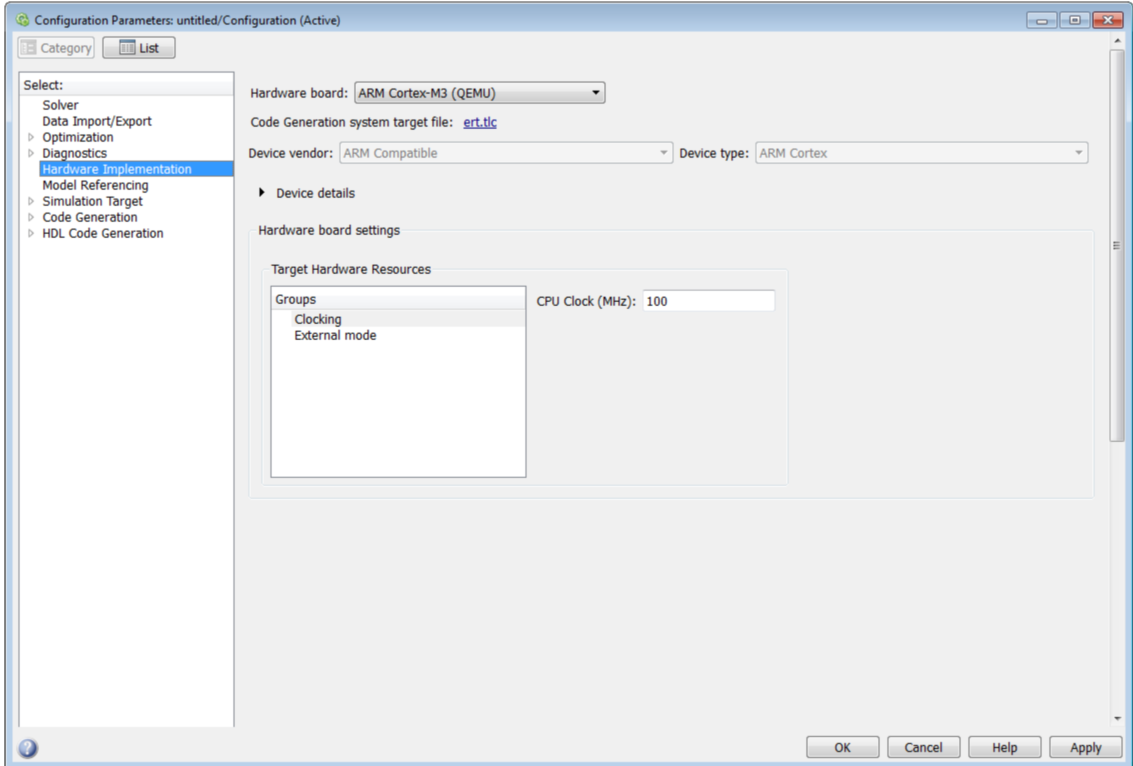
Set the value of the TCP/IP or WiFi port number, from 1024 to 65535. External mode uses this port for communications between the hardware board and host computer.

**Default:** 17725

**Verbose**

Display verbose messages in the MATLAB Command Window.

# Hardware Implementation Pane: ARM Cortex-M3 (QEMU)




| In this section... |
| --- |
| "Hardware Implementation Pane Overview" on page 13-106 |
| "Clocking" on page 13-107 |
| "External mode" on page 13-107 |

## Hardware Implementation Pane Overview

Configure the parameters for properties of the physical hardware, such as peripherals.

## Clocking

**CPU Clock (MHz)**

Specify the CPU clock frequency of a real ARM Cortex-M3 processor in MHz. The QEMU uses this value to emulate an ARM Cortex-M3 processor.

## External mode

**Communication interface**

Select the transport layer external mode uses to exchange data between the host computer and the target hardware.

**Run external mode in a background thread**

Force the external mode engine in the generated code to execute in a background task.

This option is not recommended for models that use a very small time step, or which may encounter task overruns. These situations can cause Simulink to become unresponsive.

**Port**

Set the value of the TCP/IP or WiFi port number, from 1024 to 65535. External mode uses this port for communications between the hardware board and host computer.

**Default:** 17725

**Verbose**

Display verbose messages in the MATLAB Command Window.

# Hardware Implementation Pane

**In this section...**

"Hardware Implementation Pane Overview" on page 13-108
"Build options" on page 13-110
"Clocking" on page 13-111
"DAC" on page 13-112
"UART0, UART1, UART2, and UART3" on page 13-113
"Ethernet" on page 13-116
"External mode" on page 13-118

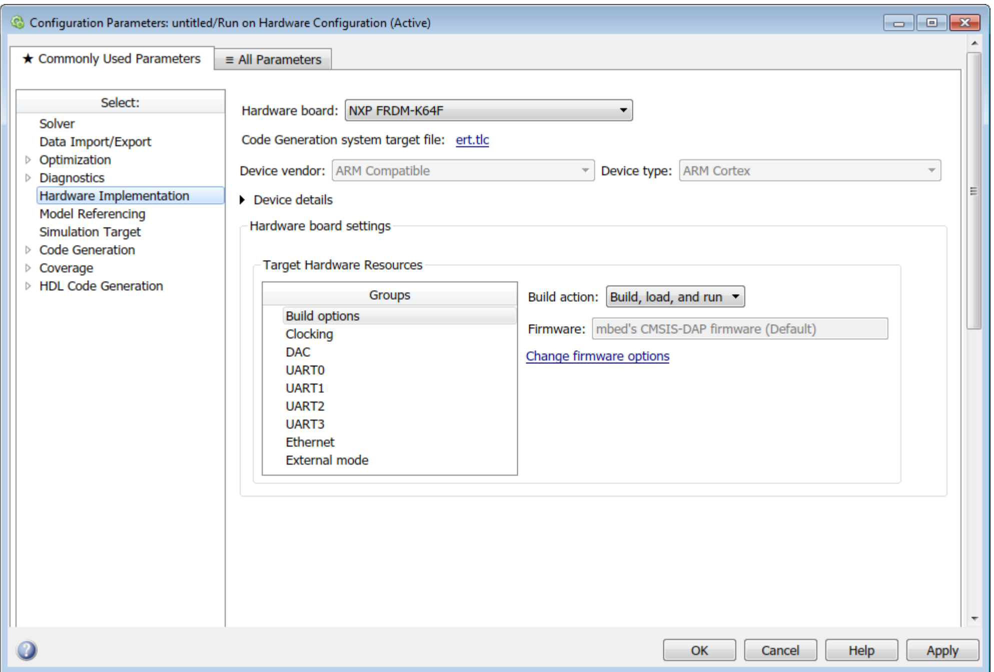
## Hardware Implementation Pane Overview

Specify the options for creating and running applications on target hardware.

### Configuration

Configure hardware board to run Simulink model.

1. In the Simulink Editor, select **Simulation** > **Model Configuration Parameters**.
2. In the Configuration Parameter dialog box, click **Hardware Implementation**.

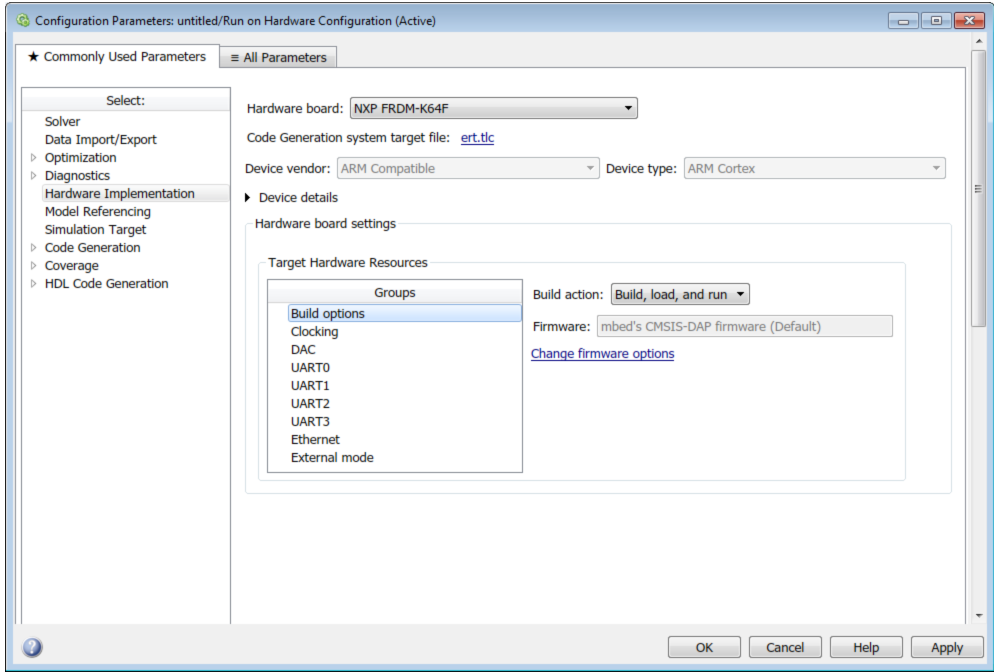



3. Set the **Hardware board** parameter to match your target hardware board.
4. Apply the changes.

### Base rate task priority

The value in this parameter defines the priority of the base rate task.

## Build options




To specify how the build process takes place during code generation, select build options.

### Build action

Specify whether you want only a build action or build, load, and run actions during code generation.

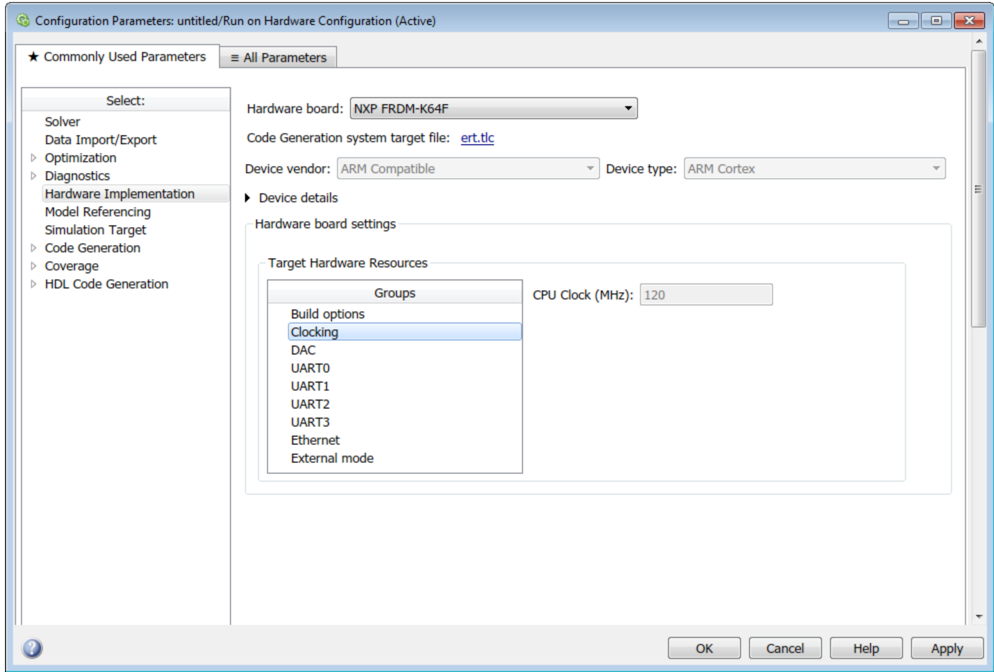
**Default:** `Build, load and run`

- `Build` — Select this option if you want to build the code during the build process.
- `Build, load and run` — Select this option to build, load, and run the generated code during the build process.

### Firmware

This is the firmware chosen during the setup to run your model on the FRDM-K64F board.

## Clocking




### CPU Clock (MHz)

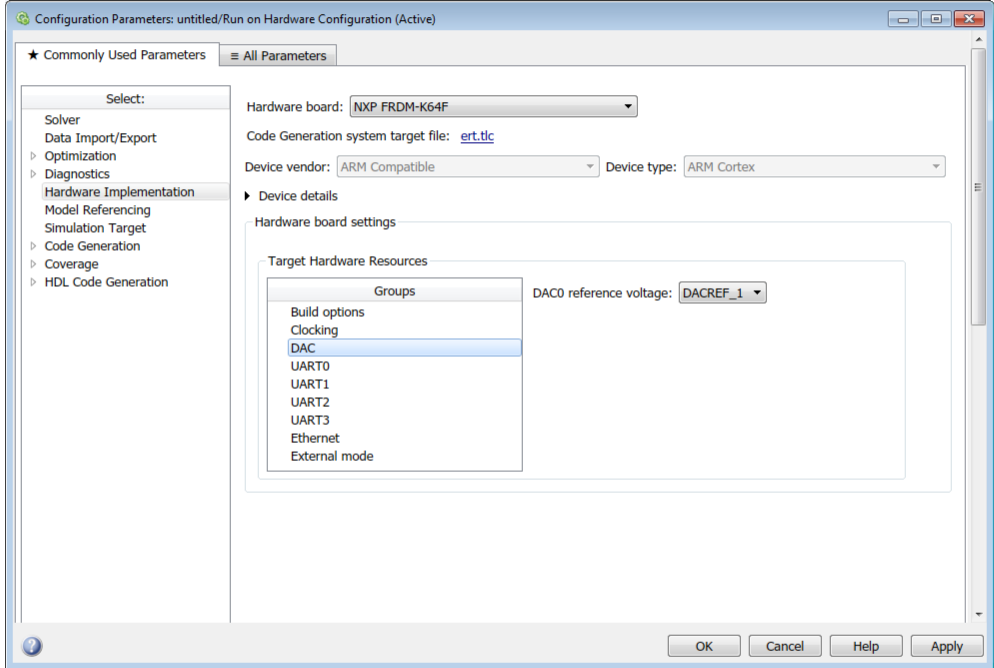
This option is for the CPU clock frequency of the FRDM-K64F processor on the target hardware.

---

**Note** This parameter appears dimmed. The value of this parameter is set to `120` MHz.

---

# DAC



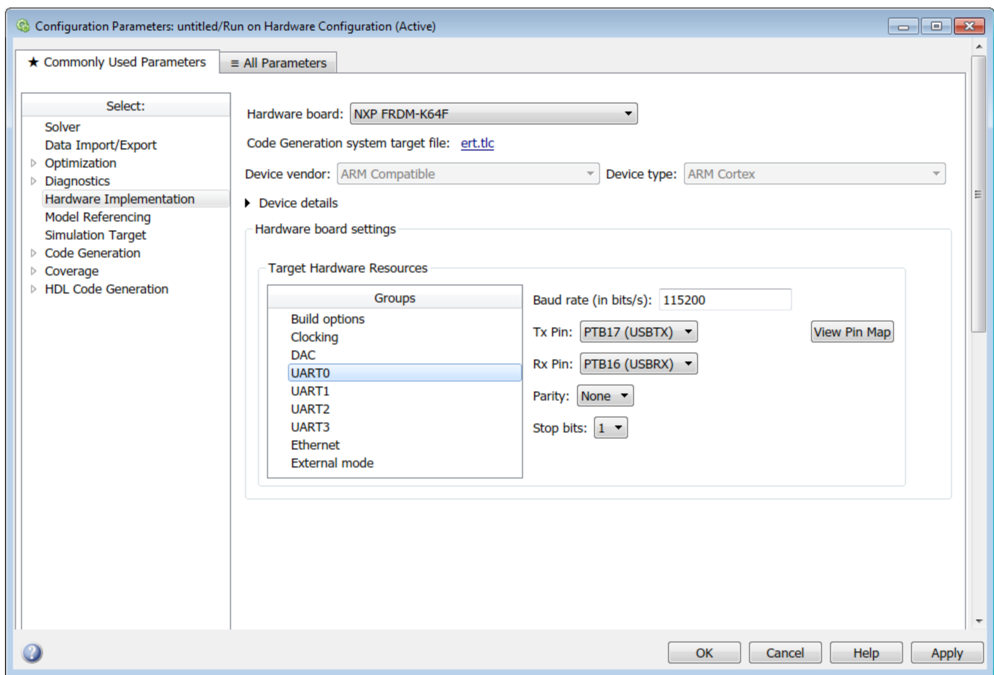

**DAC0 reference voltage**

This option is for the reference voltage of the DAC that you select.

**Default:** `DACREF_1`

`DACREF_2`

## UART0, UART1, UART2, and UART3




**Baud rate (in bits/s)**

Specify the baud for UARTx serial interfaces.

**Default:** `115200`

**Tx Pin**

Specify a transmitting pin of UARTx.

**UART0**

**Default:** `PTB17 (USBTX)`

`PTA2 (D5), PTD7, No connection`

**UART1**

**Default:** `PTC4 (D9)`

`No connection`

**UART2**

**Default:** `PTD3 (D12)`

`No connection`

**UART3**

**Default:** `PTC17 (D1)`

`PTB11 (A3), No connection`

**Rx Pin**

Specify a receiving pin of UARTx.

**UART0**

**Default:** `PTB16 (USBRX)`

`PTA1 (D3), PTD6, No connection`

**UART1**

**Default:** `PTC3 (D7)`

`No connection`

**UART2**

**Default:** `PTD2 (D11)`

`No connection`

**UART3**

**Default:** `PTC16 (D0)`

`PTB10 (A2), No connection`

**Parity**

Select the type of parity checking for serial communication. This option determines whether the UARTx generates and checks for even parity or odd parity.

**Default:** `None`

`Even, Odd`

- `None` — No parity checking
- `Even` — Even parity checking
- `Odd` — Odd parity checking

**Stop bits**

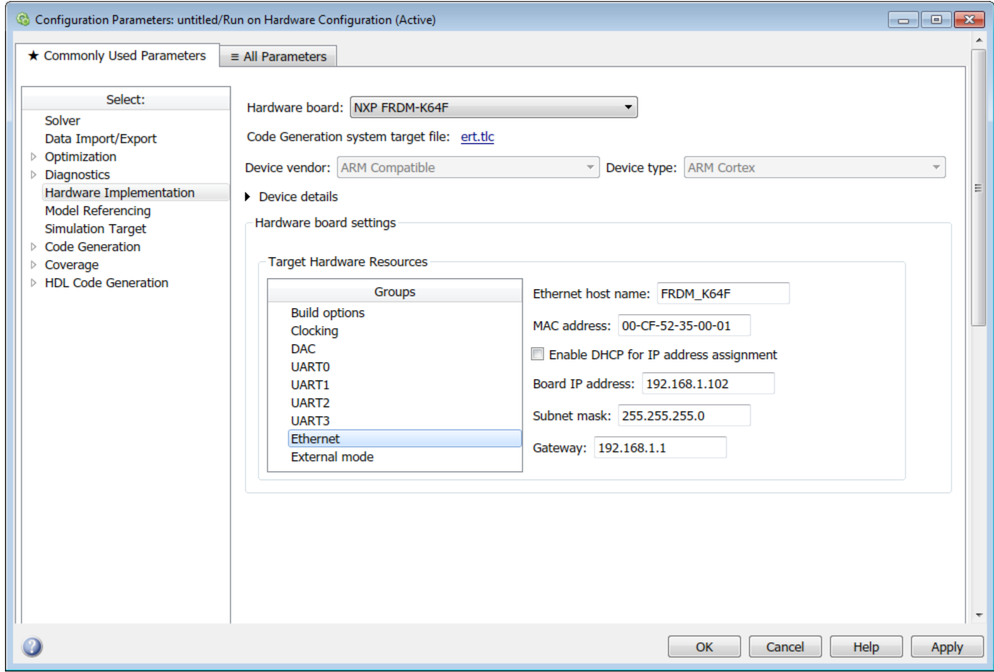
Select the number of Stop bits used to indicate end of a packet.

**Default:** `1`

`2`

- `1` — One stop bit is transmitted to indicate the end of a byte.
- `2` — Two stop bits are transmitted to indicate the end of a byte.

## Ethernet




### Ethernet host name

Specify the local host name. The local host is the board running the model.

**Default:** `FRDM_K64F`

### MAC address

Specify the Media Access Control (MAC) address, the physical network address of the board.

Under most circumstances, you do not need to change the MAC address. If you connect more than one board to a single computer so that each address is unique, change the MAC address. (You must have a separate network interface card (NIC) for each board.)

To change the MAC address, specify an address that is different from the address that belongs to any other device attached to your computer. To obtain the MAC address for

a specific board, refer to the label affixed to the board or consult the product documentation.

The MAC address must be in the six octet format. For example, `DE-AD-BE-EF-FE-ED`

**Default:** `00-CF-52-35-00-01`

**Enable DHCP for local IP address assignment**

Select this check box to configure the board to get an IP address from the local DHCP server on the network.

**Default:** `off`

`on`

**Board IP address**

Use this option for setting the IP address of the board. Change the IP address of your computer to a different subnet when you set up the network adapter. You would need to change the address if the default board IP address is in use by another device.

If so, change the board IP address according to these guidelines:

- The subnet address, typically the first 3 bytes of the board IP address, must be the same as those of the host IP address.
- The last byte of the board IP address must be different from the last byte of the host IP address.
- The board IP address must not conflict with the IP addresses of other computers. For example, if the host IP address is `192.168.8.2`, then you can use `192.168.8.3`, if available.

**Default:** `192.168.1.102`

**Subnet mask**

Specify the subnet mask for the board. The subnet mask is a mask that designates a logical subdivision of a network.

The value of the subnet mask must be the same for all devices on the network.

**Default:** `255.255.255.0`

- `1` — One stop bit is transmitted to indicate the end of a byte.
- `2` — Two stop bits are transmitted to indicate the end of a byte.

**Gateway**

Set the serial gateway to the gateway required to access the target computer.

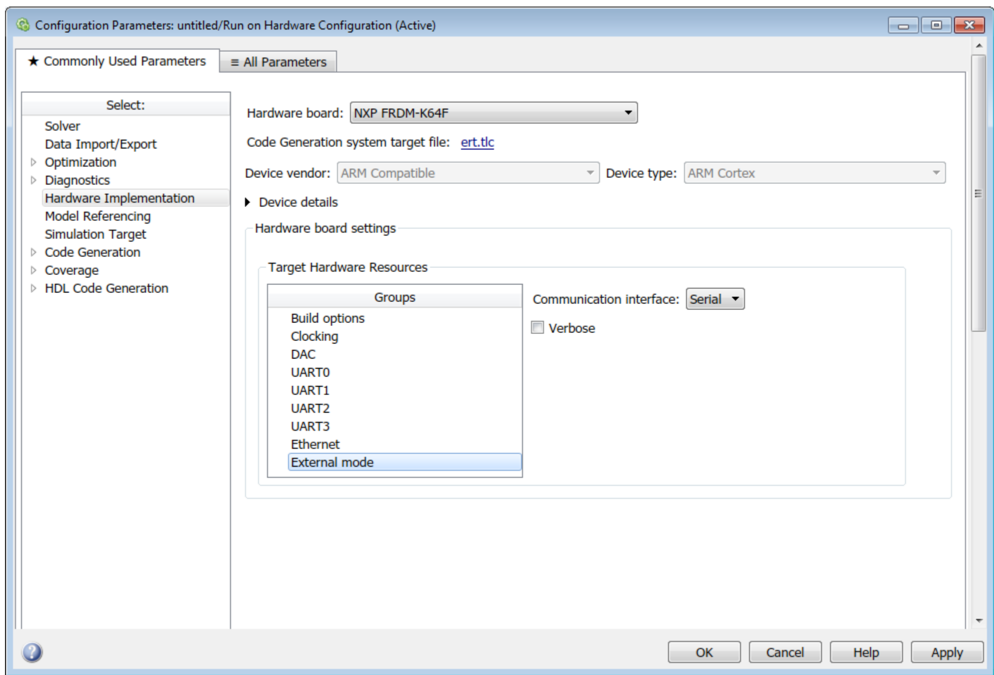
For example, when you set this parameter to `255.255.255.255`, it means that you do not use a gateway to connect to your target computer. If you connect your computers with a crossover cable, leave this property as `255.255.255.255`.

If you communicate with the target computer from within your LAN, you do not need to change this setting.

If you communicate from a host located in a LAN different from your target computer (especially via the Internet), you must define a gateway and specify its IP address in this parameter.

**Default:** `192.168.1.1`

## External mode



**Communication interface**

Use the serial option to run your model in the external mode with serial communication.

**Default:** `Serial`

**Baudrate**

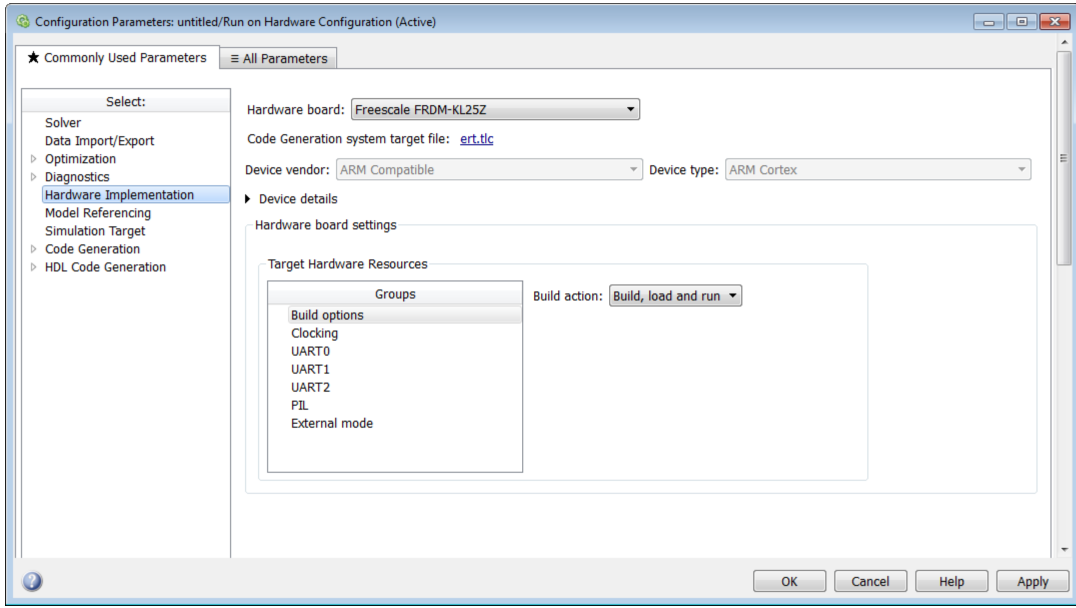
Specify the baud for UARTx serial interfaces.

**Default:** `115200`

**Verbose**

Select this check box to view the external mode execution progress and updates in the Diagnostic Viewer or in the MATLAB Command Window.

# Hardware Implementation Pane: Freescale FRDM-KL25Z




| In this section... |
| --- |
| "Code Generation Pane" on page 13-121 |
| "Scheduler options" on page 13-121 |
| "Build Options" on page 13-121 |
| "Clocking" on page 13-122 |
| "I2C0" on page 13-122 |
| "I2C1" on page 13-123 |
| "Timer/PWM" on page 13-123 |
| "UART0, UART1, and UART2" on page 13-123 |
| "PIL" on page 13-124 |
| "External mode" on page 13-125 |

## Code Generation Pane

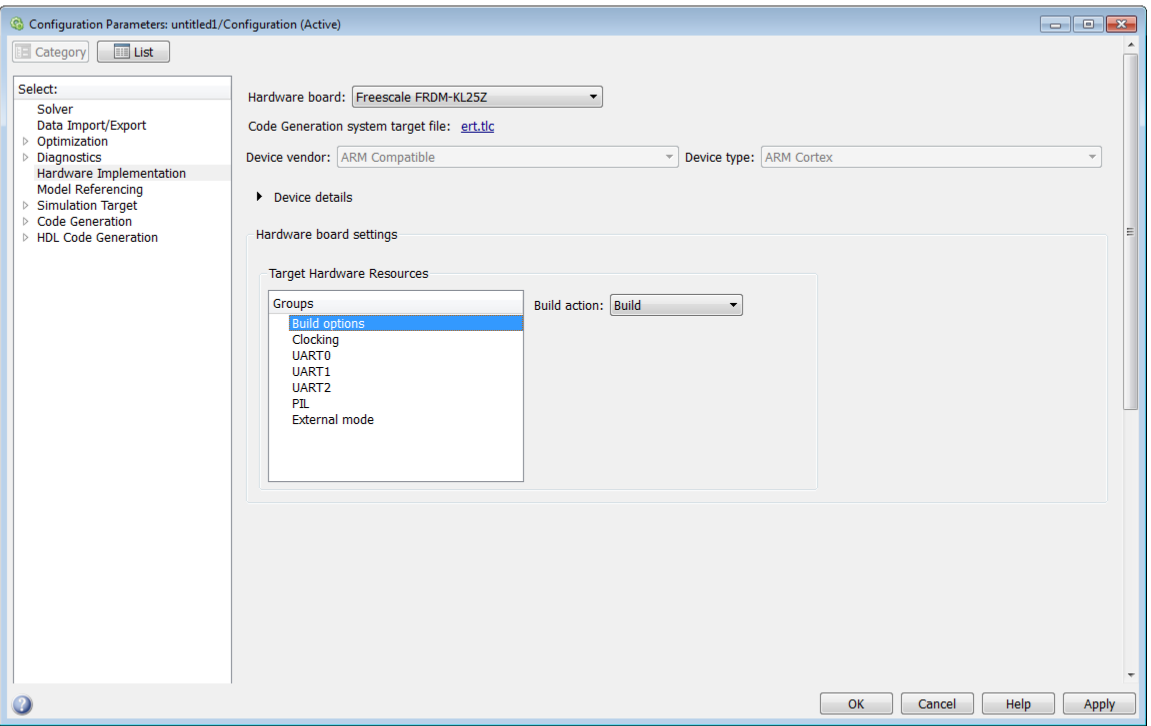
Select the System target file and the Toolchain for your hardware.

## Scheduler options

### Scheduler interrupt source

Select the source of the scheduler interrupt.

## Build Options

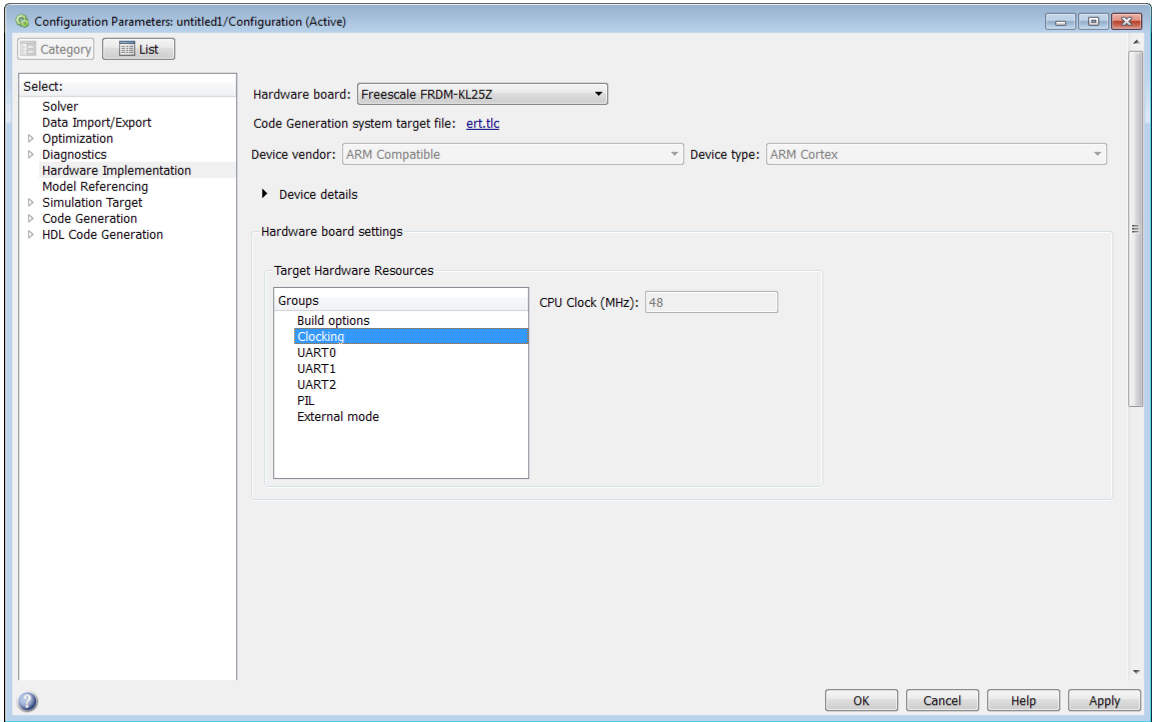



Select build options to specify how the build process should take place during code generation.

### Build action

Select an option to specify that you want only build or build, load, and run actions during the build process.

## Clocking




### CPU Clock (MHz)

This is the CPU clock frequency of the Freescale FRDM-KL25Z processor on the target hardware.

## I2C0

### SCL Pin

### SDA Pin

## I2C1

### SCL Pin

### SDA Pin

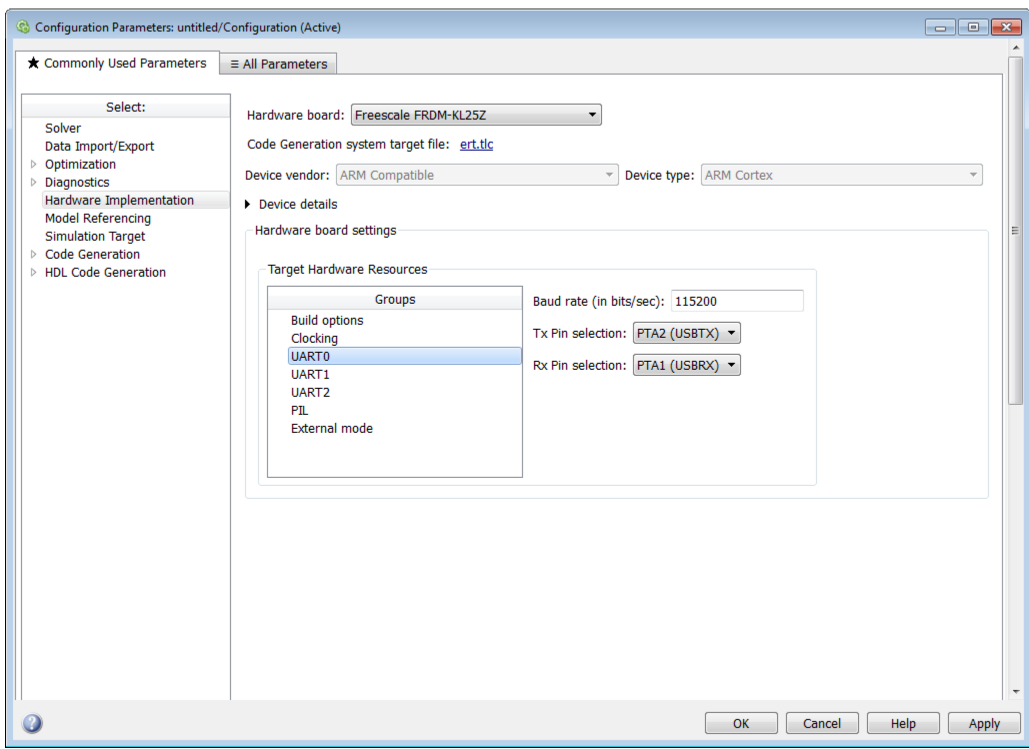
## Timer/PWM

### TPM0 Frequency (in Hz)

### TPM0 Frequency (in Hz)

### TPM2 Frequency (in Hz)

## UART0, UART1, and UART2



**Baud rate (in bits/s)**

Specify the baud rate for UARTx serial interfaces.

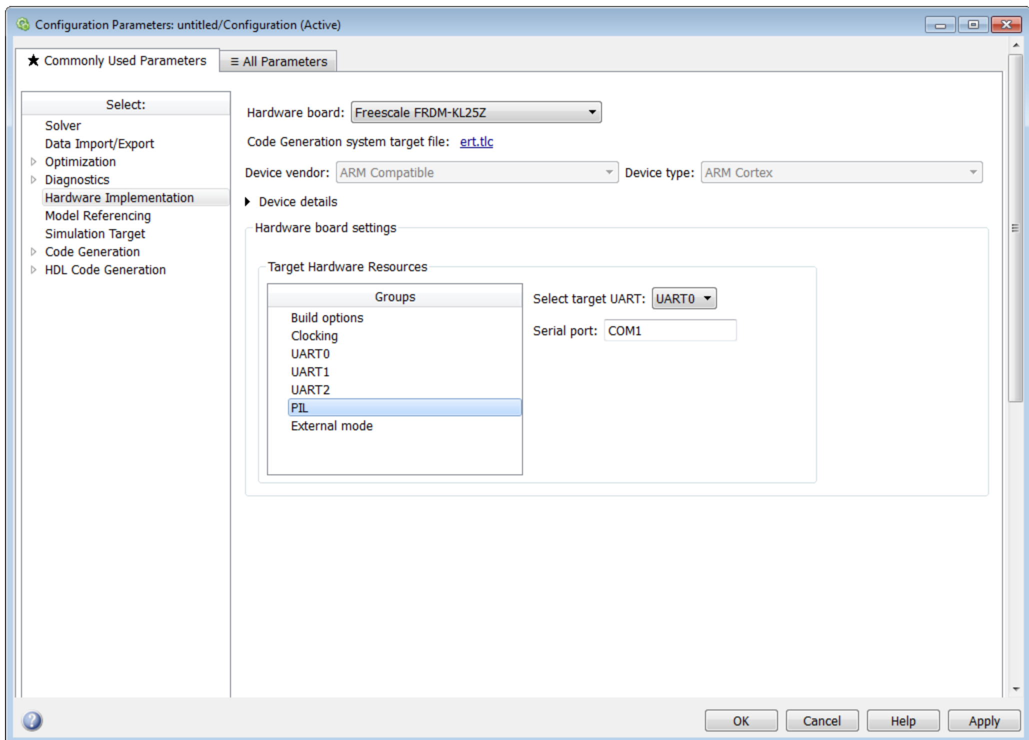
**Tx Pin**

Select a Tx pin for serial communication.

**Rx Pin**

Select an Rx pin for serial communication.

## PIL



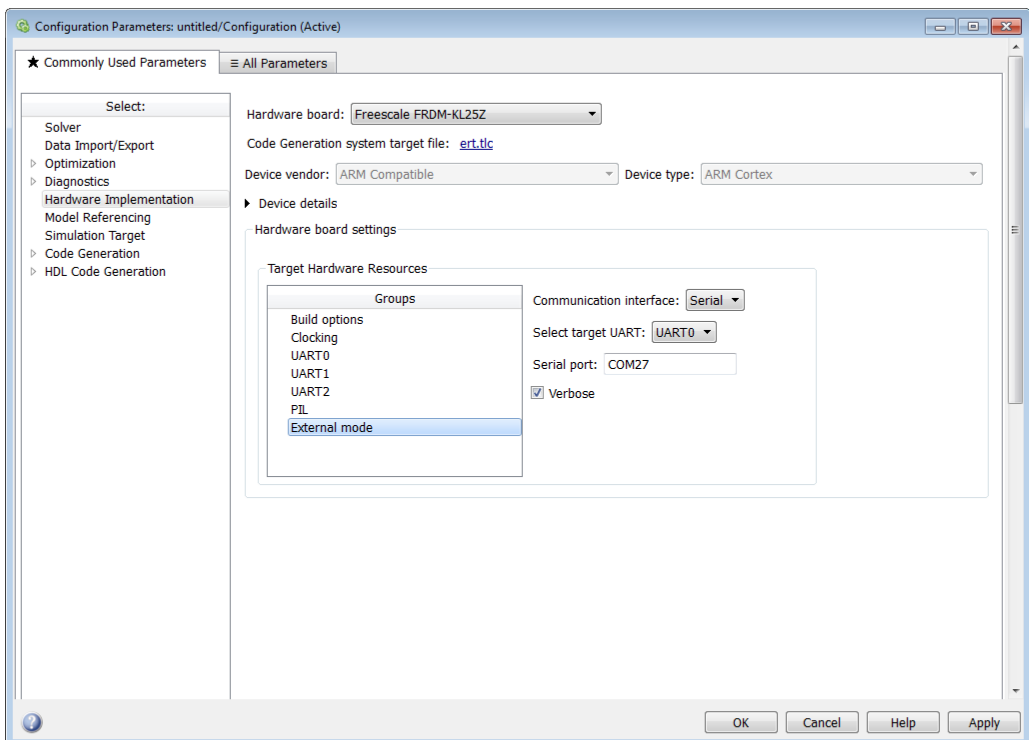

**Select target UART**

Select the target UART port for PIL communication. After selecting an UART port, you must go to the selected UART in **Configuration Parameters > Hardware Implementation pane > UARTx** and select the Tx and the Rx pins.

**Serial port**

Enter the serial port used for PIL communication.

## External mode




**Communication interface**

Use the 'serial' option to run your model in external mode with serial communication.

**Select target UART**

Select the target UART port for external mode communication.

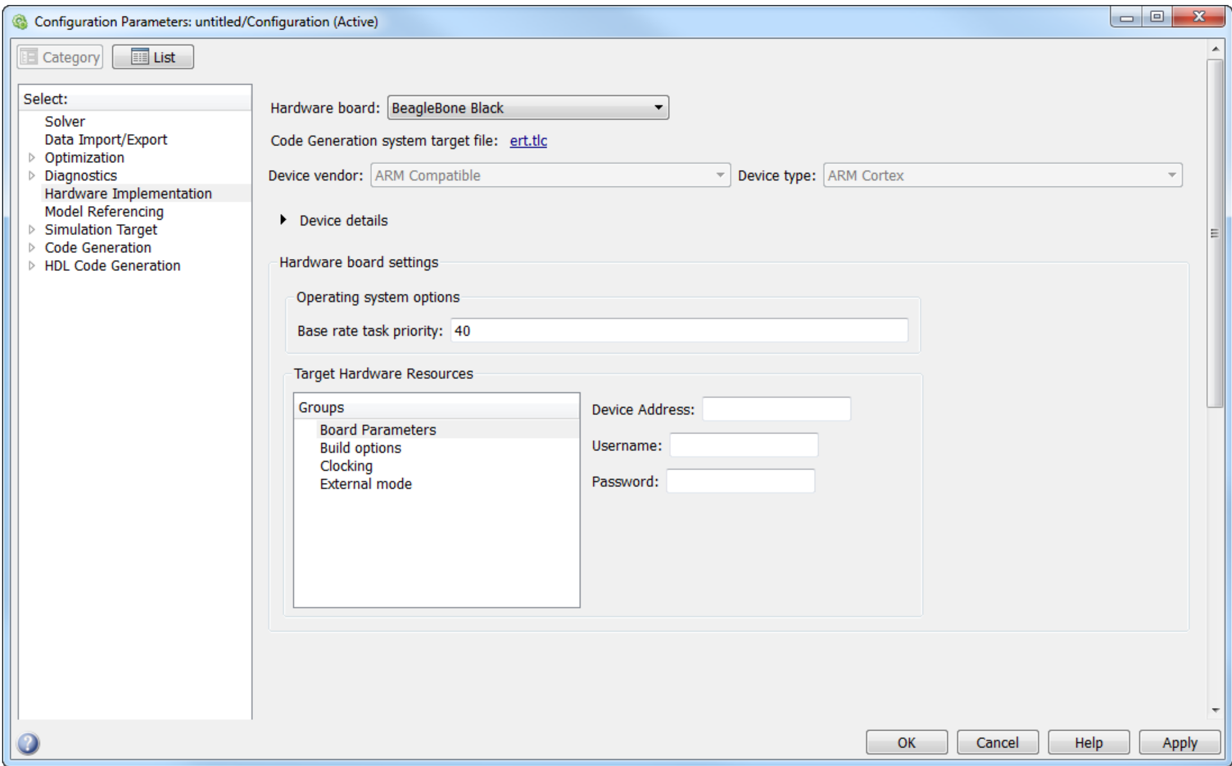
**Serial port**

Enter the serial port used for external mode communication.

**Verbose**

Select this check box to view external mode execution progress and updates in the Diagnostic Viewer or in the MATLAB command window.

# Hardware Implementation Pane: BeagleBone Black




| In this section... |
| --- |
| “Hardware Implementation Pane Overview” on page 13-128 |
| “Board Parameters” on page 13-128 |
| “Build Options” on page 13-128 |
| “Clocking” on page 13-129 |
| “Operating system options” on page 13-129 |
| “External mode” on page 13-130 |

## Hardware Implementation Pane Overview

Configure the parameters for properties of the physical hardware, such as peripherals.

## Board Parameters

**Device Address**

Specify the device address.

**Username**

Specify the user name of the root user. By default, this value is `root`.

**Password**

Specify the password of the root user.

## Build Options

**Build action**

Defines how the code generator responds when you press Ctrl+B to build your model.

### Settings

**Default:** `Build, load, and run`

`Build, load, and run`

With this option, pressing Ctrl+B or clicking Build Model:

1. Generates code from the model.
2. Compiles and links the code into an executable with libraries.
3. Loads the executable and libraries into the QEMU emulator.
4. Runs the executable in the QEMU emulator.

`Build`

With this option, pressing Ctrl+B or clicking Build Model:

1. Generates code from the model.
2. Compiles and links the code into an executable with libraries.

This option does not load and run the executable in the QEMU emulator.

#### Command-Line Information

**Parameter:** `buildAction`
**Type:** character vector
**Value:** `Build` | `Build_load_and_run` |
**Default:** `Build_load_and_run`

#### Recommended Settings

| Application | Setting |
| --- | --- |
| Debugging | `Build_load_and_run` |
| Traceability | No impact |
| Efficiency | No impact |
| Safety precaution | No impact |

#### See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

For more information about PIL and its uses, refer to the “Verifying Generated Code via Processor-in-the-Loop” topic.

## Clocking

**CPU Clock (MHz)**

Enter the actual CPU clock frequency in MHz. This value does not set the processor frequency.

The software uses this value to calculate the speed of the processor.

## Operating system options

**Base Rate Task Priority**

This parameter sets the static priority of the base rate task. By default, the priority is `40`.

## External mode

**Communication interface**

Select the transport layer external mode uses to exchange data between the host computer and the target hardware.

**Run external mode in a background thread**

Force the external mode engine in the generated code to execute in a background task.

This option is not recommended for models that use a very small time step, or which may encounter task overruns. These situations can cause Simulink to become unresponsive.

**Port**

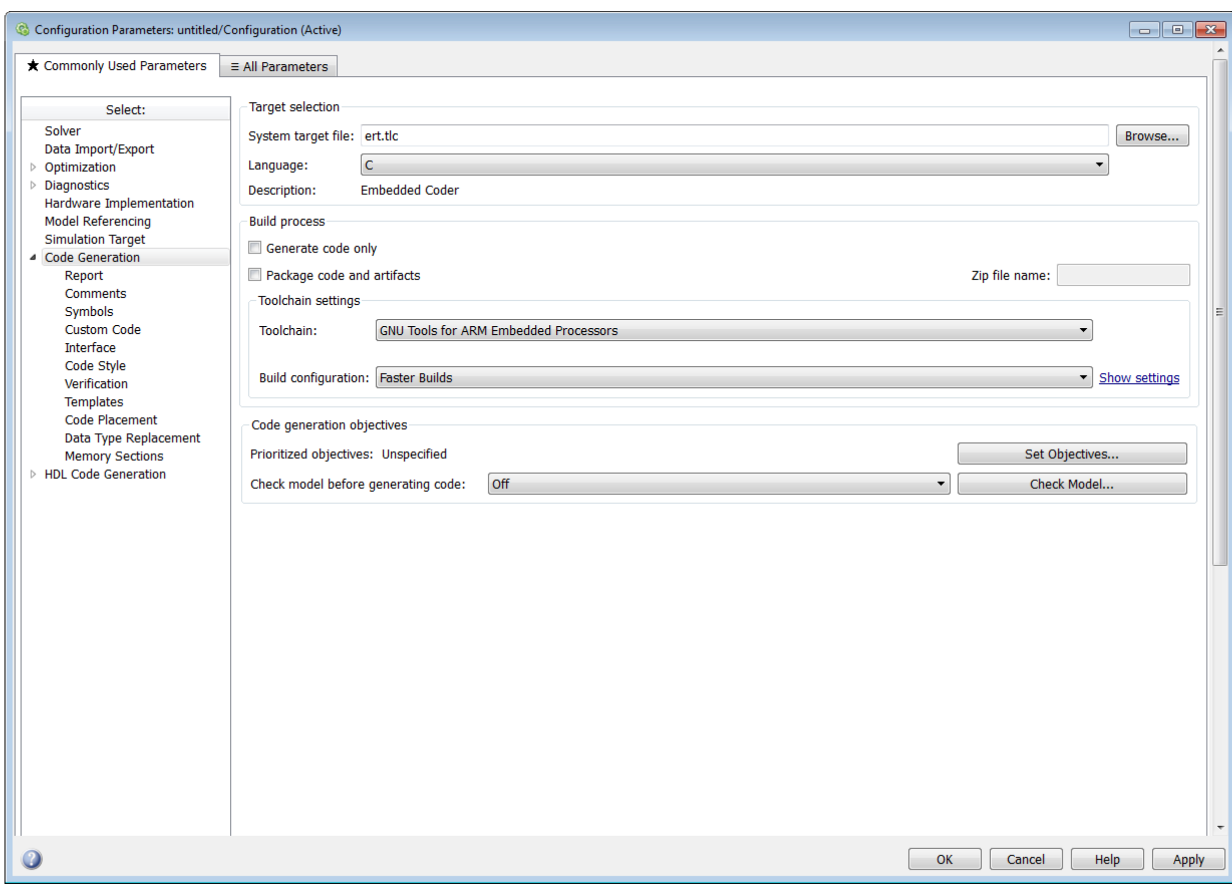
Set the value of the TCP/IP or WiFi port number, from 1024 to 65535. External mode uses this port for communications between the hardware board and host computer.

**Default:** 17725

**Verbose**

Display verbose messages in the MATLAB Command Window.

# Hardware Implementation Pane: STMicroelectronics Discovery Boards




**In this section...**

"Hardware Implementation Pane Overview" on page 13-132

"Embedded Coder Support Package for STMicroelectronics Discovery Boards Hardware Settings" on page 13-132

"Operating system options" on page 13-134

"Scheduler options" on page 13-135

| In this section... |
|---|
| "Build options" on page 13-137 |
| "Clocking" on page 13-139 |
| "PIL" on page 13-140 |
| "ADC Common" on page 13-142 |
| "ADC 1, ADC 2, ADC 3" on page 13-144 |
| "GPIO A, GPIO B, GPIO C, GPIO D, GPIO E, GPIO F, GPIO G, GPIO H, GPIO I" on page 13-146 |
| "External mode" on page 13-147 |

## Hardware Implementation Pane Overview

Configure the parameters for properties of the physical hardware, such as peripherals.

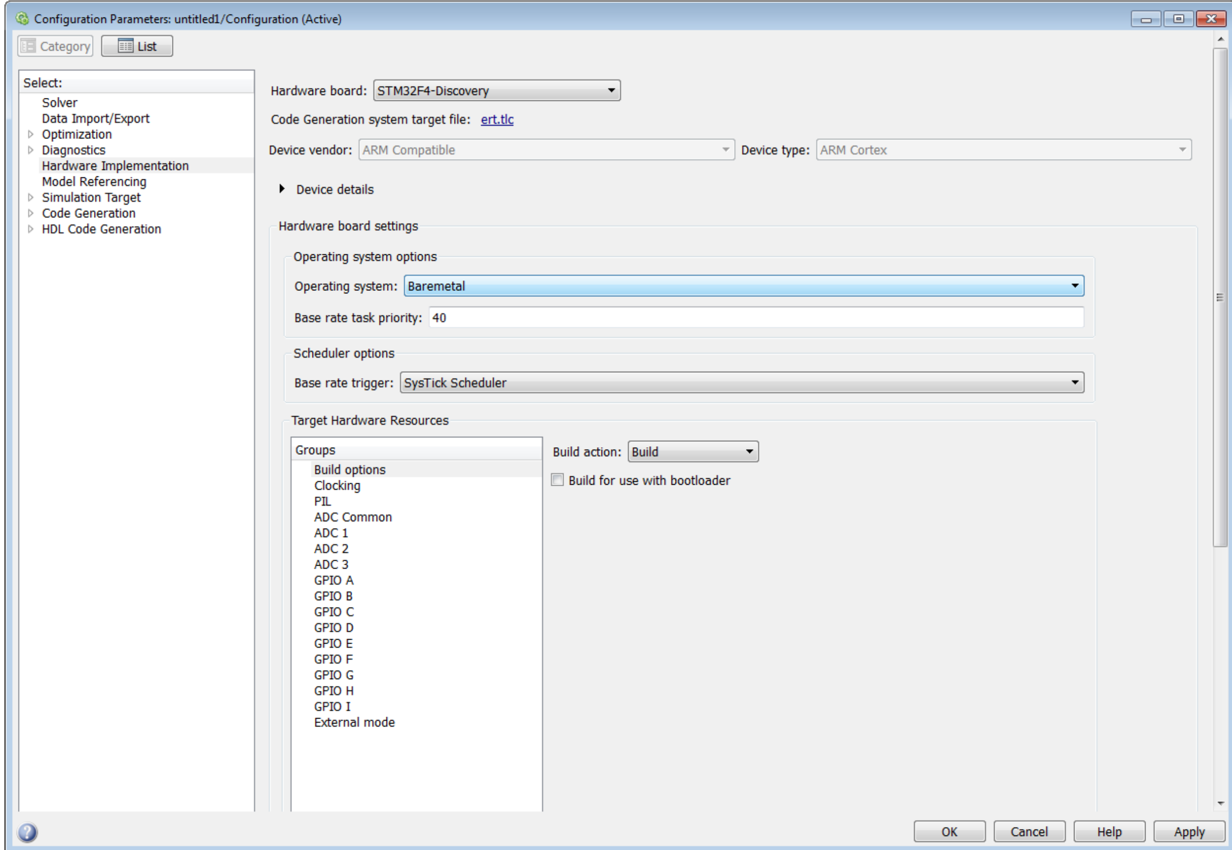
## Embedded Coder Support Package for STMicroelectronics Discovery Boards Hardware Settings

The following table provides links to topics for Support Package for STMicroelectronics® STM32F4 Discovery hardware processors.

| Peripheral Name | Description |
|---|---|
| "Operating system options" on page 13-134 | The operating system options for running the model on the target hardware |
| "Scheduler options" on page 13-135 | The scheduler parameter to select the base rate trigger |
| "Build options" on page 13-137 | The build options to specify how the build process should take place during code generation |
| "Clocking" on page 13-139 | The clocking parameters to view the CPU clock rate |
| "PIL" on page 13-140 | The Processor—in—the—Loop (PIL) parameters to configure PIL communication parameters, such as interface and COM port |

| Peripheral Name | Description |
|---|---|
| “ADC Common” on page 13-142 | The Analog to Digital Converter (ADC) parameters to set the common ADC peripheral parameters |
| “ADC 1, ADC 2, ADC 3” on page 13-144 | The parameters to configure different channels on the three ADCs, ADC1, ADC2, and ADC3 |
| “GPIO A, GPIO B, GPIO C, GPIO D, GPIO E, GPIO F, GPIO G, GPIO H, GPIO I” on page 13-146 | The GPIO parameters to configure different GPIO pins |
| “External mode” on page 13-147 | The external mode options to log and tune the parameters while the model runs on the target hardware in real-time. |

## Operating system options

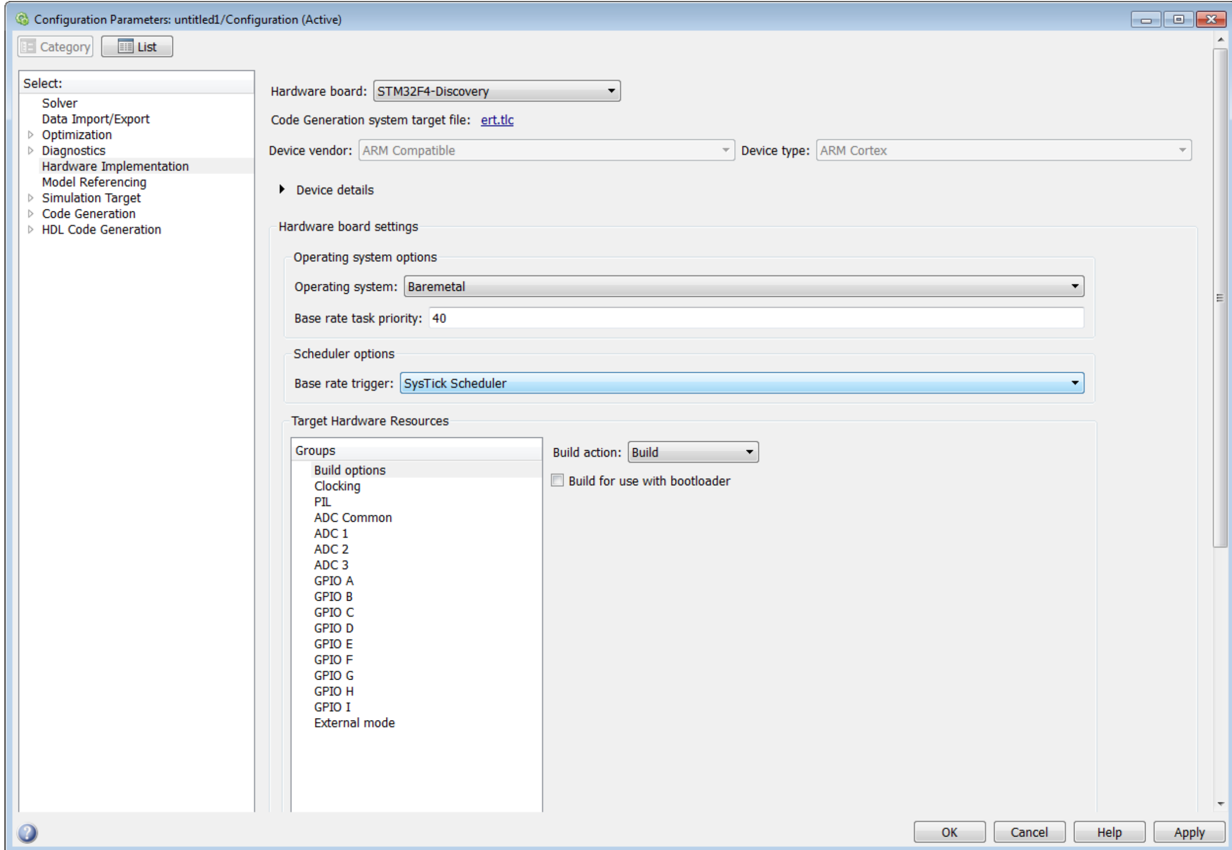



Select the operating system options to run your model on the selected target hardware.

**Operating system** — Select the operating system to run your model. The options are:

- `Baremetal` — Select this option to run a rate monotonic scheduler for your model using the interrupt selected in the **Base rate trigger**.
- `CMSIS-RTOS RTX` — Select this option to leverage the CMSIS-RTOS RTX provided by ARM to schedule the rates present in the model.

## Scheduler options




The source of the scheduler interrupt.

The **Scheduler options** appear only when you select `Baremetal` in the **Operating system** drop-down list.

The different scheduler options available are as follows:

**Base rate trigger**

- `SysTick Scheduler` – The SysTick timer interrupt schedules the model at the base rate. This timer is available in the ARM Cortex-M3 and the ARM Cortex-M4

based processors. When you select STM32F4-Discovery as the target hardware, this option is the default Scheduler option.

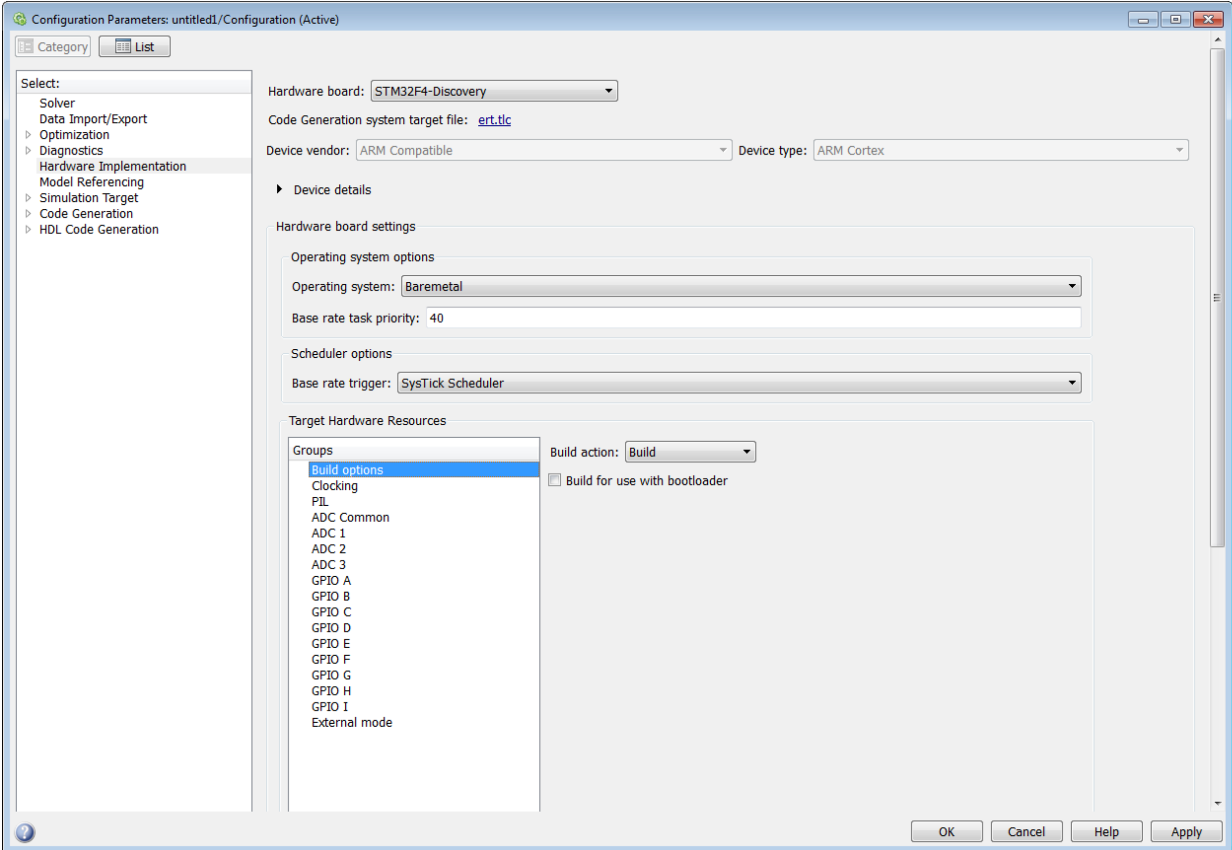
- `“Mic in” block DMA interrupt` – The model is scheduled based on the DMA interrupt rate of the Mic in block. The Mic in block generates a DMA (DMA1-Channel0-Stream3) interrupt for every Mic in block sample time. Make sure that the model does not contain blocks with sample rates higher than the Mic in block.
- `“Audio out” block DMA interrupt` – The model is scheduled based on the DMA interrupt rate of the Audio out block. The DMA interrupt rate depends on the frame size that is input to the Audio out block in the model. Make sure that the model does not contain blocks with sample rates higher than the Audio out block.

---

**Note** Make sure that your model has Mic in block if you select `“Mic in” block DMA interrupt` option and the Audio out block if you select `“Audio out” block DMA interrupt` option as the **Base rate trigger**.

---

## Build options




Use the build option to specify how the build process should take place during code generation.

### Build action

is the option to specify if you want only build or build, load and run actions during code generation.

- `Build` — Select this option if you want to build the code during the build process.
- `Build, load and run` — Select this option to build, load, and to run the generated code during the build process.

**Build for use with bootloader**

Select this check box to use the internal bootloader during the code programming process.

By selecting this option, you generate an output executable to use with bootloader for programming the target hardware memory through a standard communication channel, without having to use an IDE/third party tool.

This check box is available only when you select `Baremetal` operating system.

After the first use of bootloader with Simulink, you can also interact with the target hardware from the command line interface.

1. Create a target hardware object that connects through serial port ‘COM1’ using the command:

   ```
   obj= bootloader('COM1')
   ```
2. Load the executable generated from the Simulink model ‘model1.slx’, with the bootloader option selected, using the command:

   ```
   [err, msg ] = load(obj,'model1.hex')
   ```
3. Clear the object using the command:

   ```
   clear obj
   ```

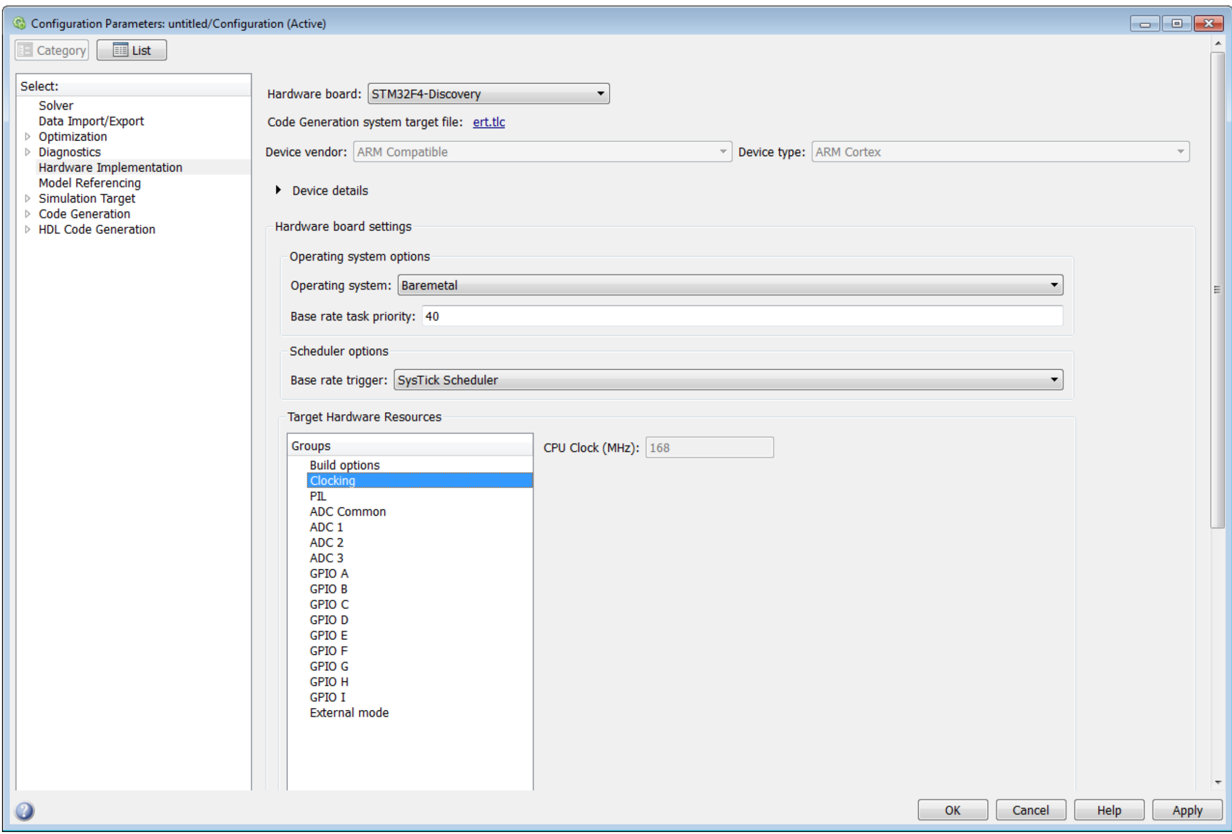
**Bootloader interface**

This is the communication interface used by the bootloader.

**Serial port**

This is the serial port used by the bootloader.

## Clocking

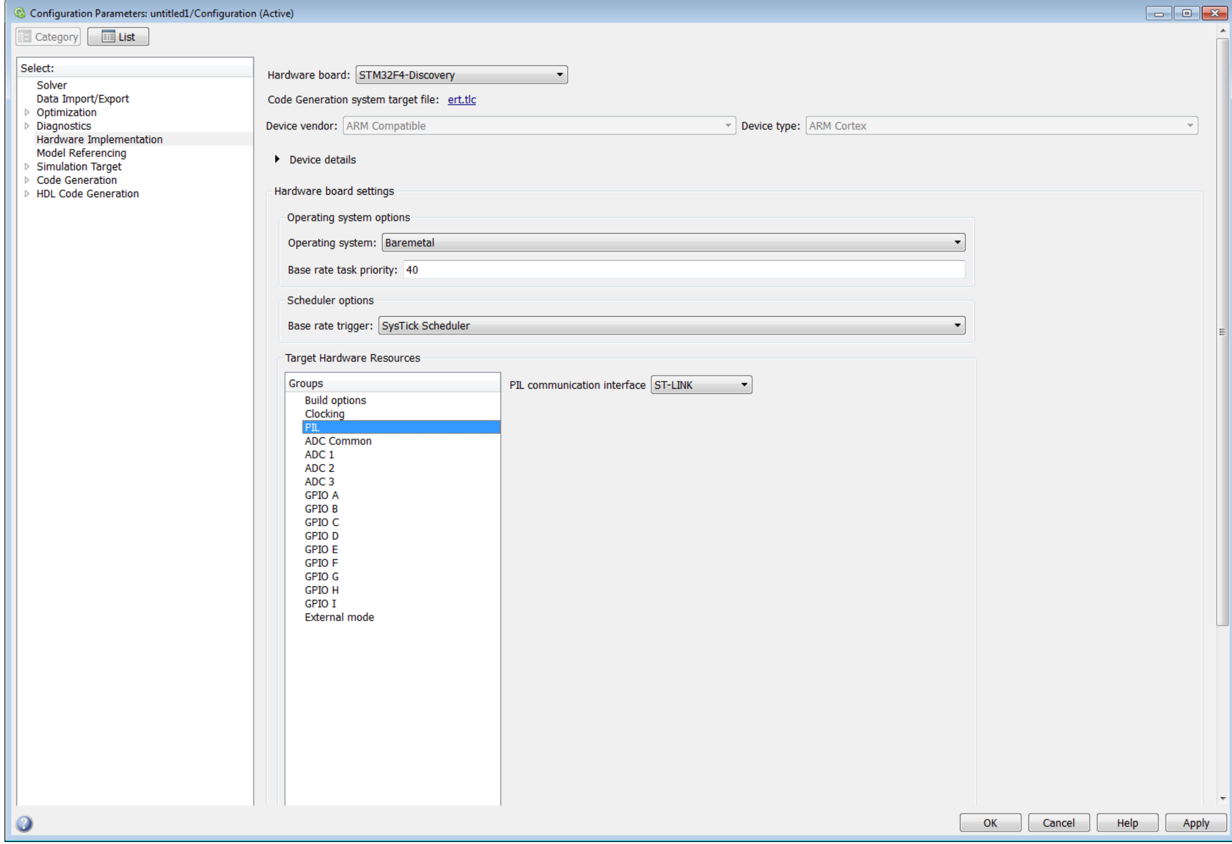



Use the clocking option to achieve the CPU Clock rate specified.

**CPU Clock (MHz)**

The CPU clock rate.

# PIL




Use the PIL options to set PIL (Processor-in-the-Loop) communications parameters.

### PIL communication interface

The interface used for PIL communication.

**Default:** `Serial (USART2)`

`ST-LINK`

### Serial port

Specify the serial port for PIL communication.

To find the serial port:

1 Go to **Start menu > Control Panel > Device Manager > Ports (COM & LPT)**.

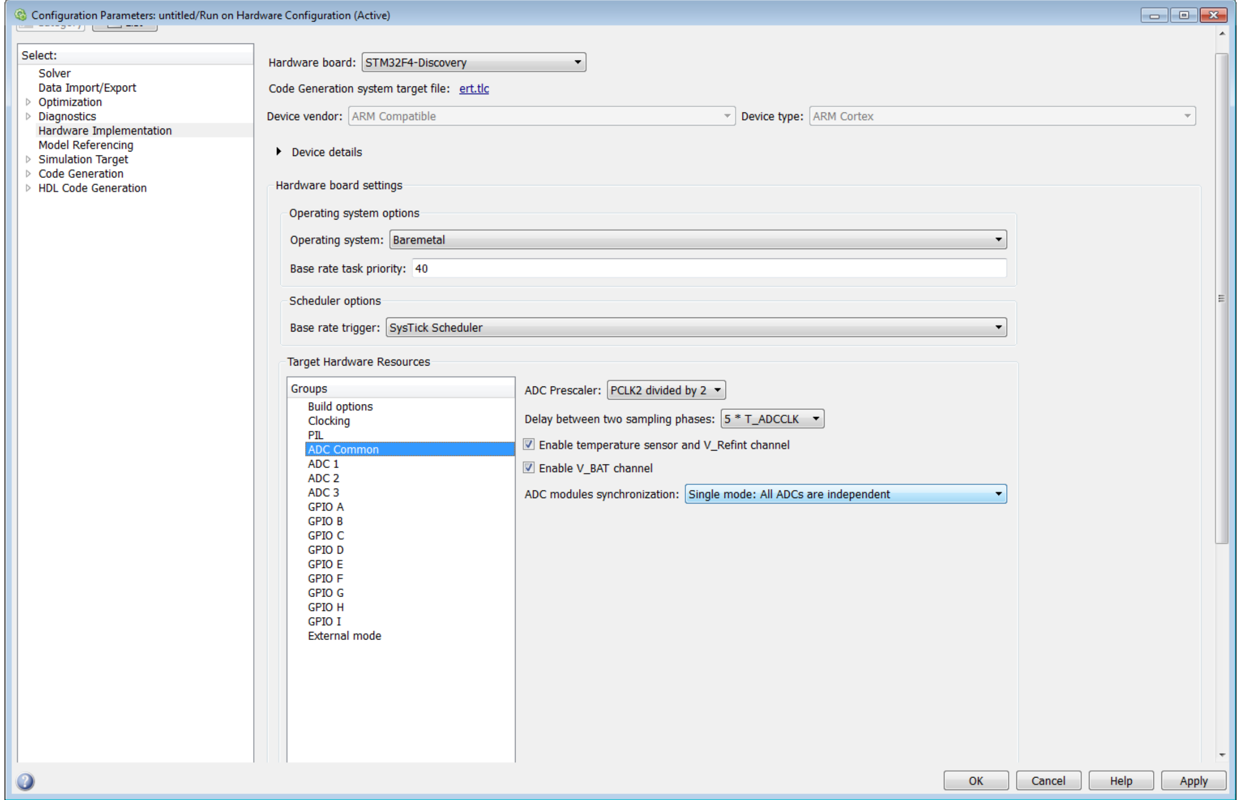
2 To expand the ports, that Windows recognizes, click the plus (+) sign. The port that you want is STMicroelectronics STLink Virtual COM Port (COMx). x is the number of the port.

   For example, if the serial port is labeled as STMicroelectronics STLink Virtual COM Port (COM1), in the **Serial port** parameter, specify the serial port as COM1.

On the hardware board, this port is named ST_LINK.

**Default:** COM1

## ADC Common




Use the ADC Common options to set the common ADC parameters.

### ADC Prescaler

The option to select the frequency of the clock, PCLK2, to ADC. The maximum frequency of the clock is set to 84 MHz.

The default value of `PCLK2 divided by 2` sets the frequency to 48 MHz.

### Delay between two sampling phases

The option to select the minimum delay that separates 2 ADC conversions in interleaved mode.

**Enable temperature sensor and V_Refint channel**

The option to enable the temperature sensor and the V_Refint channel.

**Enable V_BAT channel**

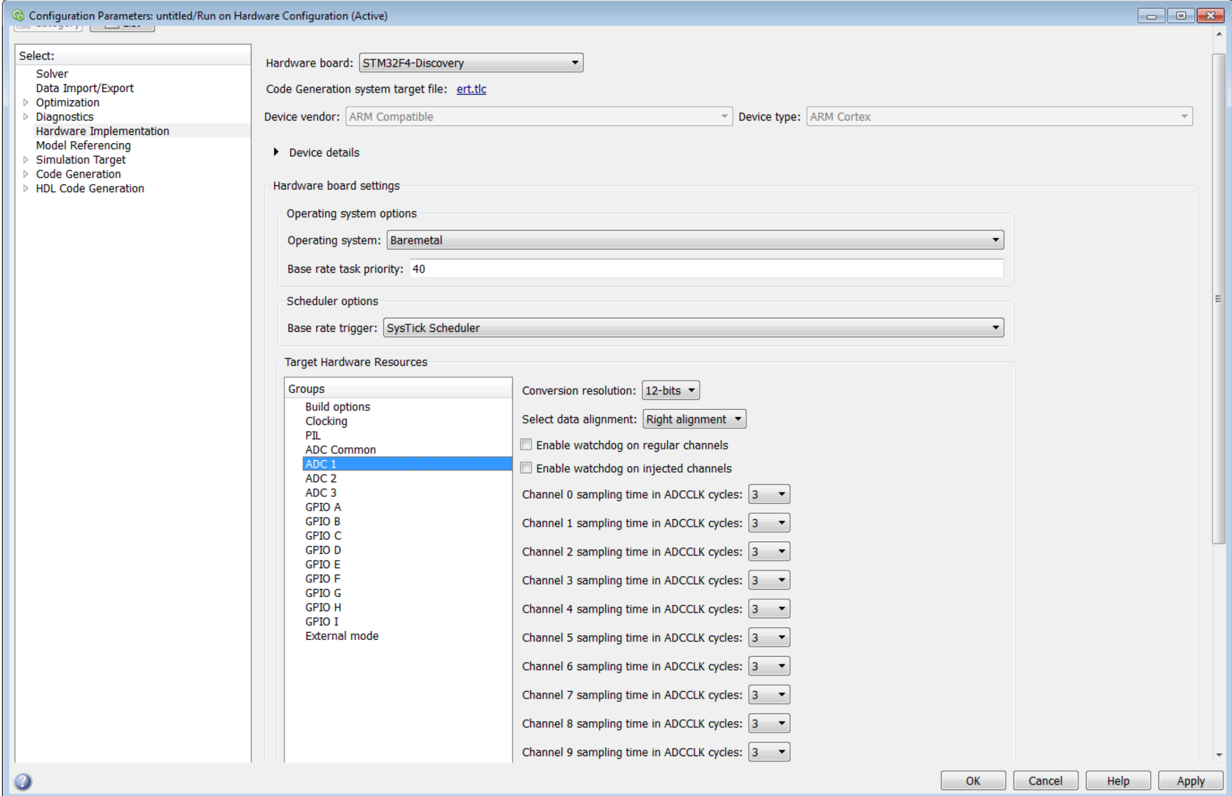
The option to enable the V_BAT channel.

**ADC modules synchronization**

The option that you select for ADC module synchronization. The different options you available are:

- `Single mode: All ADCs are independent` — Select this option when the three ADCs must perform the conversion independently.
- `Dual mode: ADC1 and ADC2 combined, ADC3 independent` — Select this option when you want to combine ADC1 and ADC2 and ADC3 to perform conversion independently.
- `Triple mode: ADC1, ADC2, and ADC3 combined` — Select this option when you want to combine ADC1, ADC2, and ADC3 together for conversion.

# ADC 1, ADC 2, ADC 3




Use the three ADC1, ADC2, and ADC3 parameters to configure the sample time in ADDCLK cycles for different channels.

### Conversion resolution

The resolution that you select for conversion.

### Convert number of channels in discontinuous mode

The number of channels to convert in discontinuous mode.

### Select data alignment

The option that you select for the alignment of data after conversion. The data can be right or left aligned.

**Enable watchdog on regular channels**

The option you select to enable the watchdog on regular channels.

**Enable watchdog on injected channels**

The option you select to enable the watchdog on injected channels.

**Enable discontinuous conversion on injected group**

The option that you select to enable the discontinuous conversion on injected group.

**Enable discontinuous conversion on regular group**

The option that you select to enable discontinuous conversion on regular group.

**DMA mode for multi ADC mode**

The option that you select for DMA mode in multi ADC mode.

**Enable DMA selection for multi ADC mode**

The option that you select to enable DMA selection for multi ADC mode.

**Multi ADC mode selection**

The option that you select for multi ADC mode.

**Watchdog on channel**

The option to select channel for watchdog.

**Watchdog lower threshold**

The lower threshold of the watchdog.

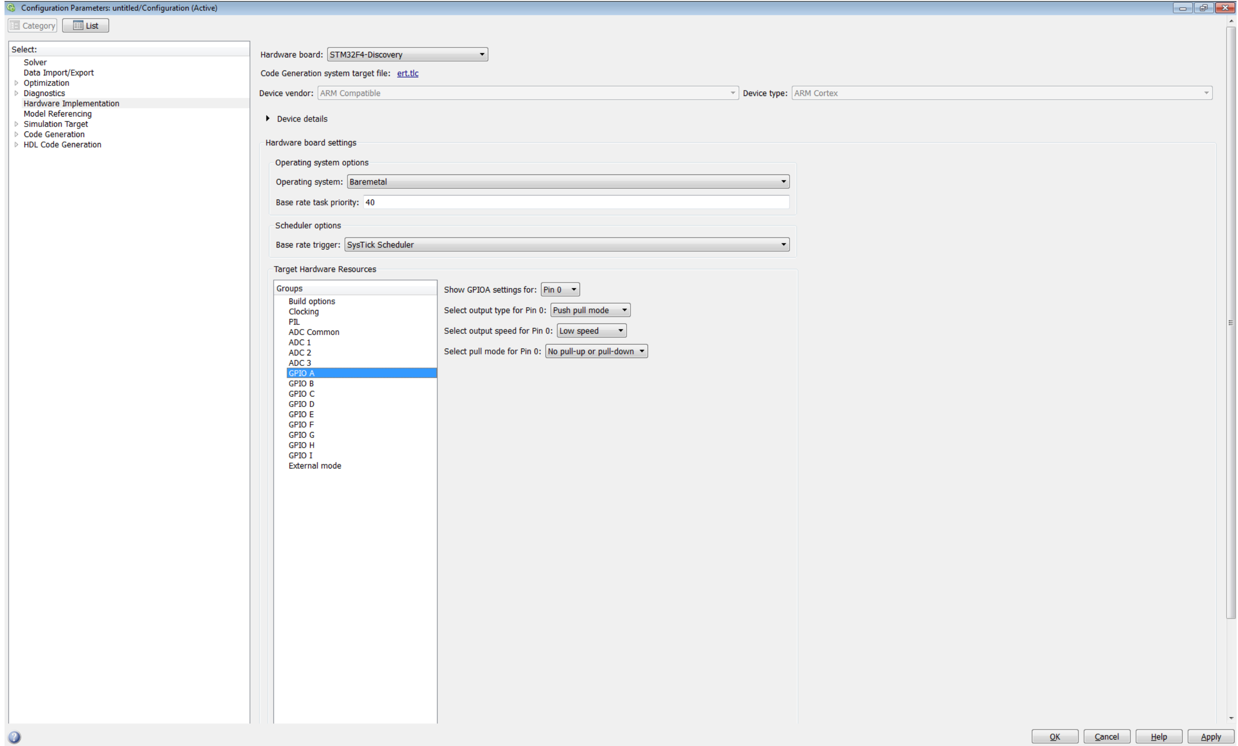
**Watchdog higher threshold**

The higher threshold of the watchdog.

**Channel # sampling time in ADCCLK cycles**

The sampling time that you select in ADCCLK cycles for channel numbers from 1 to 18.

# GPIO A, GPIO B, GPIO C, GPIO D, GPIO E, GPIO F, GPIO G, GPIO H, GPIO I




Use the GPIO A-I parameters to configure the pins for input/output.

**Show GPIO# settings for**

The pin number that you select to show the GPIO settings.

**Select output type for Pin #**

The output type that you select for pins 0 to 15.

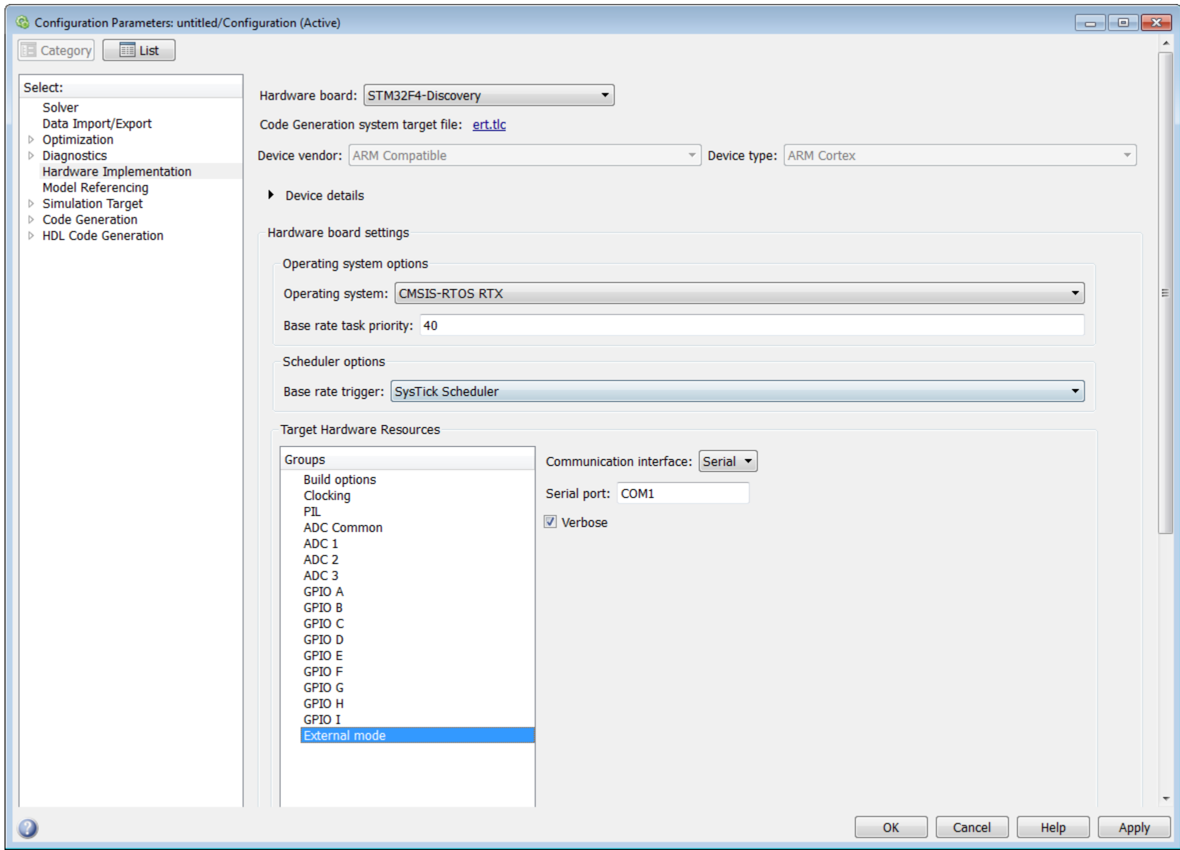
**Select output speed for Pin #**

The output speed that you select for pins 0 to 15.

**Select pull mode for Pin #**

The pull mode that you select for pins from 0 to 15.

## External mode




**Communication interface**

Use the ‘serial’ option to run your model in external mode with serial communication.

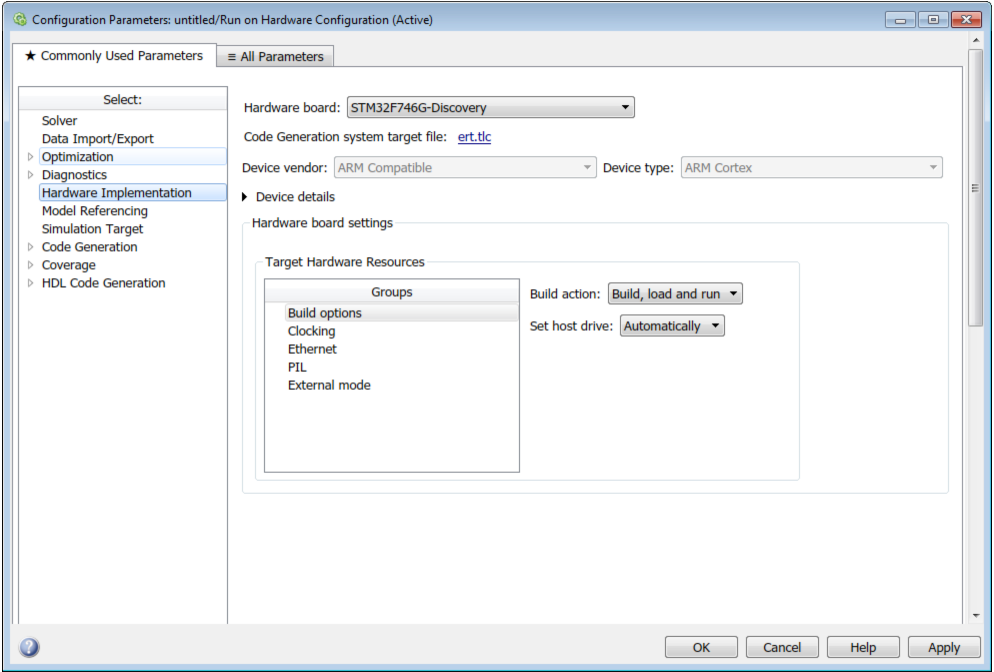
**Serial port**

Enter the serial port used by the target hardware.

**Verbose**

Select this check box to view external mode execution progress and updates in the Diagnostic Viewer or in the MATLAB command window.

# Hardware Implementation Pane



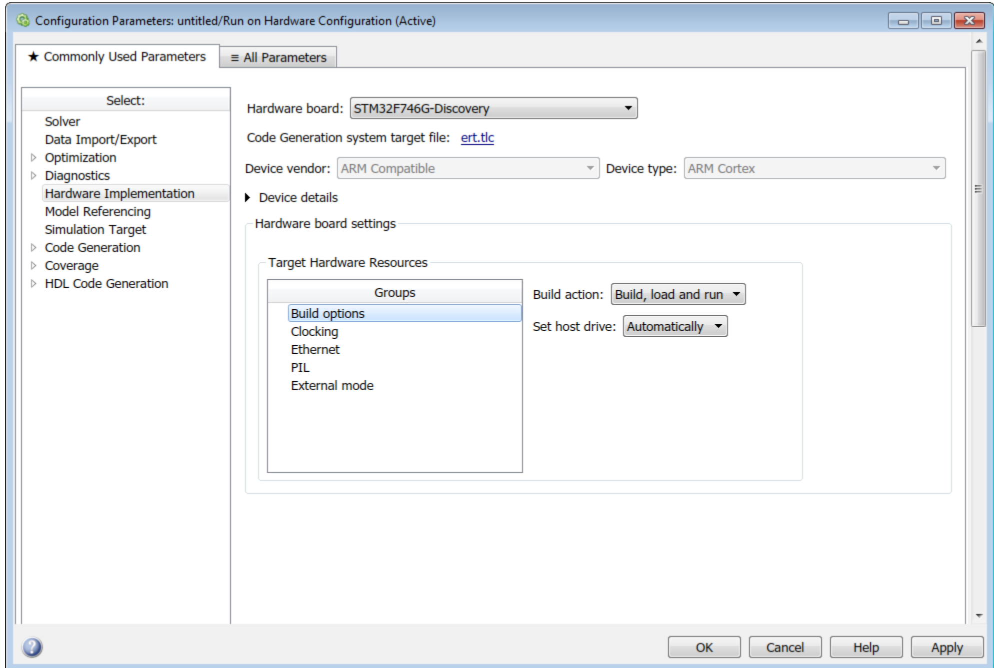

**In this section...**

"Hardware Implementation Pane Overview" on page 13-148
"Build options" on page 13-149
"Clocking" on page 13-151
"Ethernet" on page 13-152
"PIL" on page 13-154
"External mode" on page 13-156

## Hardware Implementation Pane Overview

Configure the parameters for properties of the physical hardware, such as peripherals.

## Build options




To specify how the build process takes place during code generation, select build options.

### Build action

Specify whether you want only a build action or build, load, and run actions during code generation.

**Default:** `Build, load and run`

- `Build` — Select this option if you want to build the code during the build process.
- `Build, load and run` — Select this option to build, load, and run the generated code during the build process.

### Set host drive

Specify whether you want to automatically detect or manually set the drive letter that your host computer uses to download the executable on the hardware board.

**Default:** `Automatically`

- `Automatically` — MATLAB determines which drive letter your host computer uses to download the executable on the hardware board.

  A drive letter with drive name `DIS_F746NG` is mapped to the hardware board.

  MATLAB detects this drive, copies the executable to the drive, and runs the executable on the hardware board.
- `Manually` — Select this option to specify which drive letter your host computer uses to download the executable on the hardware board.

  The drive letter that you specify in the **Drive** parameter is mapped to the hardware board.

  In the **Drive** parameter, specify only the drive letter with the drive name `DIS_F746NG`.

  For example, suppose that in the **Drive** parameter, you specify the drive letter as `F:`. In this case, MATLAB copies the executable to the `F:` drive, and runs the executable on the hardware board.

  **Note** If you set the **Set host drive** parameter to `Manually`, the **Drive** parameter is available.

**Drive**

Specify the drive letter that your host computer uses to download the executable on the hardware board.

The drive letter that you specify in the **Drive** parameter is mapped to the hardware board.

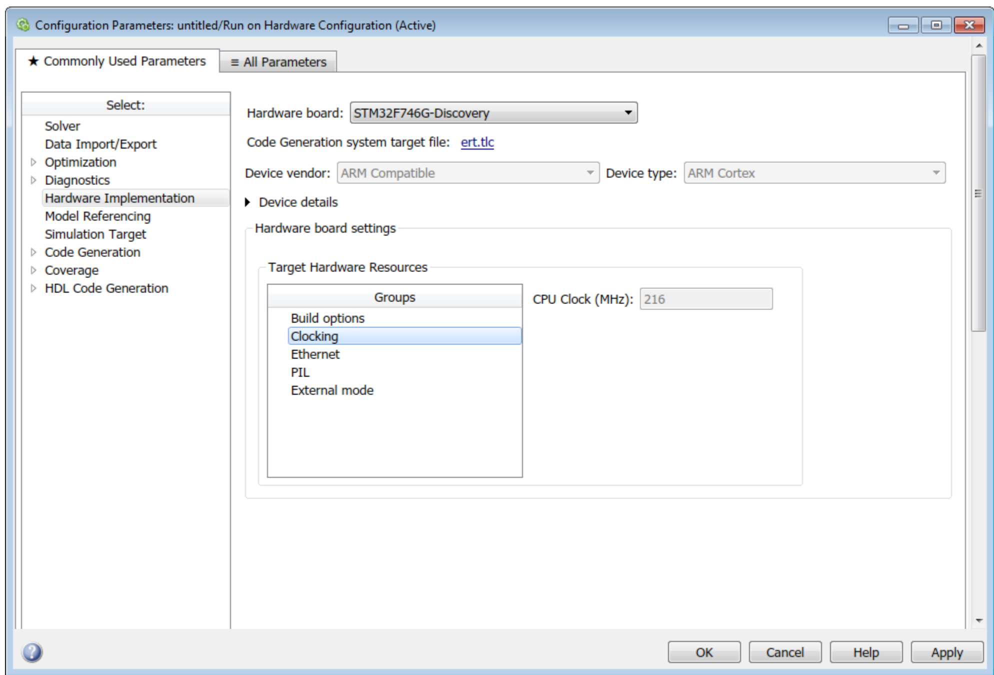
In the **Drive** parameter, specify only the drive letter with the drive name `DIS_F746NG`.

For example, suppose that in the **Drive** parameter, you specify the drive letter as `F:`. In this case, MATLAB copies the executable to the `F:` drive, and runs the executable on the hardware board.

**Note** The **Drive** parameter becomes available only when you set the **Set host drive** parameter to `Manually`.

**Default:** `F:`

## Clocking

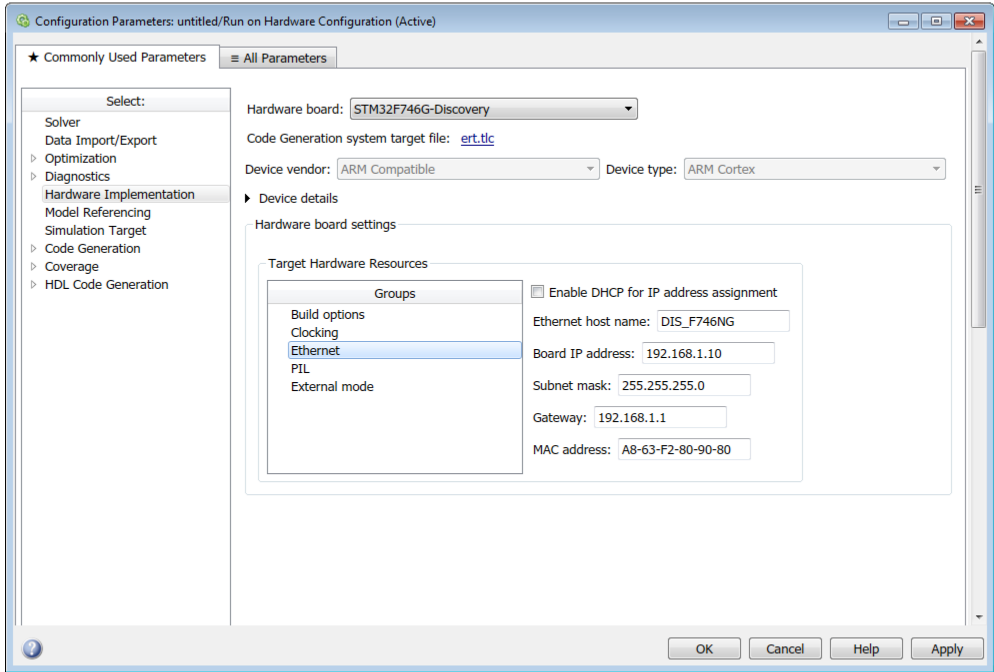



### CPU Clock (MHz)

This option is for the CPU clock frequency of the processor on the target hardware.

This parameter appears dimmed. The value of this parameter is set to `216` MHz.

## Ethernet




### Enable DHCP for local IP address assignment

Select this check box to configure the board to get an IP address from the local DHCP server on the network.

**Default:** on

on

### Ethernet host name

Specify the local host name. The local host is the board running the model.

**Default:** `DIS_F746NG`

### Board IP address

Use this option for setting the IP address of the board.

Set the board IP address according to these guidelines:

- The subnet address, typically the first 3 bytes of the board IP address, must be the same as those of the host IP address.
- The last byte of the board IP address must be different from the last byte of the host IP address.
- The board IP address must not conflict with the IP addresses of other computers. For example, if the host IP address is `192.168.8.2`, then you can use `192.168.8.3`, if available.

**Default:** `192.168.1.10`

**Subnet mask**

Specify the subnet mask for the board. The subnet mask is a mask that designates a logical subdivision of a network.

The value of the subnet mask must be the same for all devices on the network.

**Default:** `255.255.255.0`

**Gateway**

Set the serial gateway to the gateway required to access the target computer.

For example, when you set this parameter to `255.255.255.255`, it means that you do not use a gateway to connect to your target computer. If you connect your computers with a crossover cable, leave this property as `255.255.255.255`.

If you communicate with the target computer from within your LAN, you do not need to change this setting.

If you communicate from a host located in a LAN different from your target computer (especially via the Internet), you must define a gateway and specify its IP address in this parameter.

**Default:** `192.168.1.1`

**MAC address**

Specify the Media Access Control (MAC) address, the physical network address of the board.

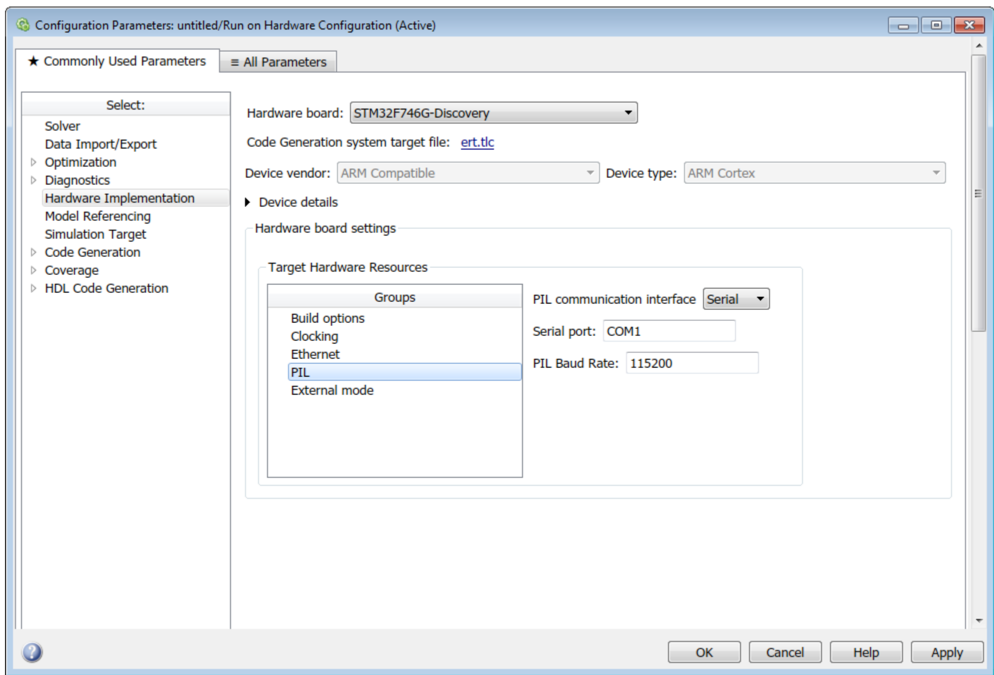
Under most circumstances, you do not need to change the MAC address. If you connect more than one board to a single computer so that each address is unique, change the MAC address. (You must have a separate network interface card (NIC) for each board.)

To change the MAC address, specify an address that is different from the address that belongs to any other device attached to your computer. To obtain the MAC address for a specific board, refer to the label affixed to the board or consult the product documentation.

The MAC address must be in the six octet format. For example, `DE-AD-BE-EF-FE-ED`

**Default:** `A8-63-F2-80-90-80`

## PIL




To set PIL (processor-in-the-loop) communications parameters, use the PIL options.

### PIL communication interface

Select the transport layer that the PIL uses to exchange data between the host and the target hardware.

**Default:** `Serial`

`TCP/IP`

**Serial port**

Specify the serial port for PIL communication.

To find the serial port:

1. Go to **Start menu > Control Panel > Device Manager > Ports (COM & LPT)**.
2. To expand the ports, that Windows recognizes, click the plus `(+)` sign. The port that you want is `STMicroelectronics STLink Virtual COM Port (COMx)`. `x` is the number of the port.

   For example, if the serial port is labeled as `STMicroelectronics STLink Virtual COM Port (COM1)`, in the **Serial port** parameter, specify the serial port as `COM1`.

On the hardware board, this port is named `ST_LINK`.

**Default:** `COM1`

**PIL Baud Rate**

Specify the baud for PIL communication.

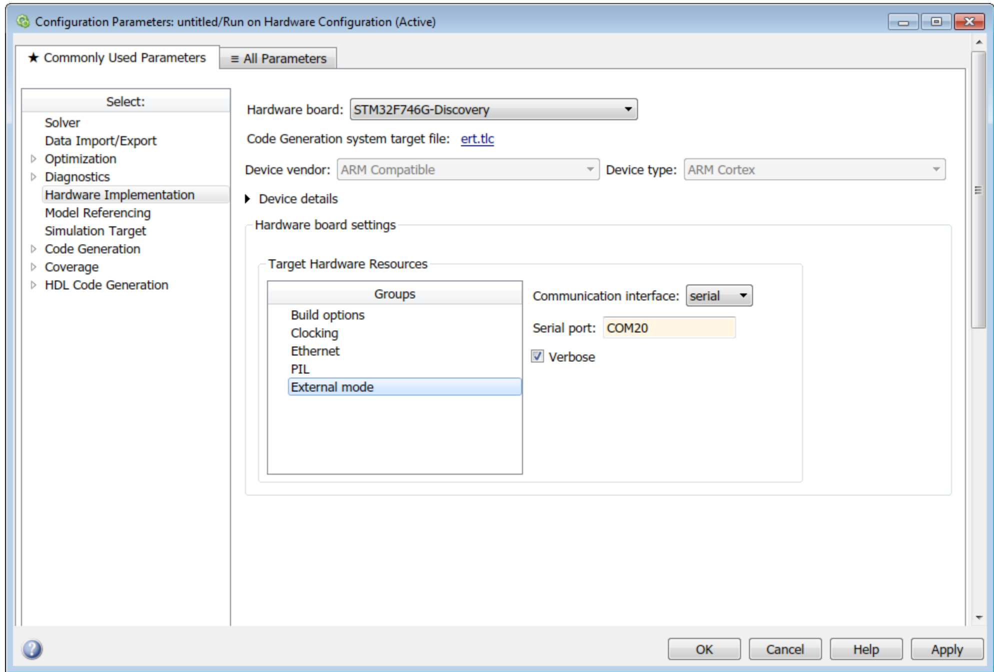
**Default:** `115200`

**Ethernet port**

The Ethernet port for TCP/IP communication.

**Default:** `17725`

## External mode




### Communication interface

Select the transport layer that the external mode uses to exchange data between the host and the target hardware.

**Default:** `Serial`

### Serial port

Enter the serial port that the target hardware uses.

To find the serial port:

1. Go to **Start menu > Control Panel > Device Manager > Ports (COM & LPT)**.
2. To expand the ports, that Windows recognizes, click the plus `(+)` sign. The port that you want is `STMicroelectronics STLink Virtual COM Port (COMx)`. `x` is the number of the port.

For example, if the serial port is labeled as `STMicroelectronics STLink Virtual COM Port (COM1)`, in the **Serial port** parameter, specify the serial port as `COM1`.

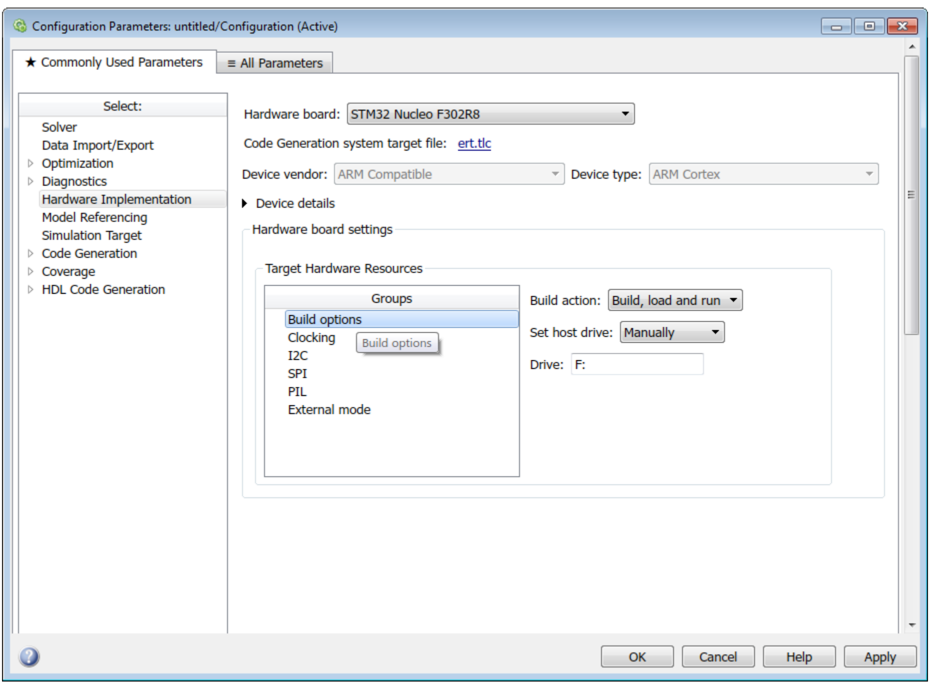
**Default:** `COM20`

**Verbose**

Select this check box to view external mode execution progress and updates in the Diagnostic Viewer or in the MATLAB Command Window.

**Default:** `on`

# Hardware Implementation Pane





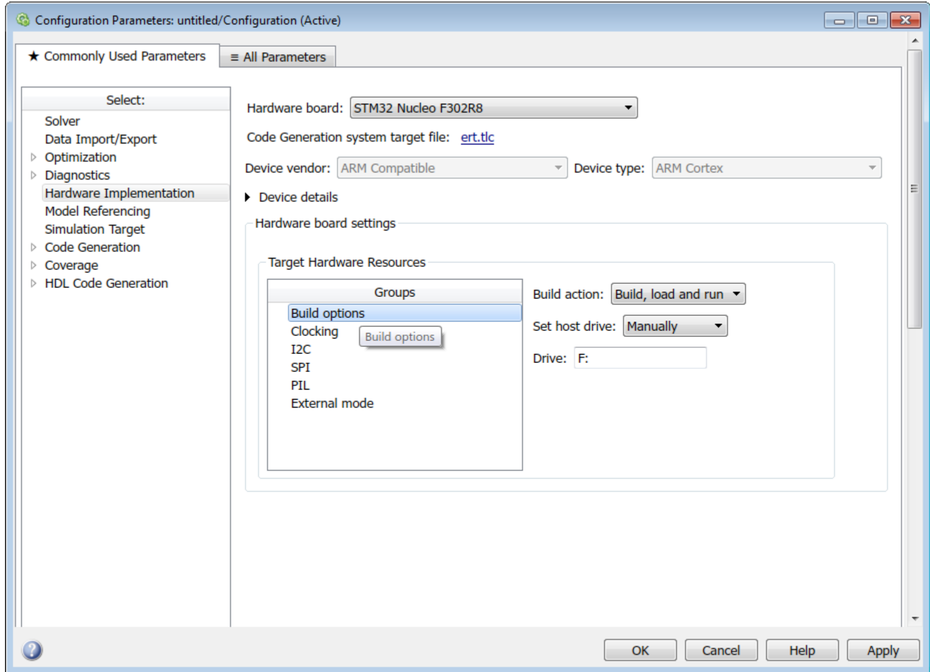

| In this section... |
| --- |
| "Hardware Implementation Pane Overview" on page 13-148 |
| "Build options" on page 13-159 |
| "Clocking" on page 13-160 |
| "I2C" on page 13-161 |
| "PIL" on page 13-162 |
| "SPI" on page 13-163 |
| "External mode" on page 13-164 |



## Hardware Implementation Pane Overview

Configure the parameters for properties of the physical hardware, such as peripherals.

# Build options




Use the build option to specify how the build process should take place during code generation.

### Build action

Specify if you want only build or build, load and run actions during code generation.

- `Build` – Select this option if you want to build the code during the build process.
- `Build, load and run` – Select this option to build, load, and to run the generated code during the build process.

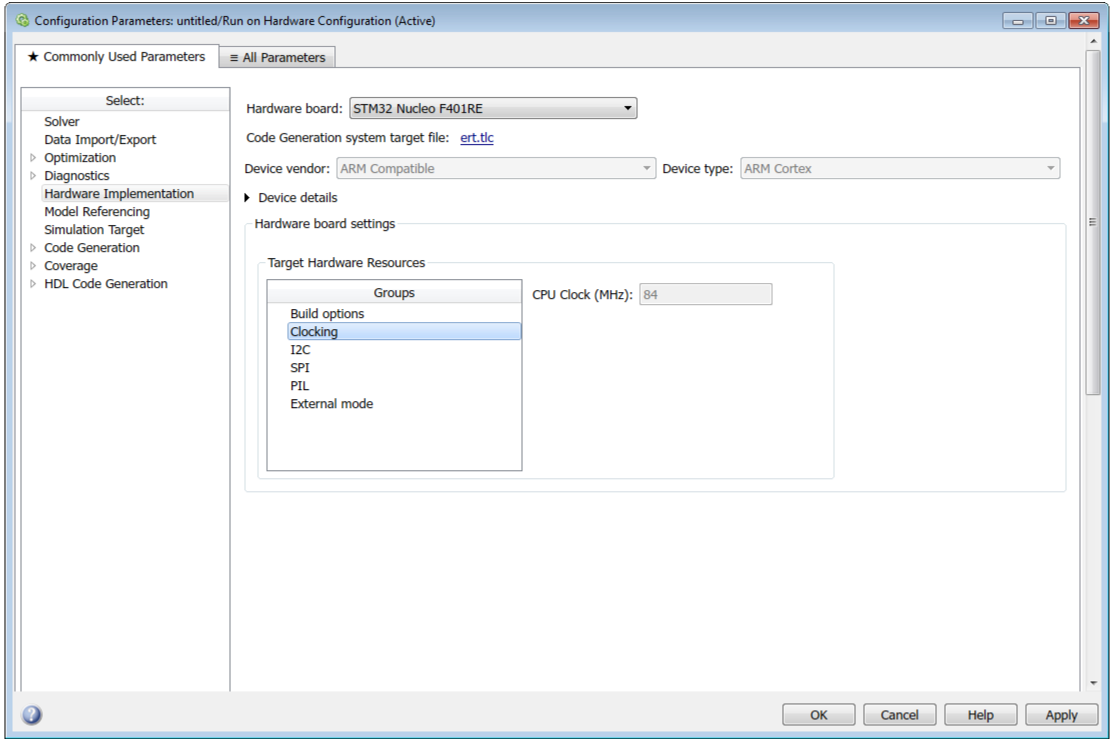
### Set host drive

Specify the drive on which the generated output bin file should be copied.

### Drive

Specify the drive letter on which you want to copy the generated output bin file.

## Clocking

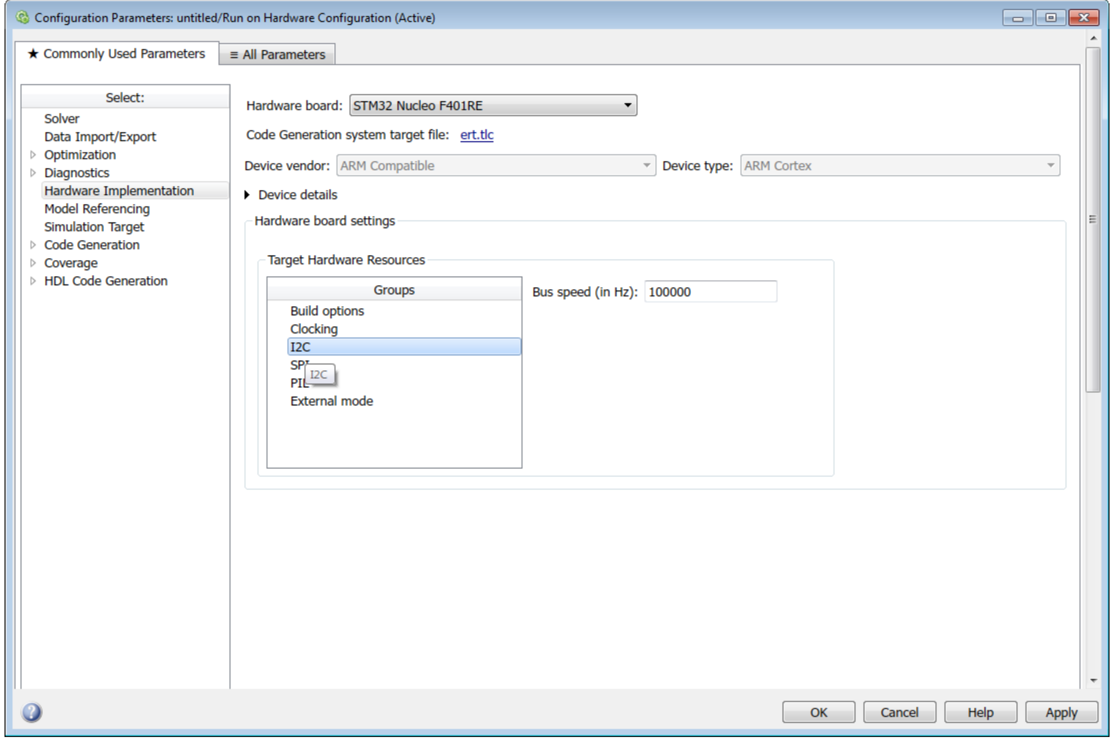



Use the clocking option to achieve the CPU Clock rate specified.

**CPU Clock (MHz)**

The CPU clock rate.

## I2C

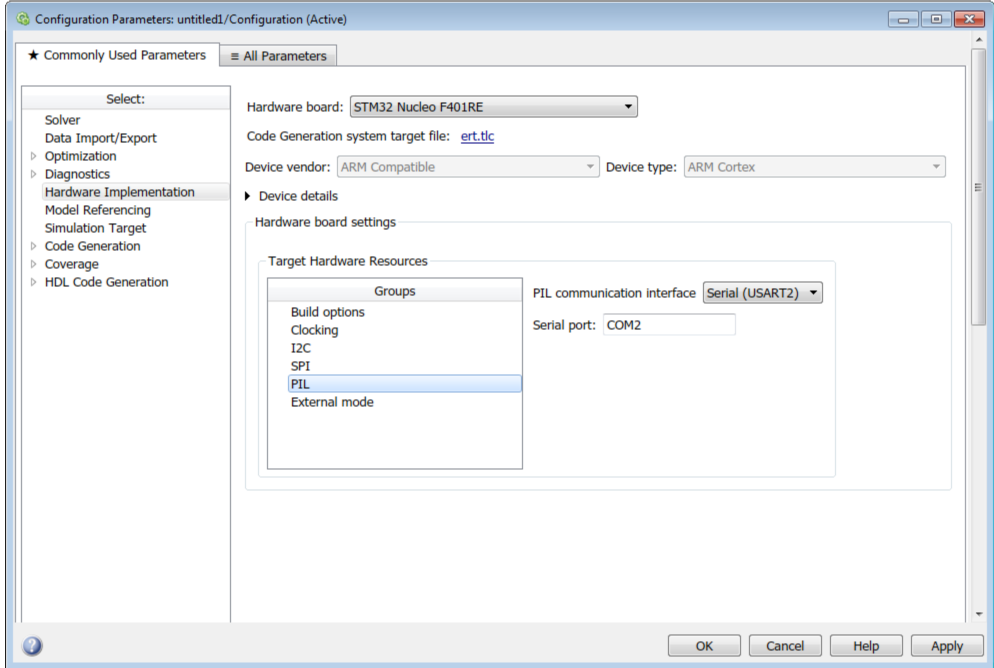



Use the PIL options to set PIL (Processor-in-the-Loop) communications parameters.

**Bus speed (in Hz)**

Use the I2C option to set the bus speed parameter. The bus speed determines the rate of data communication between the peripherals that are connected by the I2C bus.

## PIL




Use the PIL options to set PIL (Processor-in-the-Loop) communications parameters.

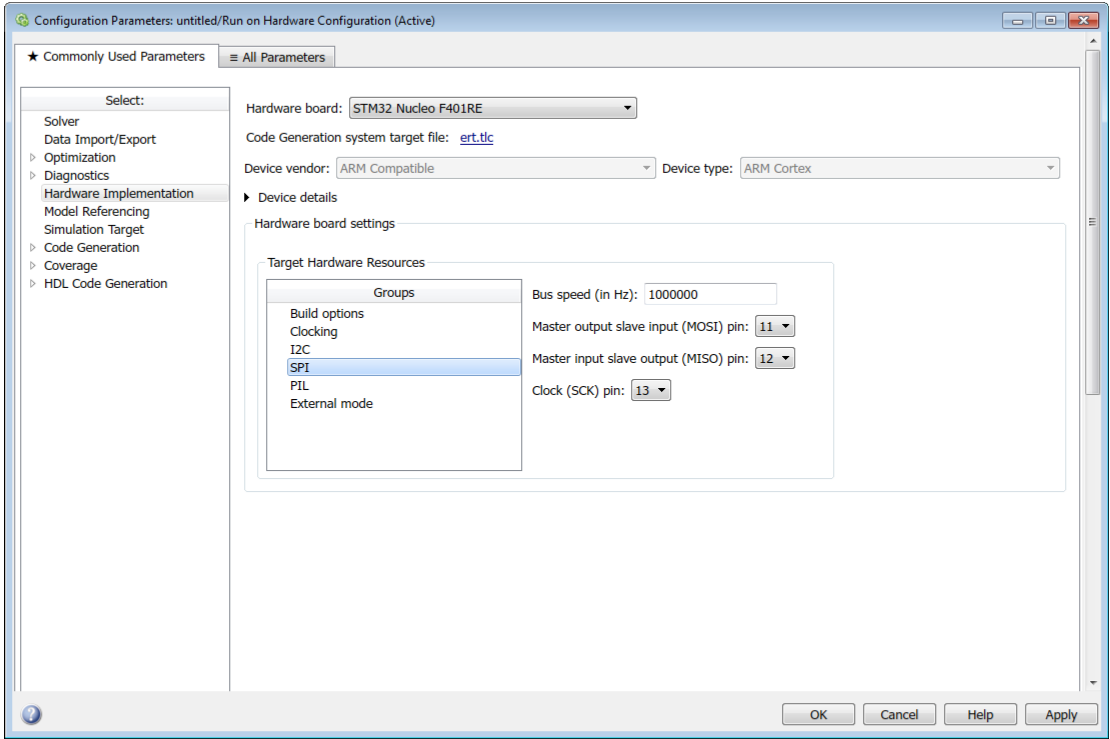
**PIL communication interface**

The serial port used PIL communication.

**Serial port**

Enter the serial port for PIL communication.

## SPI




Use the PIL options to set PIL (Processor-in-the-Loop) communications parameters.

**Bus speed (in Hz)**

The serial port used PIL communication.

**Master output slave input (MOSI) pin**

Specify the pin that connects the master output to the slave input.

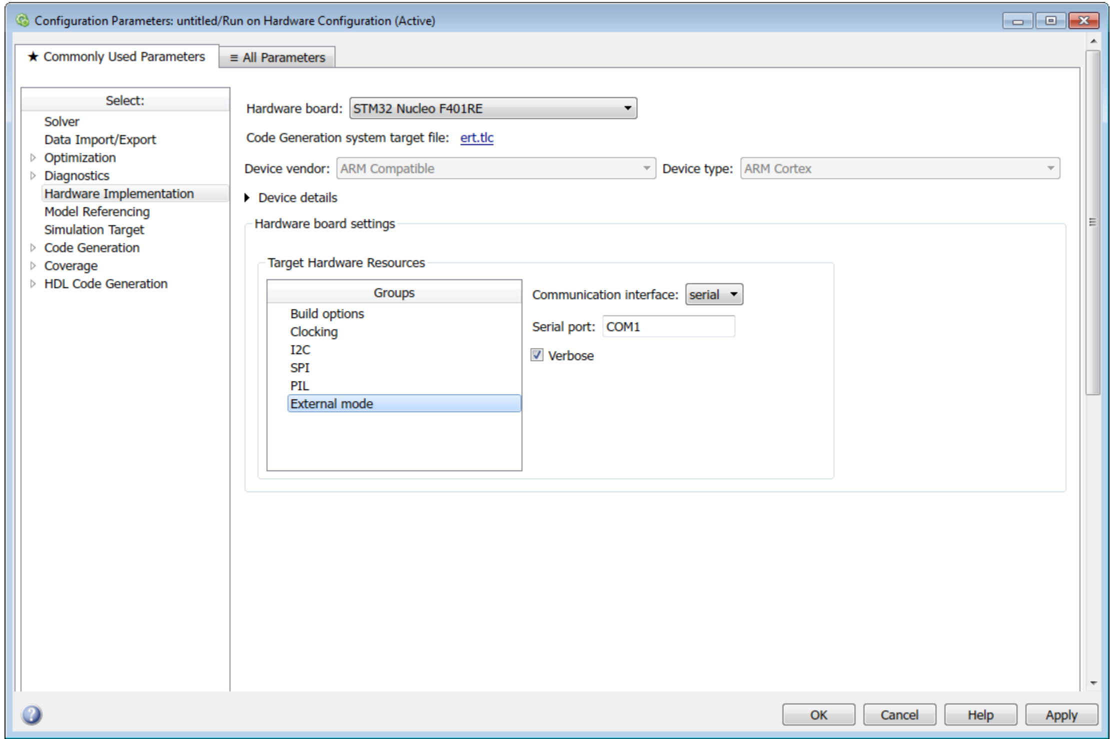
**Master input slave input (MISO) pin**

Specify the pin that connects the slave output to the master input.

**Clock (SCK) pin**

Specify the clock pin for SPI communication.

## External mode




**Communication interface**

Use the 'serial' option to run your model in the External mode with serial communication.

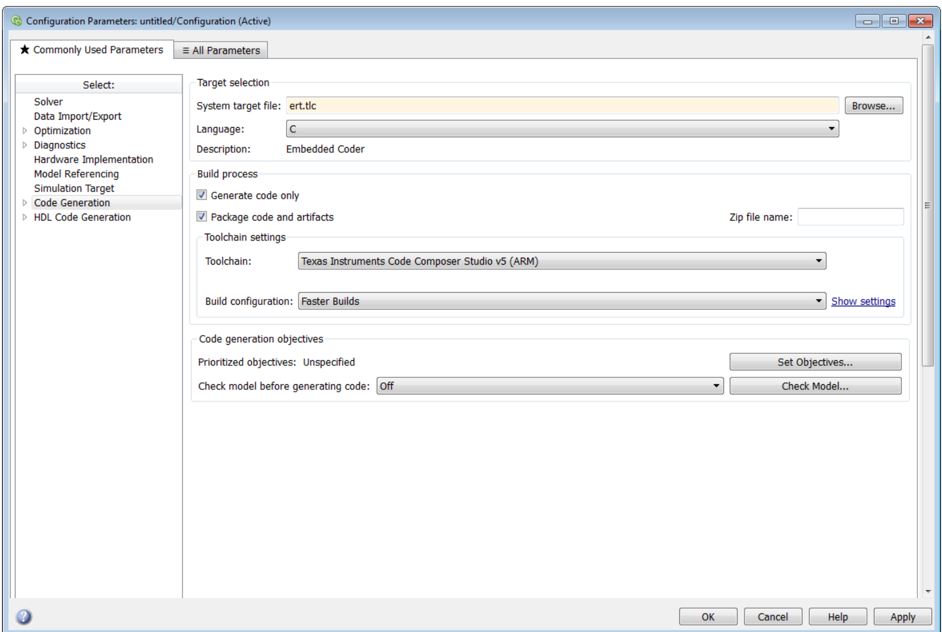
**Serial port**

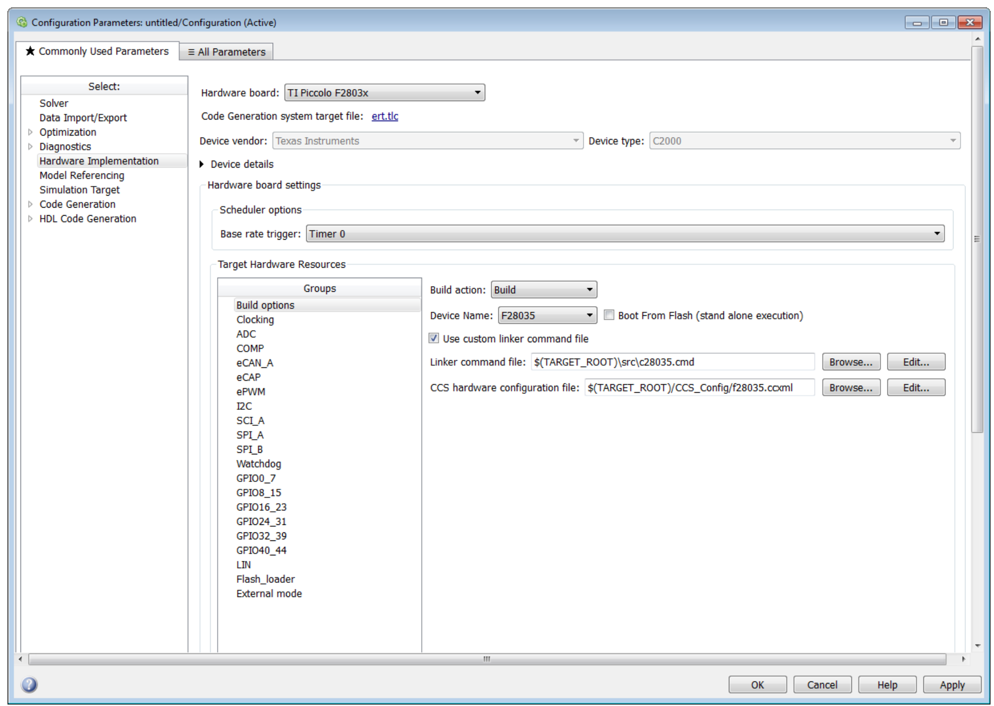
Enter the serial port used by the target hardware.

**Verbose**

Select this check box to view the External Mode execution progress and updates in the Diagnostic Viewer or in the MATLAB command window.

# Hardware Implementation Pane: Texas Instruments C2000






**In this section...**

"Hardware Implementation Pane Overview" on page 13-167
"Texas Instruments C2000 Settings" on page 13-167
"C28x-Scheduler options" on page 13-169
"C28x-Build options" on page 13-171
"C28x-Clocking" on page 13-175
"C28x-ADC" on page 13-178
"C28x-DAC" on page 13-181
"C28-COMP" on page 13-182
"C28x-eCAN_A, C28x-eCAN_B" on page 13-183
"C28x-eCAP" on page 13-186
"C28x-ePWM" on page 13-189

| In this section... |
| --- |
| "C28x-I2C" on page 13-192 |
| "C28x-SCI_A, C28x-SCI_B, C28x-SCI_C" on page 13-198 |
| "C28x-SPI_A, C28x-SPI_B, C28x-SPI_C, C28x-SPI_D" on page 13-201 |
| "C28x-eQEP" on page 13-204 |
| "C28x-Watchdog" on page 13-206 |
| "C28x-GPIO" on page 13-208 |
| "C28x-Flash_loader" on page 13-213 |
| "C28x-DMA_ch[#]" on page 13-215 |
| "C28x-LIN" on page 13-224 |
| "External mode" on page 13-230 |
| "Execution profiling" on page 13-231 |

## Hardware Implementation Pane Overview

Configure the parameters for properties of the physical hardware, such as peripherals for Texas Instruments C2000.

### See Also

- "Hardware Implementation Pane Overview" on page 13-167

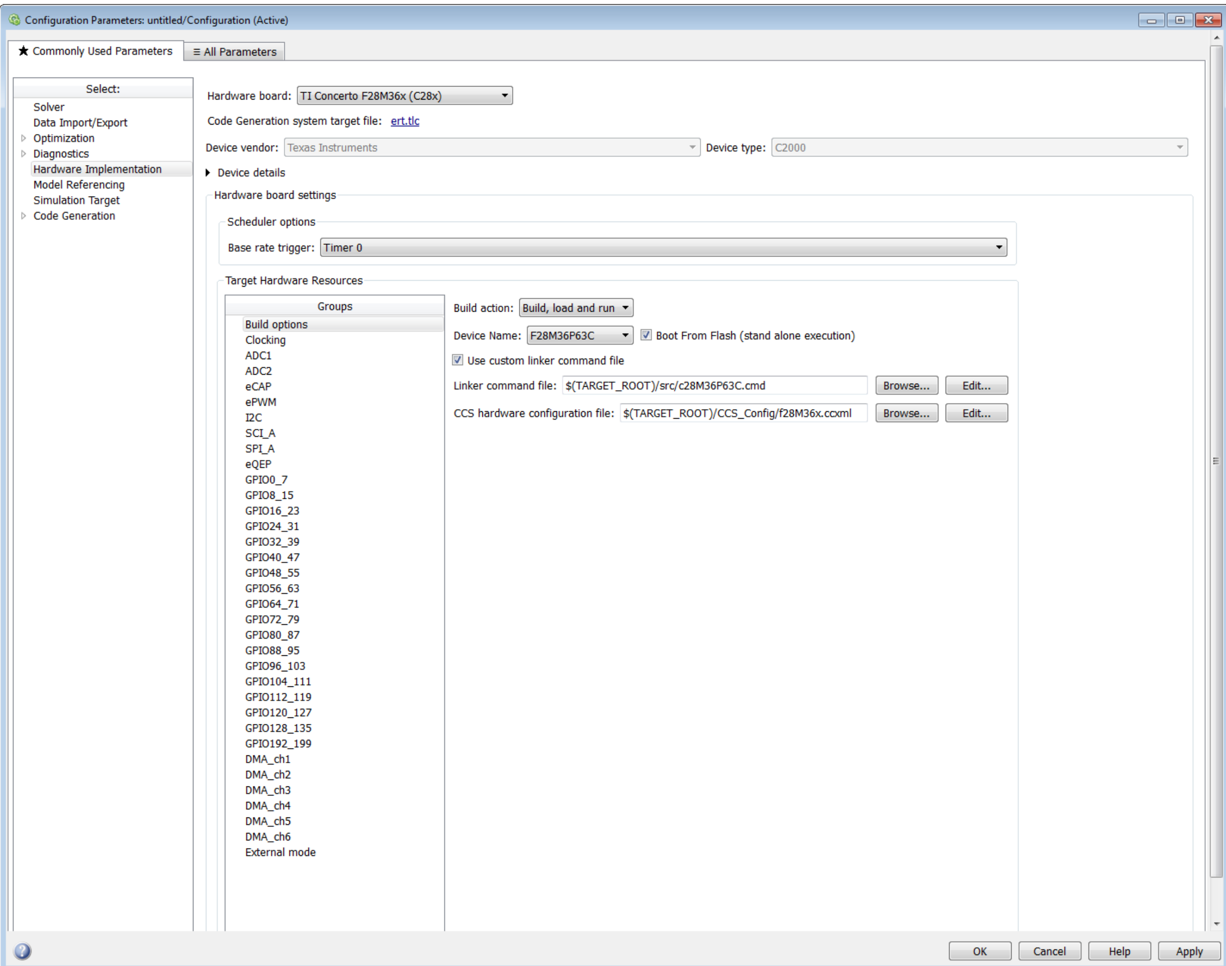
## Texas Instruments C2000 Settings

The following table provides links to topics for Texas Instruments C2000 processors.

| Peripheral Name | Description |
| --- | --- |
| "C28x-Build options" on page 13-171 | Build parameters for the build process |
| "C28x-Clocking" on page 13-175 | Clocking parameters to adjust clock settings and match custom oscillator frequencies |
| "C28x-ADC" on page 13-178 | Analog-to-Digital Converter (ADC) parameters |
| "C28-COMP" on page 13-182 | Parameters to assign COMP pins to GPIO pins |

| Peripheral Name | Description |
|---|---|
| "C28x-eCAN_A, C28x-eCAN_B" on page 13-183 | Enhanced Controller Area Network (eCAN) parameters for modules A or B |
| "C28x-eCAP" on page 13-186 | Enhanced Capture (eCAP) parameters for pin mapping to GPIO |
| "C28x-ePWM" on page 13-189 | Enhanced Pulse Width Modulation (ePWM) parameters for pin mapping to GPIO |
| "C28x-I2C" on page 13-192 | Inter-Integrated Circuit (I2C) parameters for communications |
| "C28x-SCI_A, C28x-SCI_B, C28x-SCI_C" on page 13-198 | Serial Communications Interface (SCI) parameters for communications with modules A, B, or C |
| "C28x-SPI_A, C28x-SPI_B, C28x-SPI_C, C28x-SPI_D" on page 13-201 | Serial Peripheral Interface (SPI) parameters for communications with module A, B, C, or D |
| "C28x-eQEP" on page 13-204 | Enhanced Quadrature Encoder Pulse (eQEP) parameters for pin mapping to GPIO |
| "C28x-Watchdog" on page 13-206 | Watchdog enable/disable and timing |
| "C28x-GPIO" on page 13-208 | General Purpose Input Output (GPIO) parameters for input qualification types |
| "C28x-Flash_loader" on page 13-213 | Flash memory loader/programmer |
| "C28x-DMA_ch[#]" on page 13-215 | Direct Memory Access (DMA) parameters for channels 1 to N |
| "C28x-LIN" on page 13-224 | Local Interconnect Network (LIN) parameters for communications |

## C28x-Scheduler options




### Scheduler interrupt source

The source of the scheduler interrupt. The different scheduler options available are:

- `Timer 0` - The default option to schedule all synchronous rates present in your model with CPU Timer 0. When you select this option, the CPU Timer 0 is set to honor the base rate of the model.

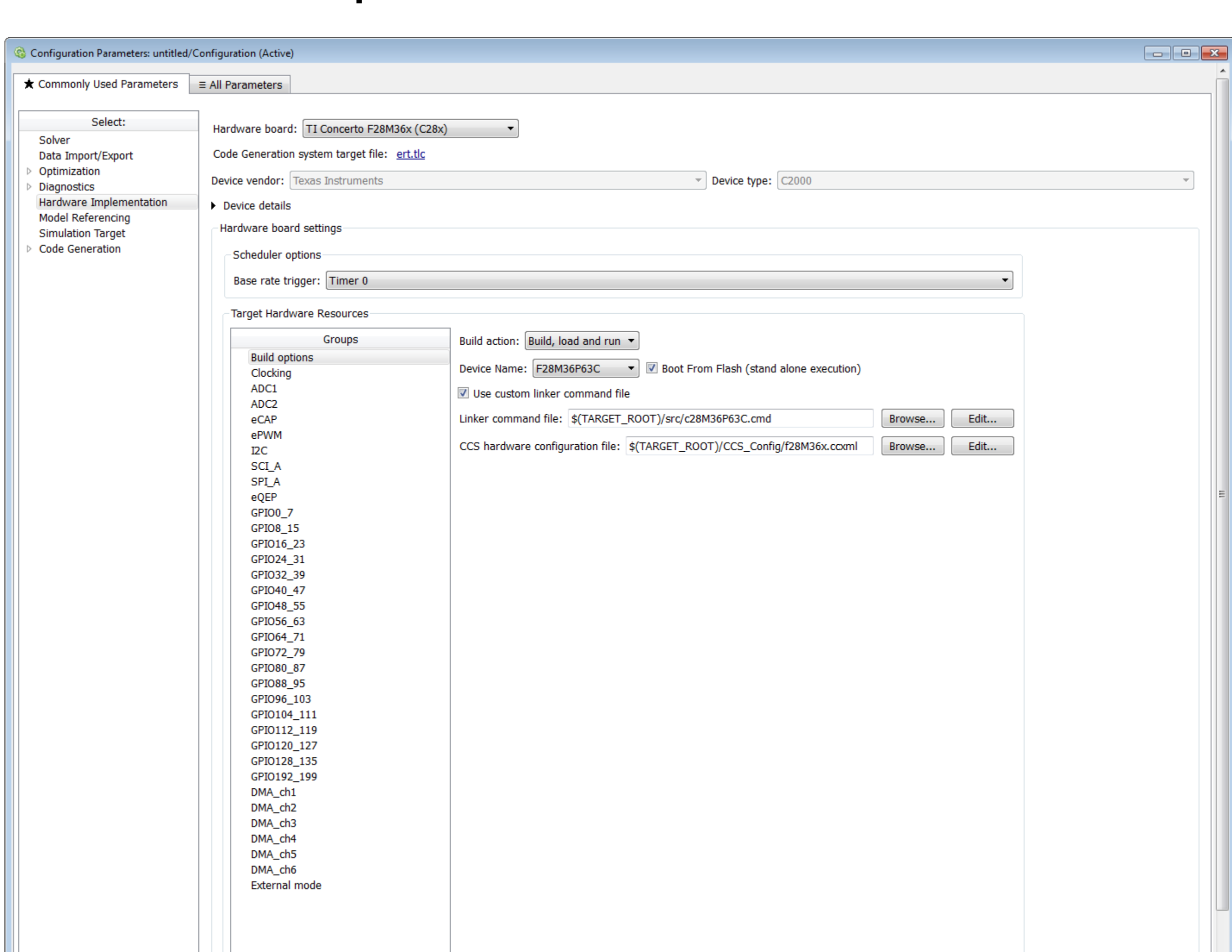
- `ADCINT1` - The option to schedule all synchronous rates present in your model with ADC interrupt 1 (ADCINT1). When you select this option, make sure that ADCINT1 triggers periodically at base rate used in the model.
- `ADCINT2` - The option to schedule all synchronous rates present in your model with ADC interrupt 2 (ADCINT2). When you select this option, make sure that ADCINT2 triggers periodically at base rate used in the model.

---

**Warning** Replacing the default scheduler interrupt source Timer 0 with ADCINT1 or ADCINT2 is an advanced maneuver. It is your responsibility to ensure that you configure ADCINT1 or ADCINT2 to trigger periodically at the specified base rate. If the ADCINT1 or ADCINT2 does not trigger periodically or triggers at a different rate than the base rate, the model execution on the target is unpredictable.

---

# C28x-Build options




Use the build options to specify how the build process should take place.

### Build action

The option to specify if you want only 'build' or 'build, load, and run' action during the build process. The `build, load, and run` option is supported only for TI Code Composer Studio v4/5/6 (C2000) tool chain.

If you select `build, load and run` option, you must provide the required CCS hardware configuration file. The TI Concerto F28M35x/ F28M36x processors support only CCS v5 and the above versions.

**Device Name**

The option to select a particular device from the selected processor family in the Target hardware parameter on the Code Generation pane.

**Boot From Flash (stand alone execution)**

The option to specify if the application has to load to the flash. If you do not select this option, the application loads to the RAM.

**Select CPU**

Select a CPU core to run the generated code on a dual-core processor, such as F2837xD. Make sure you create two different models to run in different CPU cores. Use the appropriate options of this parameter to generate the code for your models.

There are some additional parameters that appear in **Target Hardware Resources** when you select `CPU1` option. The clock settings and the CPU assignment for F2837xD processor peripherals are available only when you select `CPU1` option. If you select `CPU2`, make sure that you specify the same clock frequency as you have specified for the CPU1 model.

The GPIO pin configuration registers are available only for CPU1. For the GPIOs used in CPU2 model, you should configure the GPIO pins from the CPU1.

For a Simulink generated code that you run in CPU1, the GPIO pin configuration for CPU2 takes place automatically. However, make sure you do not use the same GPIO pins in both CPU1 and CPU2.

For a handwritten code that you run in CPU1, make sure you configure the GPIO pins for CPU2 from CPU1.

**Boot From Flash (stand alone execution)**

The option to specify if the application has to load to the flash. If you do not select this option, the application loads to the RAM.

**Use custom linker command file**

The option to indicate that the custom linker command file must be used during the build action. Select this option, if you have your own custom linker file, which you can specify in Linker command file parameter. If you do not select this option, based on the device you have selected, a default custom linker command file will be used.

**Linker command file**

The path to memory description file that is required during linking. For each family of TI processor selected under 'Target Hardware', one linker command file will be selected automatically.

For different variant of processor, you can select from the 'src' folder inside the Support Package installation path. You can also create custom linker command file and select the file path using **Browse**

**CCS hardware configuration file**

The Code Composer Studio file required for downloading the application on the hardware. Select one of the .ccxml files from the folder 'CCS_Config' folder under Support Package installation folder.

Select the file you created using **Browse**. You can also edit the .ccxml file using the **Edit** button.

The .ccxml files provided with Embedded Coder Support Package for Texas Instruments C2000 Processors are as follows:

- f28027.ccxml – TI F28027 with Texas Instruments XDS100v1 USB Emulator
- f28035.ccxml – TI F28035 with Texas Instruments XDS100v1 USB Emulator
- f28069.ccxml – TI F28069 with Texas Instruments XDS100v1 USB Emulator
- f2808.ccxm l– TI F2808 with Texas Instruments XDS100v1 USB Emulator
- f2808_eZdsp.ccxml – F2808 Spectrum Digital DSK-EVM-eZdsp onboard USB Emulator
- f28044.ccxml – TI F28044 with Texas Instruments XDS100v1 USB Emulator
- f28335.ccxml – TI F28335 with Texas Instruments XDS100v1 USB Emulator
- f28335_eZdsp.ccxml – F28335 Spectrum Digital DSK-EVM-eZdsp onboard USB Emulator
- f2812_BH2000.ccxml – Blackhawk USB2000 Controller for F2812 eZDSP
- f28x_generic.ccxml – Generic Texas Instruments XDS100v1 USB Emulator
- f28x_ezdsp_generic.ccxml – Generic Spectrum Digital eZdsp onboard USB Emulator
- f28x_ezdsp_generic.ccxml – Generic Spectrum Digital eZdsp onboard USB Emulator

- f28377S.ccxml – TI F2837xS with Texas Instruments XDS100v2 USB Emulator
- f28075.ccxml – TI F2807x with Texas Instruments XDS100v2 USB Emulator
- f28377D.ccxml – TI F2837xD with Texas Instruments XDS100v2 USB Emulator

The .ccxml files provided with Embedded Coder Support Package for Texas Instruments C2000 F28M3x Concerto™ Processors are as follows:

- f28M35x.ccxml – Texas Instruments XDS100v2 USB Emulator_0
- f28M36x.ccxml – Texas Instruments XDS100v2 USB Emulator_0

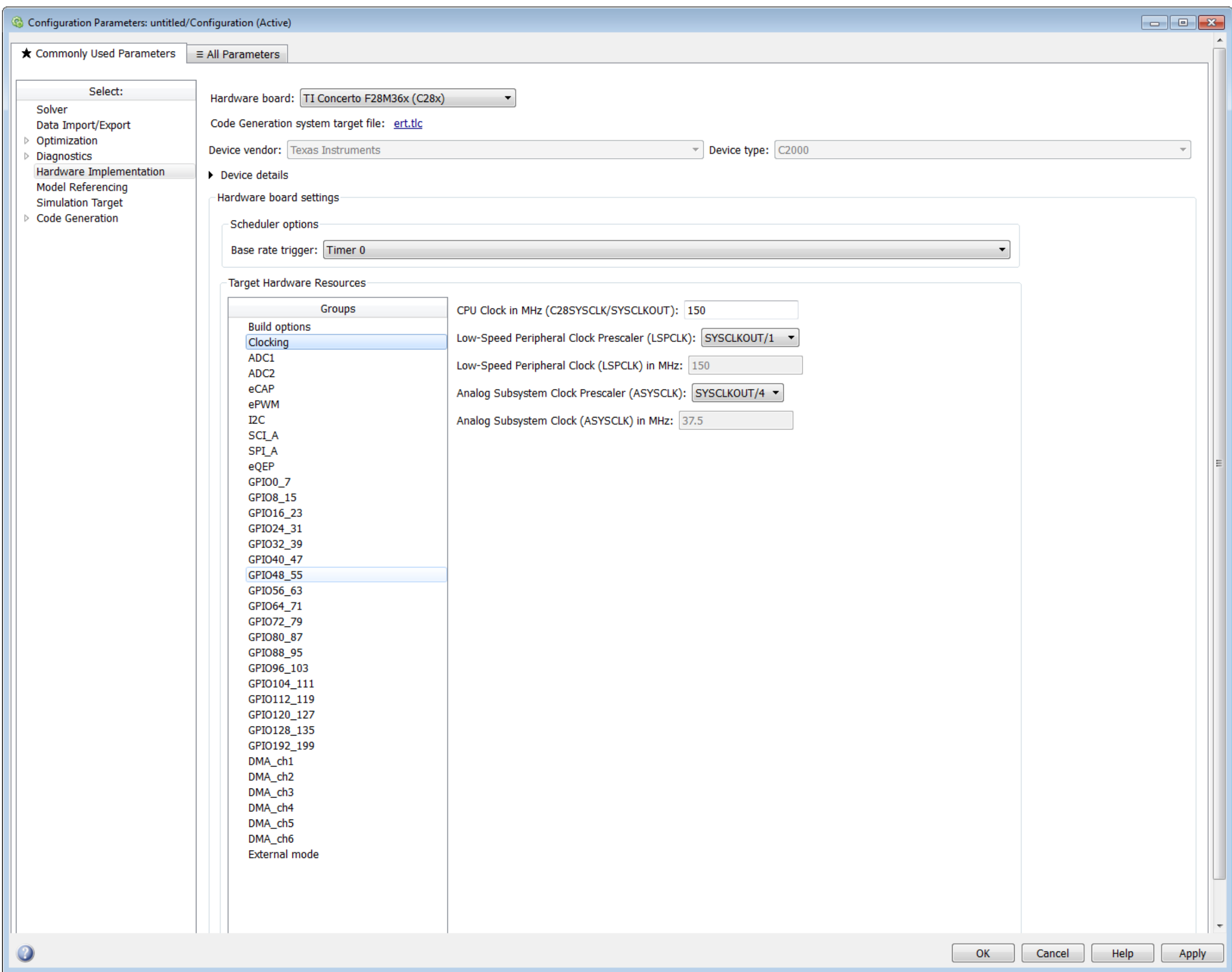
### Enable DMA to access ePWM Registers instead of CLA

The option that you can select to enable the DMA to access ePWM registers instead of CLA. This option is available only for F2806x processors.

### Remap ePWMs for DMA access (Requires silicon revision A and above)

The option that you can select to remap ePWMs registers for DMA access. This option is available only for F2833x processors.

## C28x-Clocking




Use the clocking options to help you achieve the CPU Clock rate specified on the board. The default clocking values run the CPU clock (CLKIN) at its maximum frequency. The parameters use the external oscillator frequency on the board (OSCCLK) that is recommended by the processor vendor.

You can get feedback on the closest achievable SYSCLKOUT value with the specified Oscillator clock frequency by selecting the **Auto set PLL based on OSCCLK and CPU**

**clock** check box. Alternatively, you can manually specify the PLL value for the SYSCLKOUT value calculation.

Change the clocking values if:

- You want to change the CPU frequency.
- The external oscillator frequency differs from the value recommended by the manufacturer.

To determine the CPU frequency (CLKIN), use the following equation:

CLKIN = (OSCCLK * PLLCR) / (DIVSEL or CLKINDIV)

- CLKIN is the frequency at which the CPU operates, also known as the CPU clock.
- OSCCLK is the frequency of the oscillator.
- **PLLCR** is the PLL Control Register value.
- **CLKINDIV** is the Clock in Divider.
- **DIVSEL** is the Divider Select.

The availability of the DIVSEL or CLKINDIV parameters changes depending on the processor that you select. If neither parameter is available, use the following equation:

CLKIN = (OSCCLK * PLLCR) / 2

In case of Concerto C28x processor, make sure to match the Achievable C28x SYSCLK in MHz with the value entered here.

In the **CPU clock** parameter, enter the resulting CPU clock frequency (CLKIN).

For more information, see the “PLL-Based Clock Module” section in the Texas Instruments *Reference Guide* for your processor.

**Use internal oscillator**

Use the internal zero pin oscillator on the CPU. This parameter is enabled by default.

**Oscillator clock (OSCCLK) frequency in MHz**

The oscillator frequency that is used in the processor. This parameter is not available for TI Concerto F28M35x/ F28M36x processors.

**Auto set PLL based on OSCCLK and CPU clock**

The option that helps you set the PLL control register value automatically. When you select this check box, the values in the PLLCR, DIVSEL , and the Closest achievable

SYSCLKOUT in MHz parameters are automatically calculated based on the **CPU Clock** value entered on the Board. This parameter is not available for TI Concerto F28M35x/ F28M36x processors.

**PLL control register (PLLCR)**

If you select the **Auto set PLL based on OSCCLK and CPU clock** check box, the auto calculated control register value achieves the specified CPU Clock value, based on the Oscillator clock frequency. Otherwise, you can select a value for PLL control register. This parameter is not available for TI Concerto F28M35x/ F28M36x processors.

**Clock divider (DIVSEL)**

If you select the **Auto set PLL based on OSCCLK and CPU clock** check box, the auto calculated clock divider value achieves the specified CPU Clock value based on the Oscillator clock frequency. Otherwise, you can select a value for Clock divider (DIVSEL). This parameter is not available for TI Concerto F28M35x/ F28M36x processors.

**Closest achievable SYSCLKOUT in MHz = (OSCCLK*PLLCR)/DIVSELClosest achievable SYSCLKOUT in MHz = (OSCCLK*PLLCR)/CLKINDIV**

The auto calculated feedback value that matches most closely to the desired CPU Clock value on the board, based on the values of OSCCLK, PLLCR, and the DIVSEL. This parameter is not available for TI Concerto F28M35x/ F28M36x processors.

**Low-Speed Peripheral Clock Prescaler (LSPCLK)**

The value by which to scale the LSPCLK. This value is based on the SYSCLKOUT.

**Low-Speed Peripheral Clock (LSPCLK) in MHz**

This value is calculated based on LSPCLK Prescaler. Example: SPI uses a LSPCLK.

**High-Speed Peripheral Clock Prescaler (HSPCLK)**

The value by which to scale the HSPCLK. This value is based on the SYSCLKOUT.

**High-Speed Peripheral Clock (HSPCLK) in MHZ**

This value is calculated based on HSPCLK Prescaler. Example: ADC uses a HSPCLK.

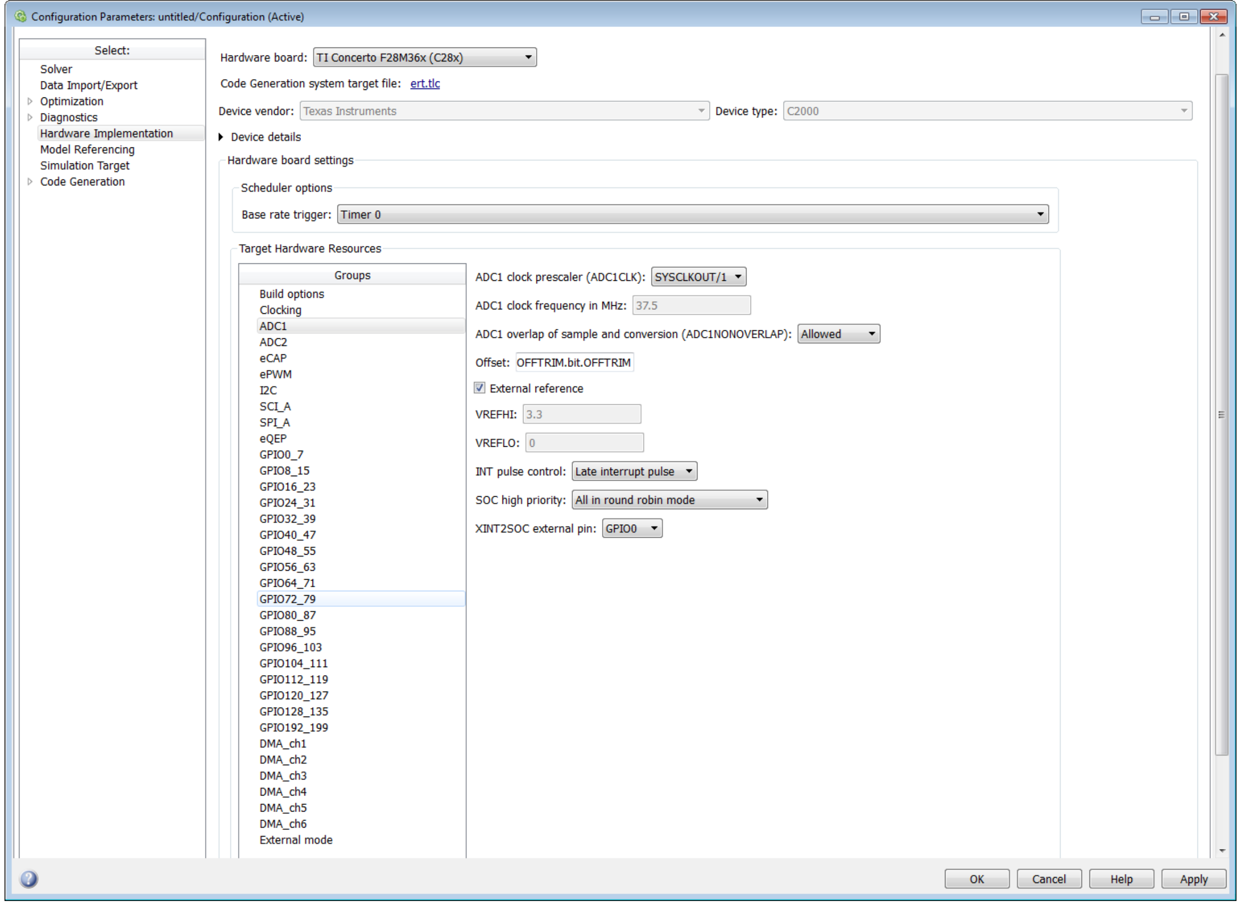
**Analog Common Interface Bus Clock (ACIB)**

The value by which to scale the bus clock. This option is available only for TI Concerto F28M35x/ F28M36x processors.

**Analog Common Interface Bus Clock (ACIB) in MHz**

This value is calculated based on the ACIB value. This option is available only for TI Concerto F28M35x/ F28M36x processors.

## C28x-ADC




The high-speed peripheral clock (HSPCLK) controls the internal timing of the ADC module. The ADC derives the operating clock speed from the HSPCLK speed in several prescaler stages. For more information about configuring these scalers, refer to "Configuring ADC Parameters for Acquisition Window Width".

You can set the following parameters for the ADC clock prescaler:

**Select the CPU core which controls ADC_x module**

Select the CPU core to control the ADC module.

**ADC clock prescaler (ADCCLK)**

The option to select the ADCCLK divider for processors c2802x, c2803x, c2806x, F28M3x, F2807x, or F2837x.

**ADC clock frequency in MHz**

The clock frequency for ADC. This is a read-only field and the value in this field is based on the value you select in **ADC clock prescaler (ADCCLK)**.

**ADC overlap of sample and conversion (ADC#NONOVERLAP)**

The option to enable or disable overlap of sample and conversion.

**ADC clock prescaler (ADCLKPS)**

The HSPCLK speed is divided by this 4-bit value as the first step in deriving the core clock speed of the ADC. The default value is 3.

**ADC Core clock prescaler (CPS)**

After dividing the HSPCLK speed by the **ADC clock prescaler (ADCLKPS)** value, setting the **ADC clock prescaler (ADCLKPS)** parameter to 1, the default value, divides the result by 2.

**ADC Module clock (ADCCLK = HSPCLK/ADCLKPS*2)/(CPS+1)) in MHz**

The clock to the ADC module and indicates the ADC operating clock speed.

**Acquisition window prescaler (ACQ_PS)**

This value does not directly alter the core clock speed of the ADC. It serves to determine the width of the sampling or acquisition period. The higher the value, the wider is the sampling period. The default value is 4.

**Acquisition window size ((ACQ_PS+1)/ADCCLK) in micro seconds/channel**

Acquisition window size determines for what time duration the sampling switch is closed. The width of SOC pulse is ADCTRL1[11:8] + 1 times the ADCLK period.

**Offset**

Enter the offset value.

**Use external reference 2.048VExternal reference**

By default, an internally generated band gap voltage reference supplies the ADC logic. However, depending on application requirements, you can enable the external reference so the ADC logic uses an external voltage reference instead. Select the check box to use a 2.048V external voltage reference.

**Use external reference**

By default, an internally generated band gap voltage reference supplies the ADC logic. However, depending on application requirements, you can enable the external

reference so the ADC logic uses an external voltage reference instead. Select the check box to use an external voltage reference.

**Continuous mode**

When the ADC generates an end of conversion (EOC) signal, generate an ADCINT# interrupt whether the previous interrupt flag has been acknowledged or not.

**ADC offset correction (OFFSET_TRIM: -256 to 255)**

The 280x ADC supports offset correction via a 9-bit value that it adds or subtracts before the results are available in the ADC result registers. Timing for results is not affected. The default value is `0`.

**VREFHIVREFLO**

When you disable the **Use external reference 2.048V** or **External reference** option, the ADC logic uses a fixed 0-volt to 3.3-volt input range and the software disables **VREFHI** and **VREFLO**. To interpret the ADC input as a ratiometric signal, select the **External reference** option. Then set values for the high voltage reference (**VREFHI**) and the low voltage reference (**VREFLO**). **VREFHI** uses the external ADCINA0 pin, and **VREFLO** uses the internal GND.

**INT pulse control**

Use this option to configure when the ADC sets ADCINTFLG ADCINTx relative to the SOC and EOC Pulses. Select `Late interrupt pulse` or `Early interrupt pulse`.

**SOC high priority**

Use this option to enable and configure **SOC high priority mode** . In `all in round robin mode`, the default selection, the ADC services each SOC interrupt in a numerical sequence.

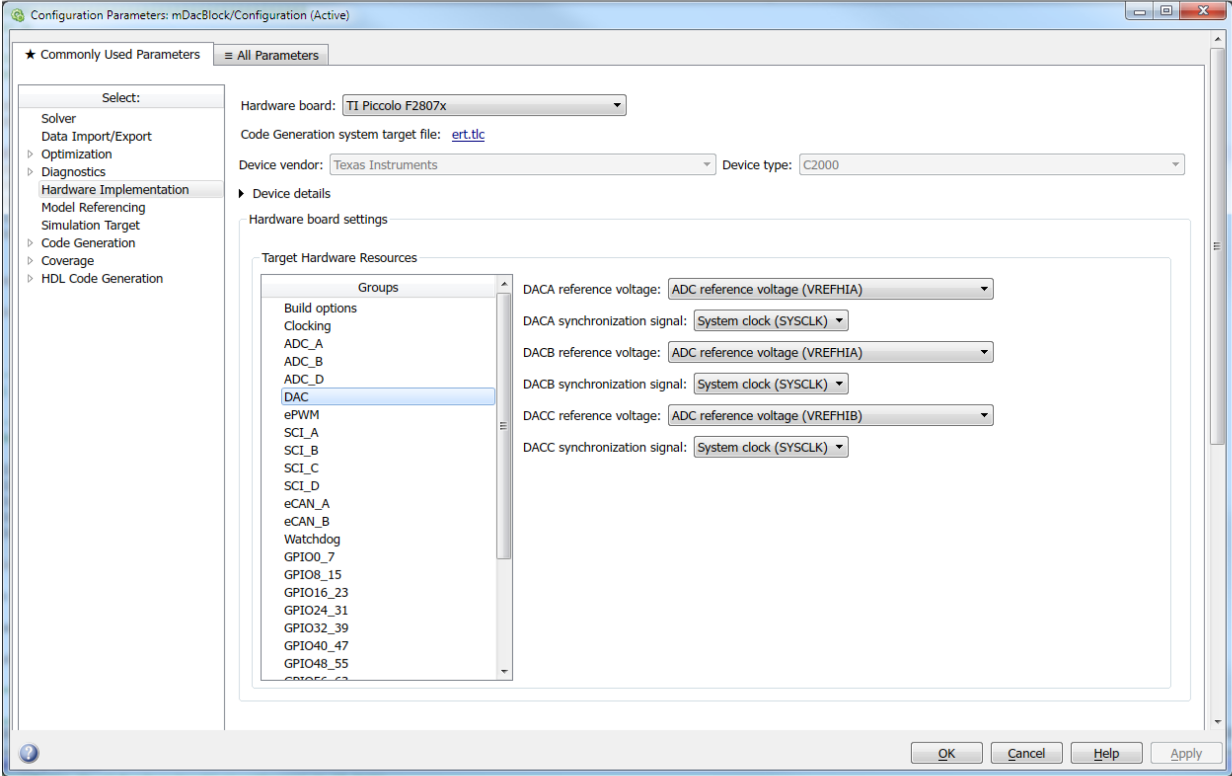
Choose one of the `high priority` selections to assign high priority to one or more of the SOCs. In this mode, the ADC operates in round robin mode until it receives a high priority SOC interrupt. The ADC finishes servicing the current SOC, services the high priority SOCs, and then returns to the next SOC in the round robin sequence.

For example, the ADC is servicing SOC8 when it receives a high priority interrupt on SOC1. The ADC completes servicing SOC8, services SOC1, and then services SOC9.

**XINT2SOC external pin**

Select the pin to which the ADC sends the XINT2SOC pulse.

## C28x-DAC




### DACA reference voltage/DACB reference voltage/DACC reference voltage

Select the reference voltage for the DAC channel A, B, or C.

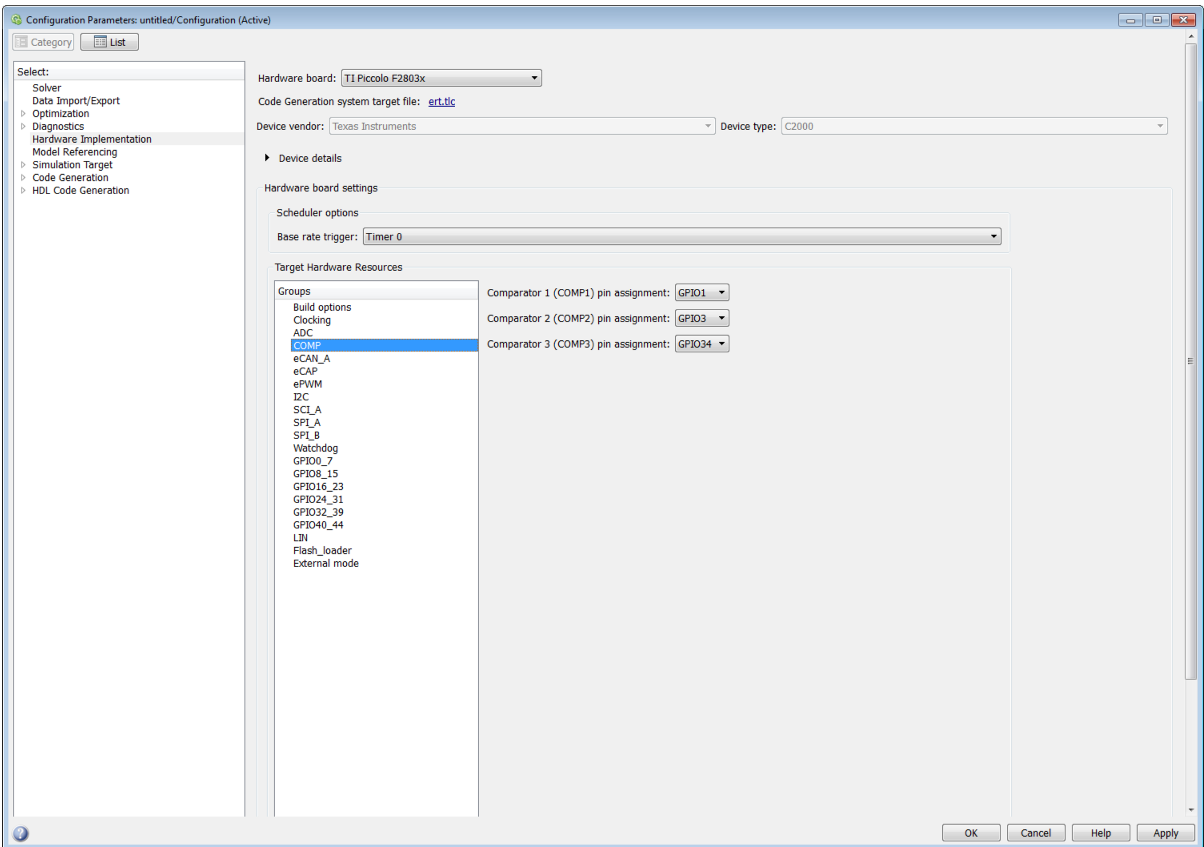
- • ADC reference voltage (VREFHIA/VREFHIB)– This is the reference voltage used for the ADC. You can use this as reference voltage VREFHIA for DAC A, DAC B and VREFHIB for DAC C.
- External reference voltage through ADCINB0 (VDAC)– This is a separate external reference voltage for DAC. Make sure you connect the ADCINB0 pin to the supply voltage.

**DACA synchronization signal/DACB synchronization signal/DACC synchronization signal**

Select the synchronization signal to load the value from the writable shadow register into the active register.

- • SYSCLK- This option is to load the value from the writable shadow register DACVALS into the active register DACVALA on the next clock cycle.
  - PWMSYNC1–12 – This option is to load the value from the writable shadow register DACVALS into the active register DACVALA on the next PWM synchronization event.

## C28-COMP



Assigns COMP pins to GPIO pins.

**Comparator 1 (COMP1) pin assignment**

Select an option from the list — `None`,`GPIO1`, `GPIO20`, `GPIO42`.

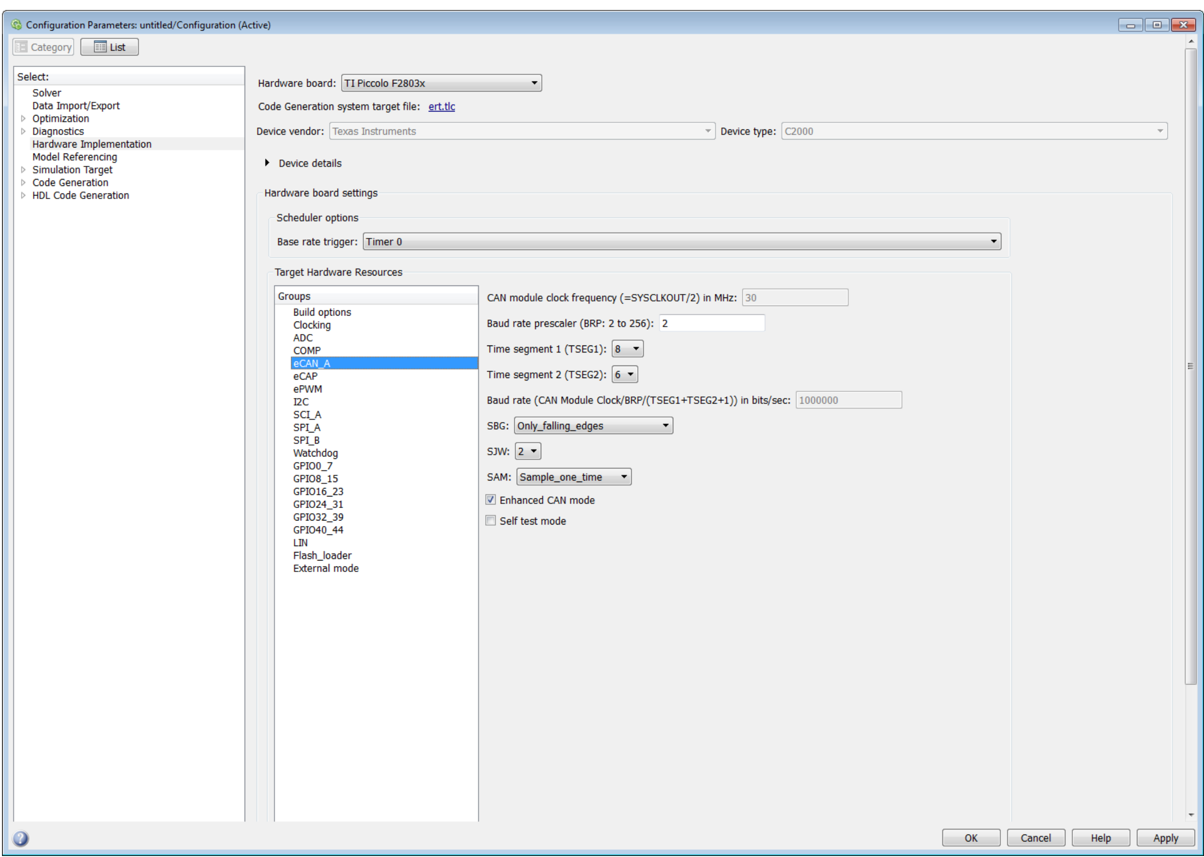
**Comparator 2 (COMP2) pin assignment**

Select an option from the list — `None`,`GPIO3`, `GPIO21`, `GPIO34`, `GPIO43`.

**Comparator 3 (COMP3) pin assignment**

Select an option from the list — `None`,`GPIO34`.

## C28x-eCAN_A, C28x-eCAN_B



For more help on setting the timing parameters for the eCAN modules, refer to "Configuring Timing Parameters for CAN Blocks" (Embedded Coder Support Package for Texas Instruments C2000 Processors). You can set the following parameters for the eCAN module:

**CAN module clock frequency (= SYSCLKOUT) in MHz:**

The clock to the enhanced CAN module. The CAN module clock frequency is equal SYSCLKOUT for processors such as c280x, c281x, c28044.

**CAN module clock frequency (=SYSCLKOUT/2) in MHz**

The clock to the enhanced CAN module. The CAN module clock frequency is equal to SYSCLKOUT/2 for processors such as piccolo, c2834x, c28x3x.

**Baud rate prescaler (BRP: 2 to 256)/Baud rate prescaler (BRP: 1 to 1024):**

Value by which to scale the bit rate.

**Time segment 1 (TSEG1):**

Sets the value of time segment 1, which, with **TSEG2** and **Baud rate prescaler**, determines the length of a bit on the eCAN bus. Valid values for **TSEG1** are from `1` through `16`.

**Time segment 2 (TSEG2):**

Sets the value of time segment 2, which, with **TSEG1** and **Baud rate prescaler**, determines the length of a bit on the eCAN bus. Valid values for **TSEG2** are from `1` through `8`.

**Baud rate (CAN Module Clock/BRP/(TSEG1 + TSEG2 +1)) in bits/sec:**

CAN module communication speed represented in bits/sec.

**SBG**

Sets the message resynchronization triggering. Options are `Only_falling_edges` and `Both_falling_and_rising_edges`.

**SJW**

Sets the synchronization jump width, which determines how many units of TQ a bit can be shortened or lengthened when resynchronizing.

**SAM**

Number of samples used by the CAN module to determine the CAN bus level. Selecting `Sample_one_time` samples once at the sampling point. Selecting `Sample_three_times` samples once at the sampling point and twice before at a distance of TQ/2. The CAN module makes a majority decision from the three points.

### Enhanced CAN Mode

To enable time-stamping and to use **Mailbox Numbers** 16 through 31 in the C2000 eCAN blocks, enable this parameter. Texas Instruments documentation refers to this “HECC mode”.

### Self test mode

If you set this parameter to `True`, the eCAN module goes to loopback mode. Loopback mode sends a “dummy” acknowledge message back without needing an acknowledge bit. The default is `False`.

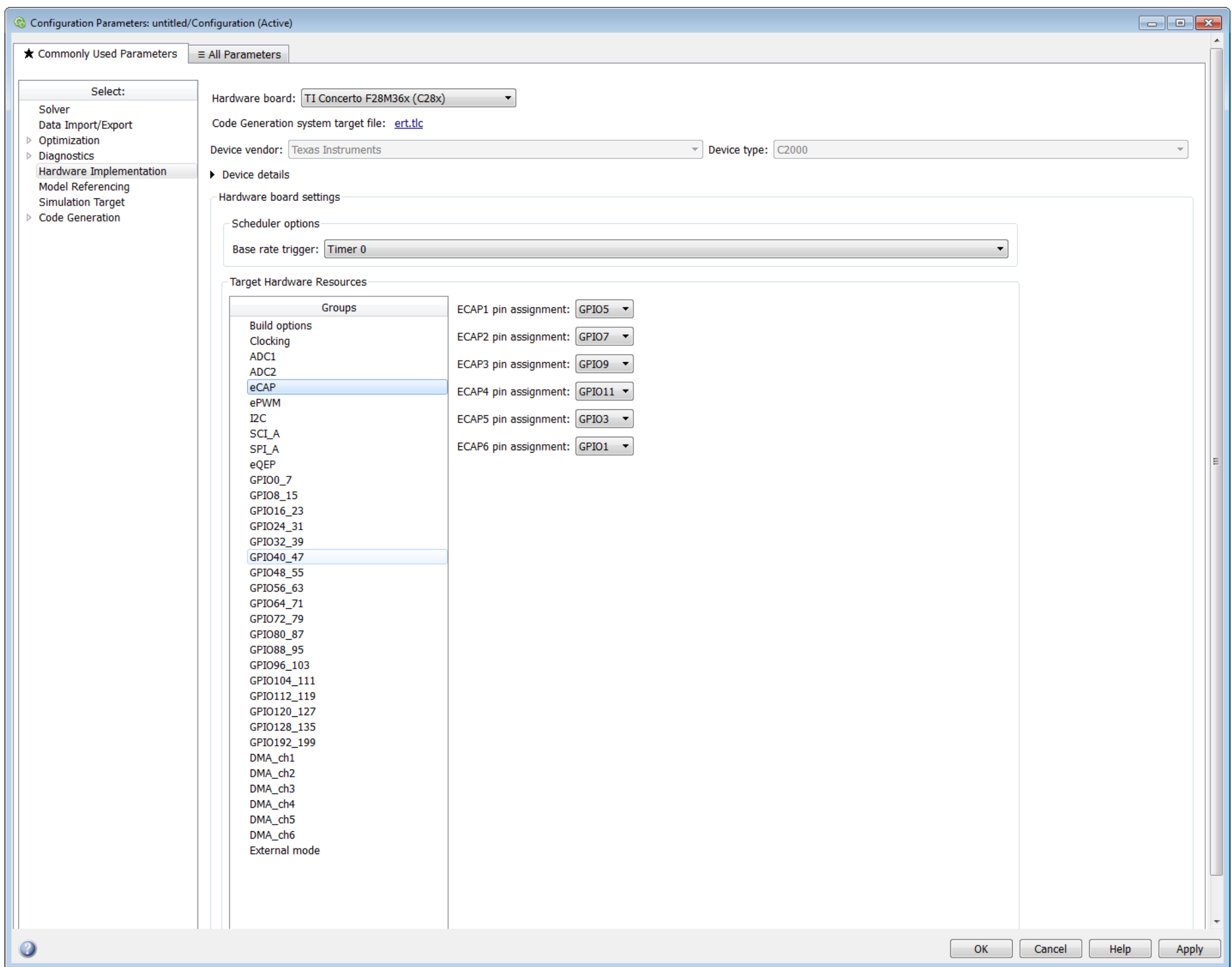
### Pin assignment (Tx)

Assigns the CAN transmit pin to use with the `eCAN_B` module. Possible values are `GPIO8`, `GPIO12`, `GPIO16`, and `GPIO20`.

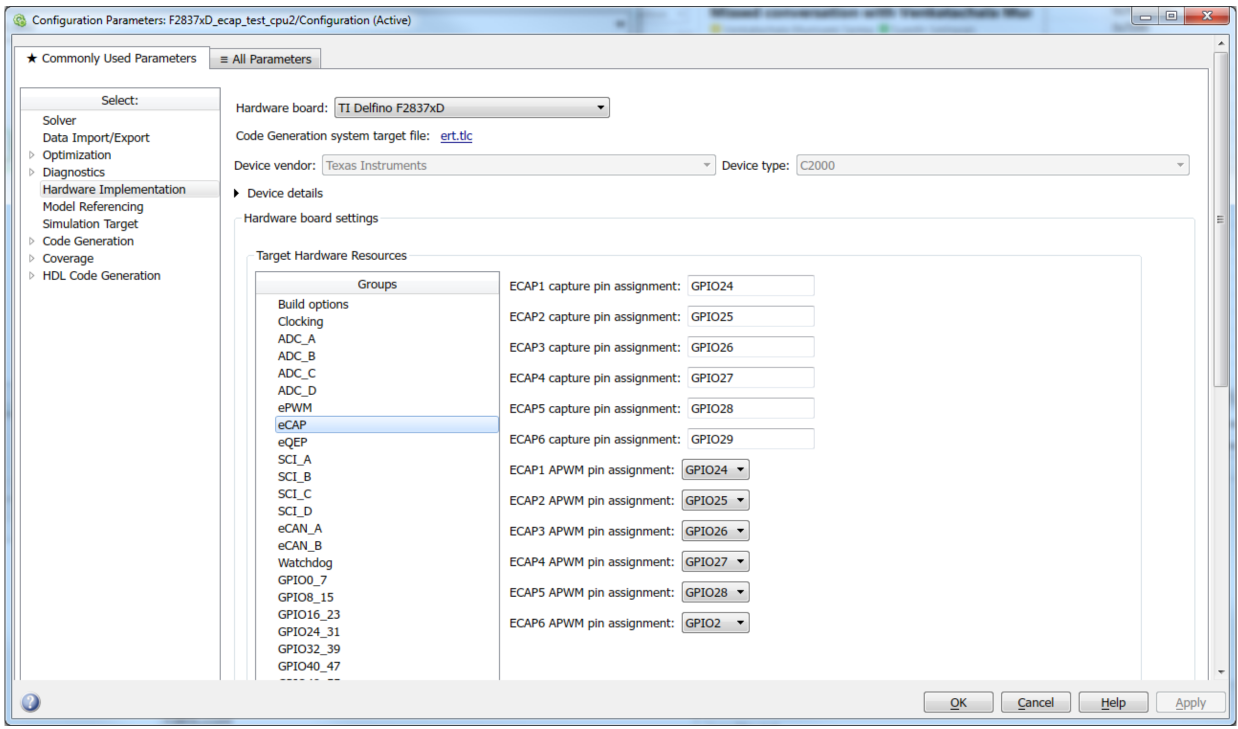
### Pin assignment (Rx)

Assigns the CAN receive pin to use with the `eCAN_B` module. Possible values are `GPIO10`, `GPIO13`, `GPIO17`, and `GPIO21`.

## C28x-eCAP




The following screen shows the eCAP parameters for F2837xS, F2837xD and F2807x processors.




Assigns eCAP pins to GPIO pins.

**ECAP1 pin assignment**

Select a GPIO pin for ECAP1 module.

**ECAP2 pin assignment**

Select a GPIO pin for ECAP2 module.

**ECAP3 pin assignment**

Select a GPIO pin for ECAP3 module.

**ECAP4 pin assignment**

Select a GPIO pin for ECAP4 module.

**ECAP5 pin assignment**

Select a GPIO pin for ECAP5 module.

**ECAP6 pin assignment**

Select a GPIO pin for ECAP6 module.

The parameters described below are only for F2837xS, F2837xD, and F2807x processors.

**ECAP1 capture pin assignment**

Select GPIO pin for ECAP1 module when using in eCAP (capture) operating mode.

**ECAP2 capture pin assignment**

Select a GPIO pin for ECAP2 module.

**ECAP3 capture pin assignment**

Select GPIO pin for ECAP3 module when using in eCAP (capture) operating mode.

**ECAP4 capture pin assignment**

Select GPIO pin for ECAP4 module when using in eCAP (capture) operating mode.

**ECAP5 capture pin assignment**

Select GPIO pin for ECAP5 module when using in eCAP (capture) operating mode.

**ECAP6 capture pin assignment**

Select GPIO pin for ECAP6 module when using in eCAP (capture) operating mode.

**ECAP1 APWM pin assignment**

Select GPIO pin for ECAP1 module when using in APWM operating mode.

**ECAP2 APWM pin assignment**

Select GPIO pin for ECAP2 module when using in APWM operating mode.

**ECAP3 APWM pin assignment**

Select GPIO pin for ECAP3 module when using in APWM operating mode.

**ECAP4 APWM pin assignment**

Select GPIO pin for ECAP4 module when using in APWM operating mode.

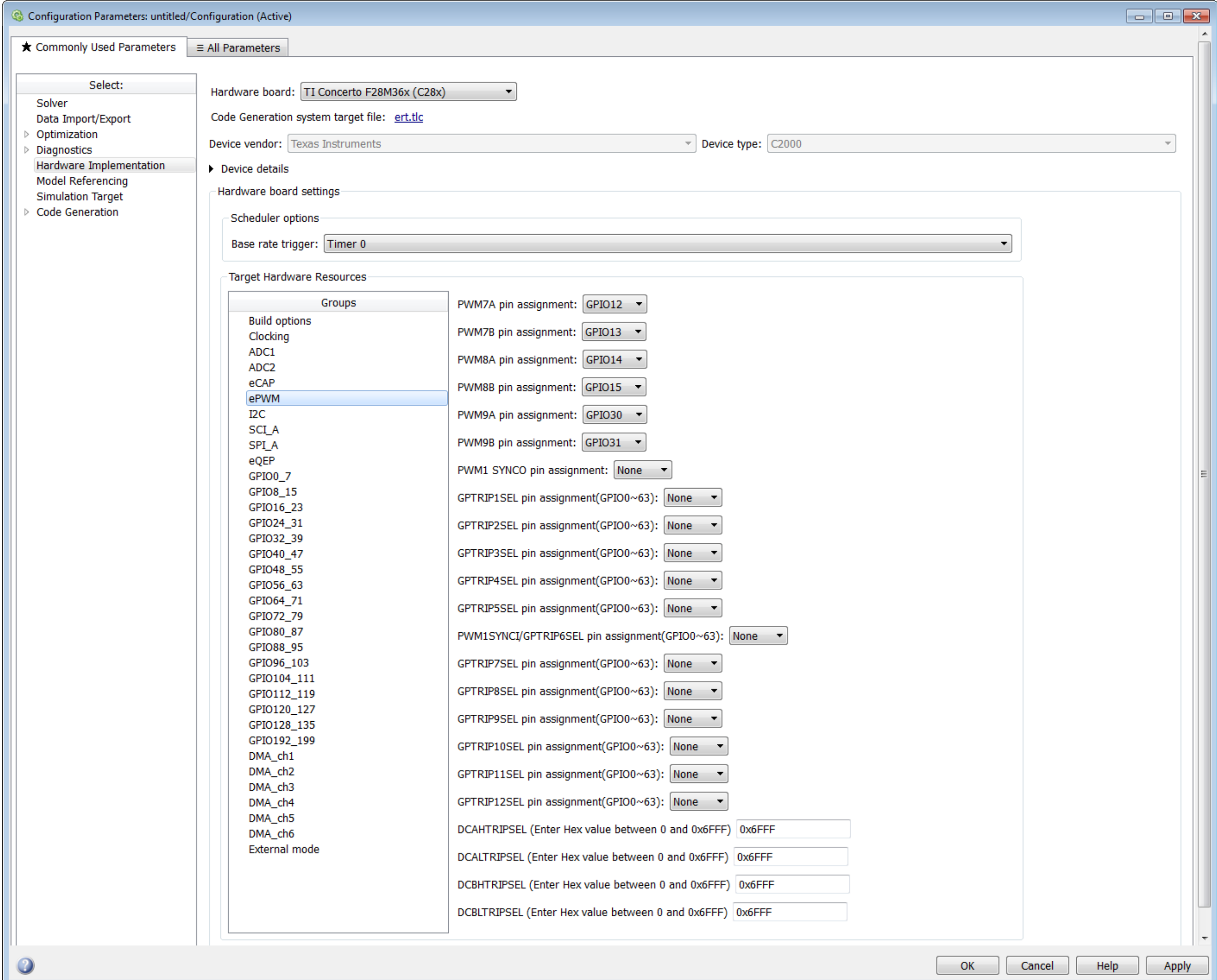
**ECAP5 APWM pin assignment**

Select GPIO pin for ECAP5 module when using in APWM operating mode.

**ECAP6 APWM pin assignment**

Select GPIO pin for ECAP6 module when using in APWM operating mode.

## C28x-ePWM




Assigns ePWM signals to GPIO pins.

### EPWM clock divider (EPWMCLKDIV)

This parameter appears only for F2807x and F2837x processors. This parameter allows you to select the ePWM clock divider.

### TZ1 pin assignment

Assigns the trip-zone input 1 (TZ1) to a GPIO pin. Choices are `None` (the default), `GPIO12`, and `GPIO15`.

### TZ2 pin assignment

Assigns the trip-zone input 2 (TZ2) to a GPIO pin. Choices are `None` (the default), `GPIO16`, and `GPIO28`.

### TZ3 pin assignment

Assigns the trip-zone input 3 (TZ3) to a GPIO pin. Choices are `None` (the default), `GPIO17`, and `GPIO29`.

### TZ4 pin assignment

Assigns the trip-zone input 4 (TZ4) to a GPIO pin. Choices are `None` (the default), `GPIO17`, and `GPIO28`.

### TZ5 pin assignment

Assigns the trip-zone input 5 (TZ5) to a GPIO pin. Choices are `None` (the default), `GPIO16`, and `GPIO28`.

### TZ6 pin assignment

Assigns the trip-zone input 6 (TZ6) to a GPIO pin. Choices are `None` (the default), `GPIO17`, and `GPIO29`.

**Note** The TZ# pin assignments are available only for TI F280x processors.

### SYNCI pin assignment

Assigns the ePWM external sync pulse input (SYNCI) to a GPIO pin. Choices are `None` (the default), `GPIO6`, and `GPIO32`.

### SYNCO pin assignment

**Note** SYNCI and SYNCO pin assignments are available for TI F28044, TI F280x, TI Delfino F2833x, TI Delfino F2834x, TI Piccolo F2802x, TI Piccolo F2803x, TI Piccolo F2806 processors.

Assigns the ePWM external sync pulse output (SYNCO) to a GPIO pin. Choices are `None` (the default), `GPIO6`, and `GPIO33`.

**PWM#A, PWM#B, PWM#C pin assignment**

The PWM # A, PWM # B, PWM # C pin assignment.

**GPTRIP#SEL pin assignment(GPIO0~63)**

Assigns the ePWM trip-zone input to a GPIO pin.

---

**Note** The GPTRIP#SEL pin assignment is available only for TI Concerto F28M35x/F28M36x processors.

---

**PWM1SYNCI/ GPTRIP6SEL pin assignment**

Assigns the ePWM sync pulse input (SYNCI) to a GPIO pin.

---

**Note** The PWM1SYNCI/GPTRIP#SEL pin assignments are available only for TI Concerto F28M35x/F28M36x processors.

---

**DCAHTRIPSEL (Enter Hex value between 0 and 0x6FFF) / DCBHTRIPSEL (Enter Hex value between 0 and 0x6FFF)**

Assigns the Digital Compare A High Trip Input to a GPIO pin.

---

**Note** DCAHTRIPSel pin assignment is available only for TI Concerto F28M35x/F28M36x processors.

---

**DCALTRIPSEL (Enter Hex value between 0 and 0x6FFF) / DCBLTRIPSEL (Enter Hex value between 0 and 0x6FFF)**

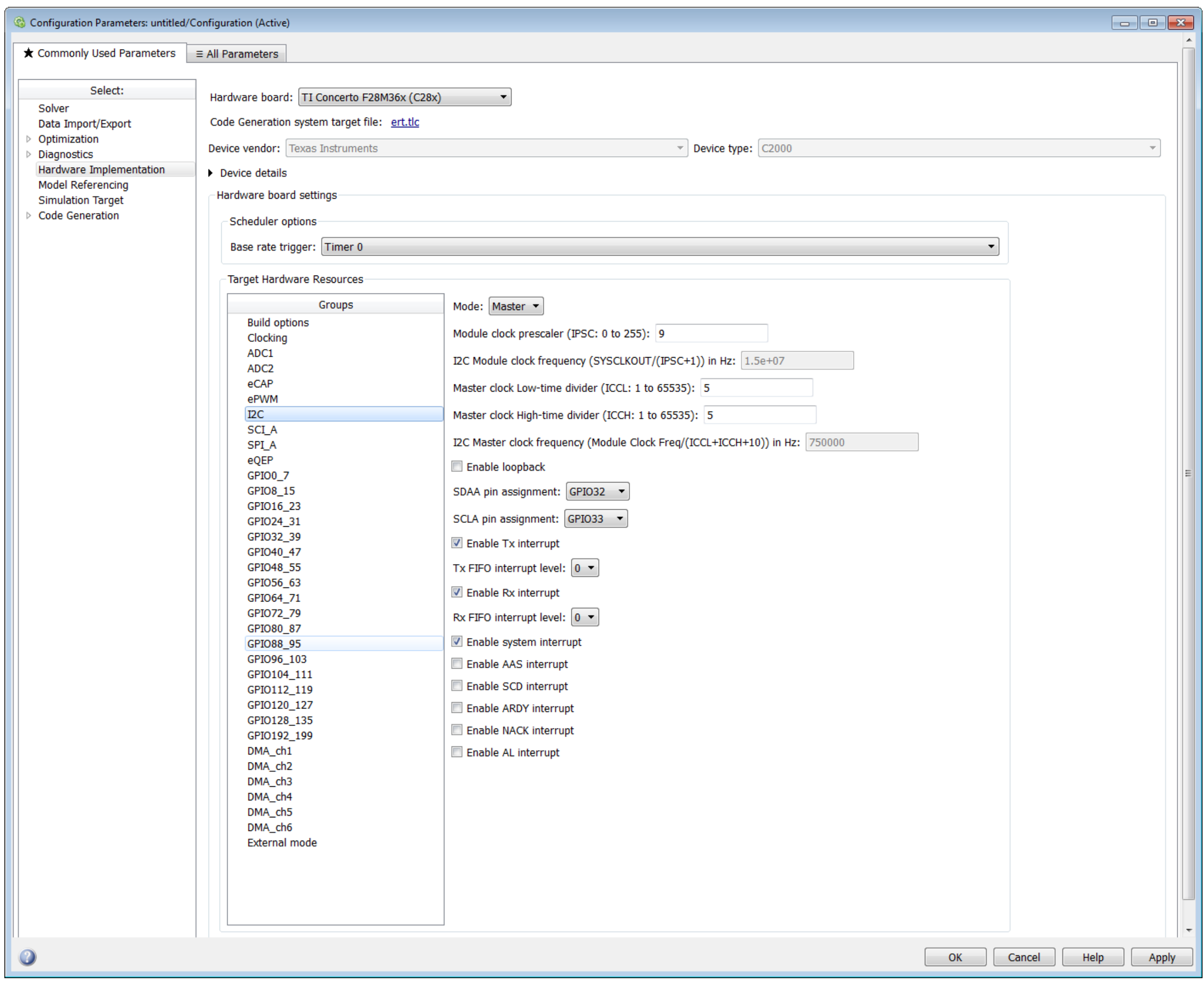
Assigns the Digital Compare A High Trip Input to a GPIO pin.

---

**Note** The DCALTRIPSEL pin assignment is available only for TI Concerto F28M35x/F28M36x processors.

---

# C28x-I2C




Report or set Inter-Integrated Circuit parameters. For more information, consult the *TMS320x280x/ TMS320F28M35x/ TMS320F28M36x Inter-Integrated Circuit Module Reference Guide*, Literature Number: SPRU721A/ SPRUH22F/ SPRUHE8B available on the Texas Instruments Web site.

**Mode**

Configure the I2C module as `Master` or `Slave`.

If a module is an I2C master, it:

Initiates communication with slave nodes by sending the slave address and requesting data transfer to or from the slave.

Outputs the **Master clock frequency** on the serial clock line (SCL) line.

If a module is an I2C slave, it:

- Synchronizes itself with the serial clock line (SCL) line.
- Responds to communication requests from the master.

When **Mode** is `Slave`, you can configure the **Addressing format**, **Address register**, and **Bit count** parameters.

The **Mode** parameter corresponds to bit 10 (MST) of the I2C Mode Register (I2CMDR).

**Addressing format**

If **Mode** is `Slave`, determine the addressing format of the I2C master, and set the I2C module to the same mode:

- `7-Bit Addressing`, the normal address mode.
- `10-Bit Addressing`, the expanded address mode.
- `Free Data Format`, a mode that does not use addresses. (If you **Enable loopback**, the `Free data format` is not supported.)

The **Addressing format** parameter corresponds to bit 3 (FDF) and bit 8 (XA) of the I2C Mode Register (I2CMDR).

**Own address register**

If **Mode** is `Slave`, enter the 7-bit (0–127) or 10-bit (0–1023) address this I2C module uses while it is a slave.

This parameter corresponds to bits 9–0 (OAR) of the I2C Own Address Register (I2COAR).

**Bit count**

If **Mode** is `Slave`, set the number of bits in each data *byte* the I2C module transmits and receives. This value must match that of the I2C master.

This parameter corresponds to bits 2–0 (BC) of the I2C Mode Register (I2CMDR).

**Module clock prescaler (IPSC: 0 to 255):**

If **Mode** is `Master`, configure the module clock frequency by entering a value 0–255.

*Module clock frequency* = *I2C input clock frequency* / (*Module clock prescaler* + 1)

The I2C specifications require a module clock frequency between 7 MHz and 12 MHz.

The *I2C input clock frequency* depends on the DSP input clock frequency and the value of the PLL Control Register divider (PLLCR). For more information on setting the PLLCR, consult the documentation for your specific Digital Signal Controller.

This **Module clock prescaler (IPSC: 0 to 255):** corresponds to bits 7–0 (IPSC) of the I2C Prescaler Register (I2CPSC).

**I2C Module clock frequency (SYSCLKOUT / (IPSC+1)) in Hz:**

This field displays the frequency the I2C module uses internally. To set this value, change the **Module clock prescaler**.

For more information about this value, consult the “Formula for the Master Clock Period” section in the *TMS320x280x Inter-Integrated Circuit Module Reference Guide*, Literature Number: SPRU721, on the Texas Instruments Web site.

**I2C Master clock frequency (Module Clock Freq/(ICCL+ICCH+10)) in Hz:**

This field displays the master clock frequency.

For more information about this value, consult the “Clock Generation” section in the *TMS320x280x/ TMS320F28M35x/ TMS320F28M36x Inter-Integrated Circuit Module Reference Guide*, Literature Number: SPRU721/ SPRUH22F/ SPRUHE8B, available on the Texas Instruments Web site.

**Master clock Low-time divider (ICCL: 1 to 65535):**

When **Mode** is `Master`, this divider determines the duration of the low state of the SCL line on the I2C-bus.

The low-time duration of the master clock = Tmod x (ICCL + d).

For more information, consult the “Formula for the Master Clock Period” section in the *TMS320x280x/ TMS320F28M35x/ TMS320F28M36x Inter-Integrated Circuit*

*Module Reference Guide*, Literature Number: SPRU721A/ SPRUH22F/ SPRUHE8B, available on the Texas Instruments Web site.

This parameter corresponds to bits 15–0 (ICCL) of the Clock Low-Time Divider Register (I2CCLKL).

**Master clock High-time divider (ICCH: 1 to 65535):**

When **Mode** is `Master`, this divider determines the duration of the high state on the serial clock pin (SCL) of the I2C-bus.

The high-time duration of the master clock = Tmod x (ICCL + d).

For more information about this value, consult the “Formula for the Master Clock Period” section in the *TMS320x280x/ TMS320F28M35x/ TMS320F28M36x Inter-Integrated Circuit Module Reference Guide*, Literature Number: SPRU721A, SPRUH22f, SPRUHE8B, available on the Texas Instruments Web site.

This parameter corresponds to bits 15–0 (ICCH) of the Clock High-Time Divider Register (I2CCLKH).

**Enable loopback**

When **Mode** is `Master`, enable or disable digital loopback mode. In digital loopback mode, I2CDXR transmits data over an internal path to I2CDRR, which receives the data after a configurable delay.

The delay, measured in DSP cycles, equals (I2C input clock frequency/module clock frequency) x 8.

While **Enable loopback** is enabled, free data format addressing is not supported.

This parameter corresponds to bit 6 (DLB) of the I2C Mode Register (I2CMDR).

**SDAA pin assignment**

Select a GPIO pin as I2C data bidirectional port.

This parameter is available only for TI Concerto F28M36x (C28x) processors.

**SCLA pin assignment**

Select the GPIO pin as I2C clock bidirectional port.

This parameter is available only for TI Concerto F28M36x (C28x) processors.

**Enable Tx interrupt**

This parameter corresponds to bit 5 (TXFFIENA) of the I2C Transmit FIFO Register (I2CFFTX).

**Tx FIFO interrupt level**

This parameter corresponds to bits 4–0 (TXFFIL4-0) of the I2C Transmit FIFO Register (I2CFFTX).

**Enable Rx interrupt**

This parameter corresponds to bit 5 (RXFFIENA) of the I2C Receive FIFO Register (I2CFFRX).

**Rx FIFO interrupt level**

This parameter corresponds to bit 4–0 (RXFFIL4-0) of the I2C Receive FIFO Register (I2CFFRX).

**Enable system interrupt**

Select this parameter to display and individually configure the following five Basic I2C Interrupt Request parameters in the Interrupt Enable Register (I2CIER):

- Enable AAS interrupt
- Enable SCD interrupt
- Enable ARDY interrupt
- Enable NACK interrupt
- Enable AL interrupt

**Enable AAS interrupt**

Enable the addressed-as-slave interrupt.

When enabled, the I2C module generates an interrupt (AAS bit = 1) upon receiving one of the following:

- Its **Own address register**
- A general call (all zeros)
- A data byte is in free data format

When enabled, the I2C module clears the interrupt (AAS = 0) upon receiving one of the following:

- Multiple START conditions (7-bit addressing mode only)

- A slave address that is different from **Own address register** (10-bit addressing mode only)
- A NACK or a STOP condition

This parameter corresponds to bit 6 (AAS) of the Interrupt Enable Register (I2CIER).

**Enable SCD interrupt**

Enable stop condition detected interrupt.

When enabled, the I2C module generates an interrupt (SCD bit = 1) when the CPU detects a stop condition on the I2C bus.

When enabled, the I2C module clears the interrupt (SCD = 0) upon one of the following events:

- The CPU reads the I2CISRC while it indicates a stop condition
- A reset of the I2C module
- Someone manually clears the interrupt

This parameter corresponds to bit 5 (SCD) of the Interrupt Enable Register (I2CIER).

**Enable ARDY interrupt**

Enable register-access-ready interrupt enable bit.

When enabled, the I2C module generates an interrupt (ARDY bit = 1) when the previous address, data, and command values in the I2C module registers have been used and new values can be written to the I2C module registers.

This parameter corresponds to bit 2 (ARDY) of the Interrupt Enable Register (I2CIER).

**Enable NACK interrupt**

Enable no acknowledgment interrupt enable bit.

When enabled, the I2C module generates an interrupt (NACK bit = 1) when the module is a transmitter in master or slave mode and it receives a NACK condition.

This parameter corresponds to bit 1 (NACK) of the Interrupt Enable Register (I2CIER).

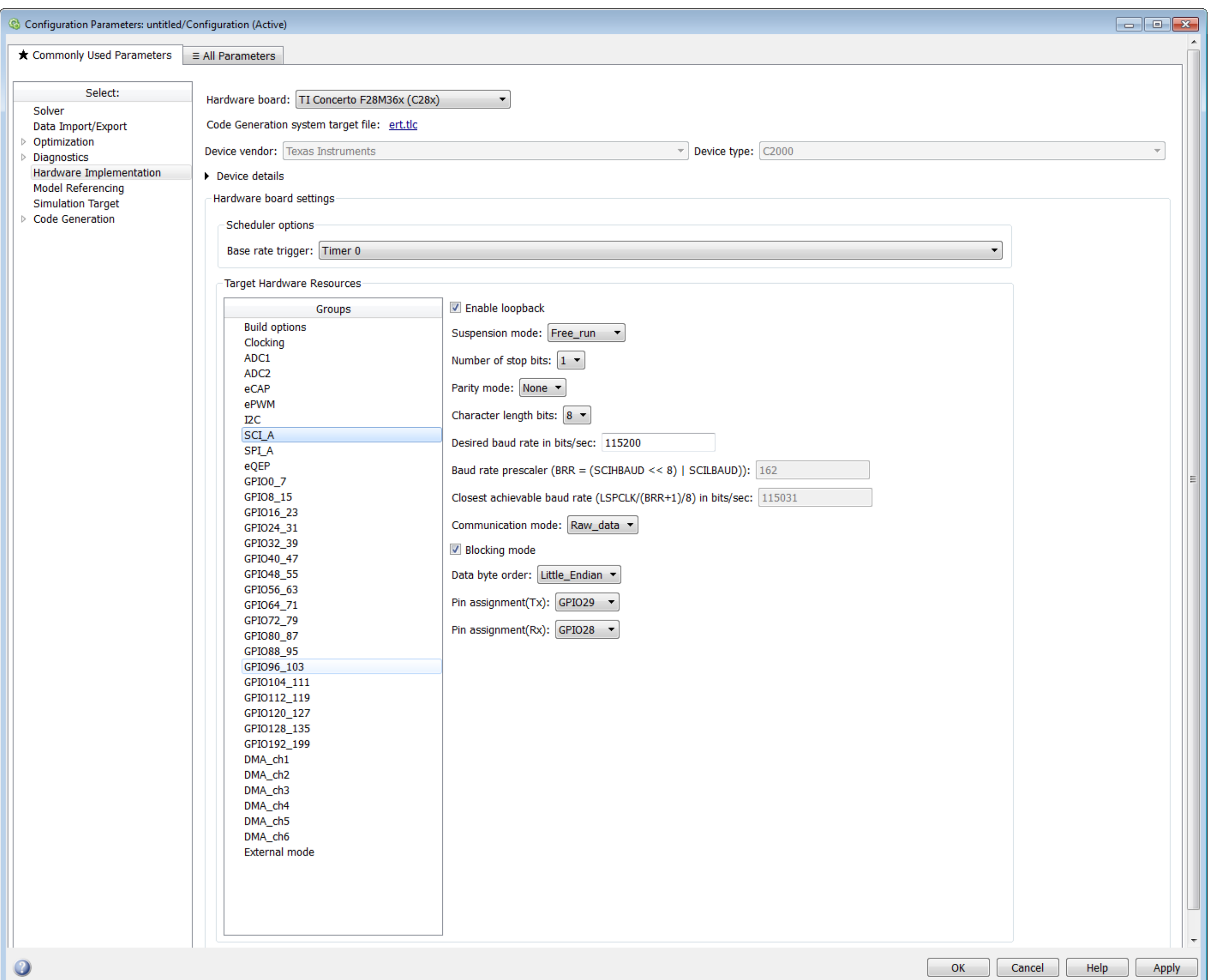
**Enable AL interrupt**

Enable arbitration-lost interrupt.

When enabled, the I2C module generates an interrupt (AL bit = 1) when the I2C module is operating as a master transmitter and looses an arbitration contest with another master transmitter.

This parameter corresponds to bit 0 (AL) of the Interrupt Enable Register (I2CIER).

## C28x-SCI_A, C28x-SCI_B, C28x-SCI_C



The serial communications interface parameters you can set for module A. These parameters are:

**Enable loopback**

Select this parameter to enable the loopback function for self-test and diagnostic purposes only. When this function is enabled, a C28x DSP Tx pin is internally connected to its Rx pin and can transmit data from its output port to its input port to check the integrity of the transmission.

**Baud rate**

Baud rate for transmitting and receiving data. Select from `115200` (the default), `57600`, `38400`, `19200`, `9600`, `4800`, `2400`, `1200`, `300`, and `110`.

**Suspension mode**

Type of suspension to use when debugging your program with Code Composer Studio. When your program encounters a breakpoint, the suspension mode determines whether to perform the program instruction. Available options are `Hard_abort`, `Soft_abort`, and `Free_run`. `Hard_abort` stops the program immediately. `Soft_abort` stops when the current receive/transmit sequence is complete. `Free_run` continues running regardless of the breakpoint.

**Number of stop bits**

Select whether to use `1` or `2` stop bits.

**Parity mode**

Type of parity to use. Available selections are `None`, `Odd` parity, or `Even` parity. `None` disables parity. `Odd` sets the parity bit to one if you have an odd number of ones in your bytes, such as 00110010. `Even` sets the parity bit to one if you have an even number of ones in your bytes, such as 00110011.

**Character length bits**

Length in bits of each transmitted or received character, set to 8 bits.

**Desired baud rate in bits/sec**

The desired baud rate specified by the user.

**Baud rate prescaler (BRR = (SCIHBAUD << 8) | SCILBAUD))**

The baud rate prescaler.

**Closest achievable baud rate (LSPCLK/(BRR+1)/8) in bits/sec**

The closest achievable baud rate calculated based on LSPCLK and BRR.

#### Communication mode

Select `Raw_data` or `Protocol` mode. Raw data is unformatted and sent whenever the transmitting side is ready to send, whether the receiving side is ready or not. Without a wait state, deadlock conditions do not occur. Data transmission is asynchronous. With this mode, it is possible the receiving side could miss data, but if the data is noncritical, using raw data mode can avoid blocking processes.

When you select protocol mode, some handshaking between host and processor occurs. The transmitting side sends $SND to indicate it is ready to transmit. The receiving side sends back $RDY to indicate it is ready to receive. The transmitting side then sends data and, when the transmission is completed, it sends a checksum.

Advantages to using protocol mode include:

- Avoids deadlock
- Determines whether data is received without errors (checksum)
- Determines whether data is received by processor
- Determines time consistency; each side waits for its turn to send or receive

---

**Note** Deadlocks can occur if one SCI Transmit block tries to communicate with more than one SCI Receive block on different COM ports when both are blocking (using protocol mode). Deadlocks cannot occur on the same COM port.

---

#### Blocking mode

If this option is set to True, system waits until data is available to read (when data length is reached). If this option is set to False, system checks FIFO periodically (in polling mode) for data to read. If data is present, the block reads and outputs the contents. When data is not present, the block outputs the last value and continues.

#### Data byte order

Select `Little Endian` or `Big Endian`, to match the endianness of the data being moved.

#### Pin assignment (Tx)

Assigns the SCI transmit pin to use with the SCI module.

#### Pin assignment (Rx)

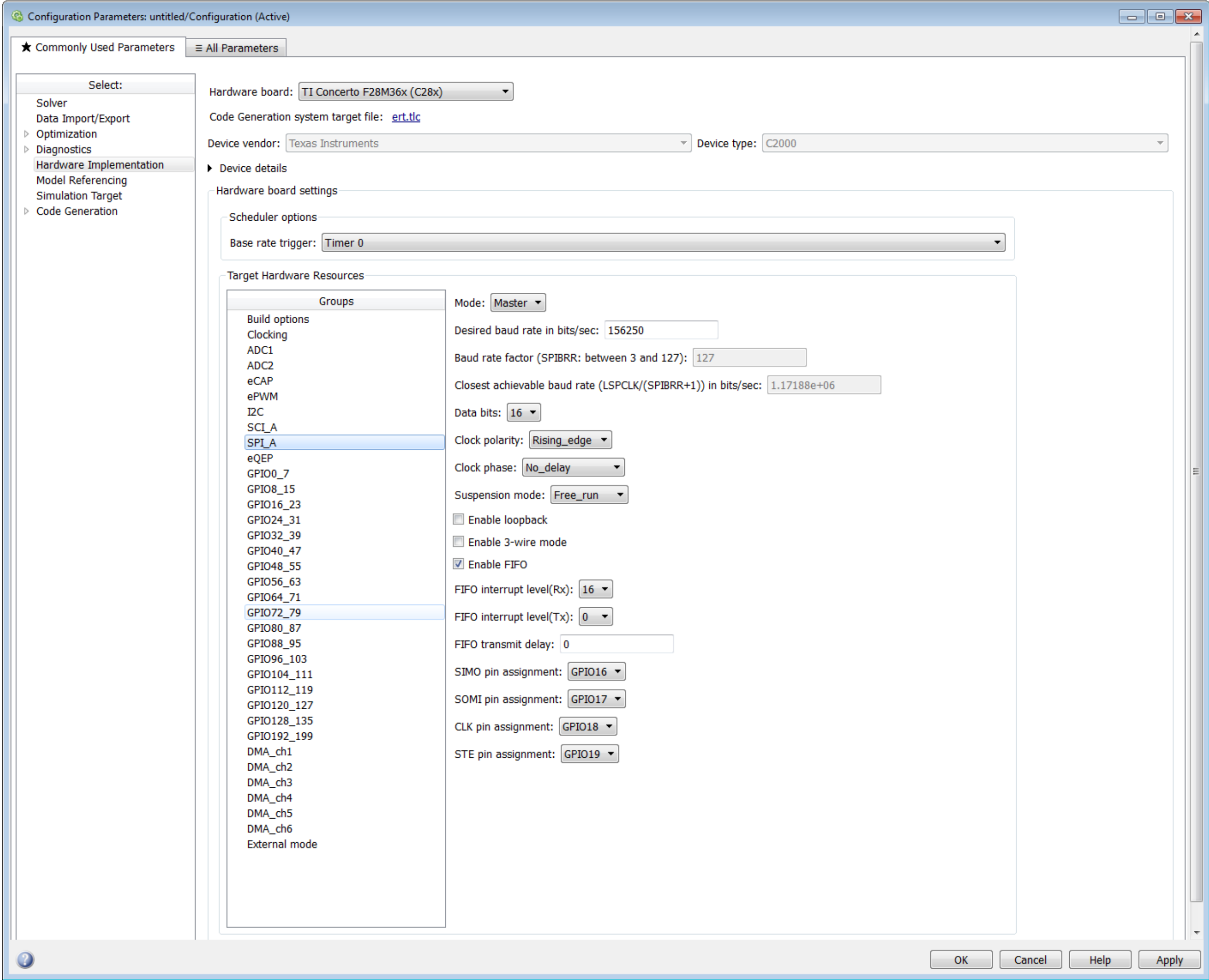
Assigns the SCI receive pin to use with the SCI module.

---

**Note** The SCI_B and SCI_C are available only for TI F280x processors.

---

## C28x-SPI_A, C28x-SPI_B, C28x-SPI_C, C28x-SPI_D




The serial peripheral interface parameters you can set for the A module. These parameters are:

**Mode**

Set to `Master` or `Slave`.

**Desired baud rate in bits/sec**

The desired baud rate specified by the user.

**Baud rate factor (SPIBRR: between 3 and 127)**

To set the **Baud rate factor**, search for “Baud Rate Determination” and “SPI Baud Rate Register (SPIBRR) Bit Descriptions” in *TMS320x28xx, 28xxx DSP Serial Peripheral Interface (SPI) Reference Guide*, Literature Number: SPRU059, available on the Texas Instruments Web Site.

**Closest achievable baud rate (LSPCLK/(SPIBRR+1)) in bits/sec**

The closest achievable baud rate calculated based on LSPCLK and SPIBRR.

**Data bits**

Length in bits from `1` to `16` of each transmitted or received character. For example, if you select `8`, the maximum data that can be transmitted using SPI is $2^{8-1}$. If you send data greater than this value, the buffer overflows.

**Clock polarity**

Select `Rising_edge` or `Falling_edge`.

**Clock phase**

Select `No_delay` or `Delay_half_cycle`.

**Suspension mode**

Type of suspension to use when debugging your program with Code Composer Studio. When your program encounters a breakpoint, the selected suspension mode determines whether to perform the program instruction. Available options are `Hard_abort`, `Soft_abort`, and `Free_run`. `Hard_abort` stops the program immediately. `Soft_abort` stops when the current receive or transmit sequence is complete. `Free_run` continues running regardless of the breakpoint.

**Enable loopback**

Select this option to enable the loopback function for self-test and diagnostic purposes only. When this function is enabled, the Tx pin on a C28x DSP is internally connected to its Rx pin and can transmit data from its output port to its input port to check the integrity of the transmission.

**Enable 3-wire mode**

Enable SPI communication over three pins instead of the normal four pins.

**Enable FIFO**

Set true or false.

**FIFO interrupt level (Rx)**

Set level for receive FIFO interrupt.

**FIFO interrupt level (Tx)**

Set level for transmit FIFO interrupt.

**FIFO transmit delay**

Enter FIFO transmit delay (in processor clock cycles) to pause between data transmissions. Enter an integer.

**CLK pin assignment**

Assigns the SPI something (CLK) to a GPIO pin. Choices are `None` (default), `GPIO14`, or `GPIO26`.

CLK pin assignment is not available for TI Concerto F28M35x/F28M36x processors.

**SOMI pin assignment**

Assigns the SPI value (SOMI) to a GPIO pin. Choices are `None` (default), `GPIO13`, or `GPIO25`.

**STE pin assignment**

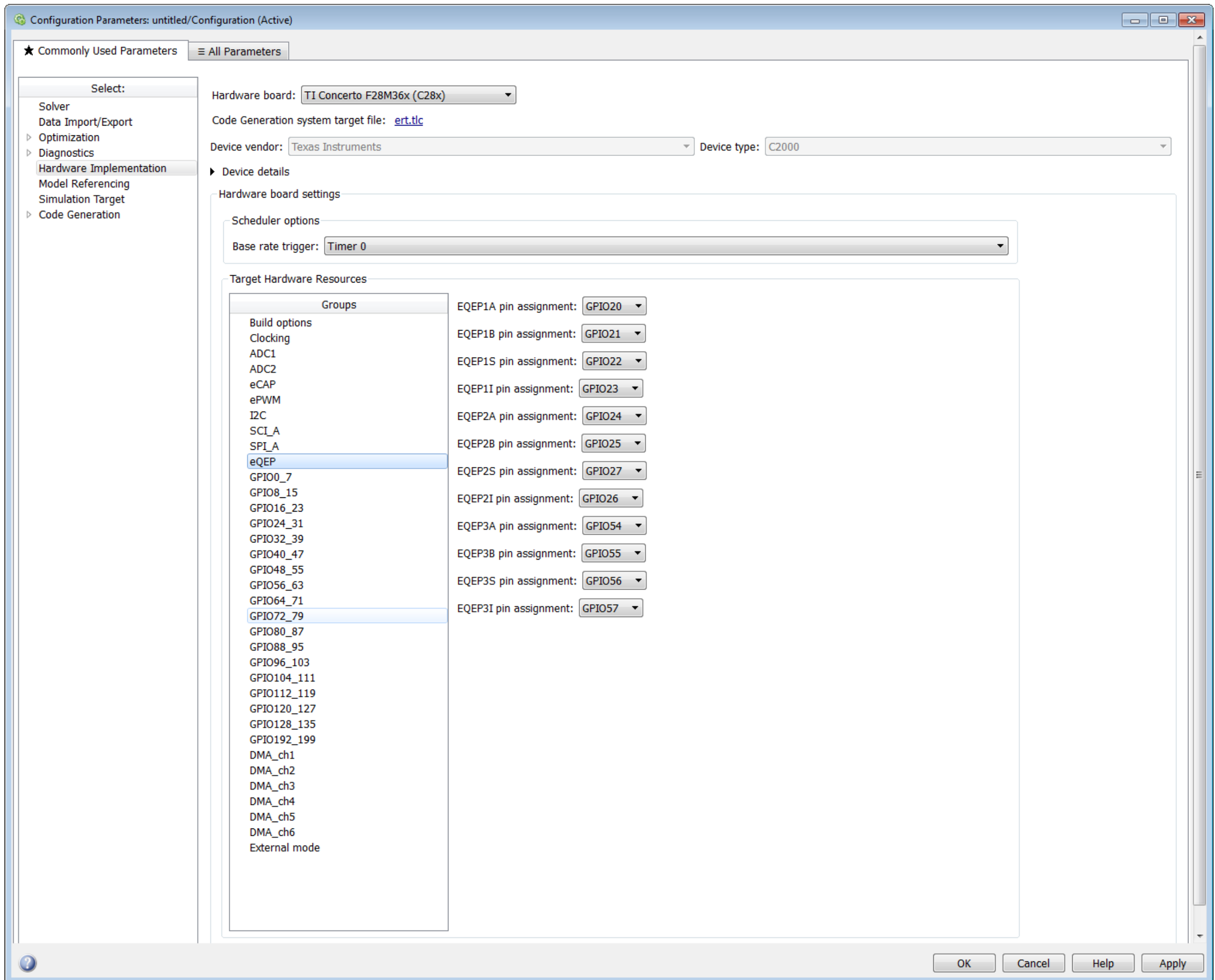
Assigns the SPI value (STE) to a GPIO pin. Choices are `None` (default), `GPIO15`, or `GPIO27`.

STE pin assignment is not available for TI Concerto F28M35x/ F28M36x processors.

**SIMO pin assignment**

Assigns the SPI something (SIMO) to a GPIO pin. Choices are `None` (default), `GPIO12`, or `GPIO24`.

# C28x-eQEP




Assigns eQEP pins to GPIO pins.

### EQEP1A pin assignment

Select an option from the list—`None`, `GPIO20`, `GPIO50`, `GPIO64`, or `GPIO106`.

**EQEP1B pin assignment**

Select an option from the list—`None`, `GPI021`, `GPIO51`, `GPIO65`, or `GPIO107`.

**EQEP1S pin assignment**

Select an option from the list—`None`, `GPIO22`, `GPIO52`, `GPIO66`, or `GPIO108`.

**EQEP1I pin assignment**

Select an option from the list—`None`, `GPIO23`, `GPIO53`, `GPIO67`, or `GPIO109`.

**EQEP2A pin assignment**

Select an option from the list—`None`, `GPIO24`, `GPIO54`, `GPIO66`, or `GPIO110`. The pin numbers shown in the list vary based on the processor selected.

**EQEP2B pin assignment**

Select an option from the list—`None`, `GPIO25`, `GPIO55`, `GPIO67`, or `GPIO111`. The pin numbers shown in the list vary based on the processors selected.

**EQEP2S pin assignment**

Select an option from the list—`None`, `GPIO27`, `GPIO31`, `GPIO57`, `GPIO67`, or `GPIO113`.

**EQEP2I pin assignment**

Select an option from the list—`None`, `GPIO26`, `GPIO30`, `GPIO56`, `GPIO64`, or `GPIO112`.

**EQEP3A pin assignment**

Select an option from the list—`None`, `GPIO54`, or `GPIO112`. This parameter is available only for TI Concerto F28M36x (C28x) processors.

**EQEP3B pin assignment**

Select an option from the list—`None`, `GPIO55`, or `GPIO113`. This parameter is available only for TI Concerto F28M36x (C28x) processors.

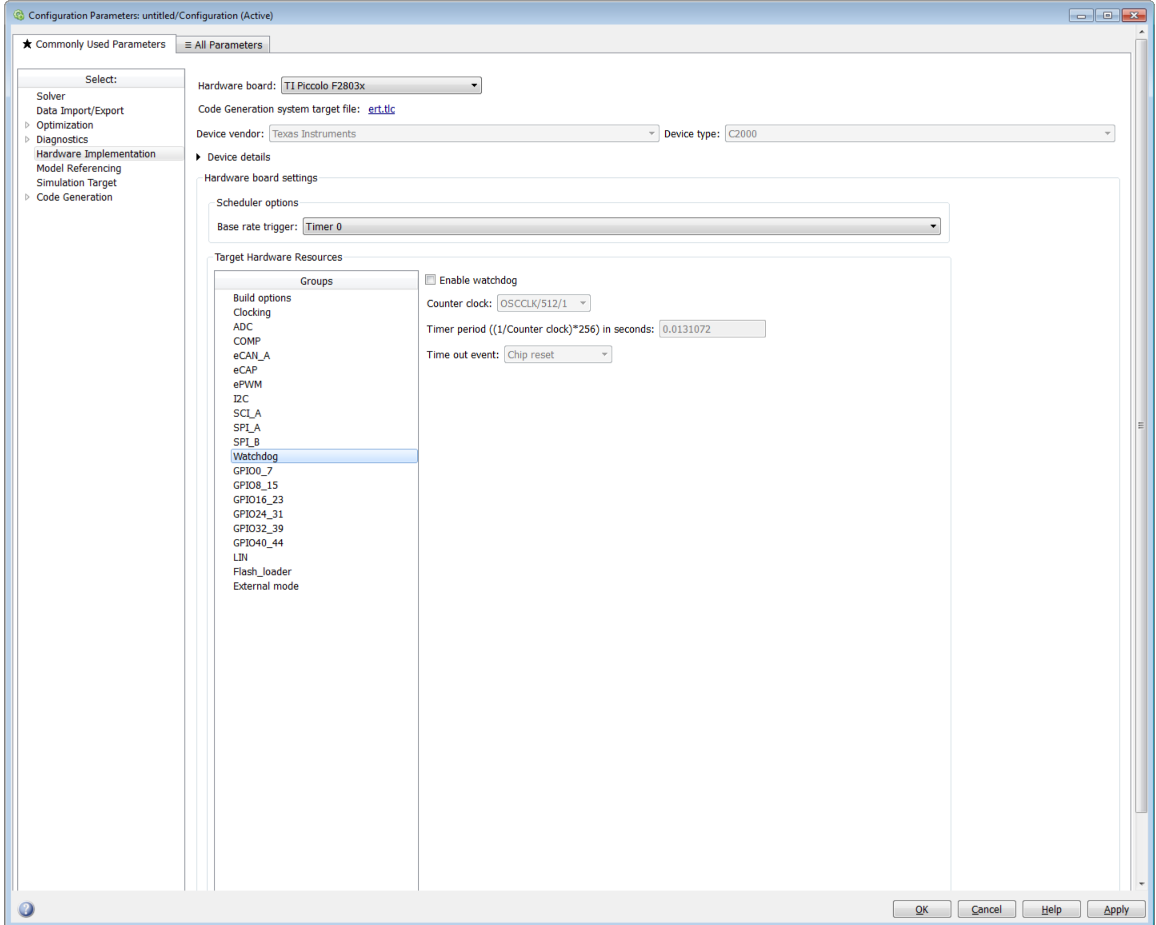
**EQEP3S pin assignment**

Select an option from the list—`None`, `GPIO56`, or `GPI110`.

**EQEP3I pin assignment**

Select an option from the list—`None`, `GPIO57`, or `GPIO111`.

## C28x-Watchdog




When enabled, if the software fails to reset the watchdog counter within a specified interval, the watchdog resets the processor or generates an interrupt. This feature enables the processor to recover from some fault conditions.

For more information, locate the *Data Manual* or *System Control and Interrupts Reference Guide* for your processor on the Texas Instruments Web site.

**Enable watchdog**

Enable the watchdog timer module.

This parameter corresponds to bit 6 (WDDIS) of the Watchdog Control Register (WDCR) and bit 0 (WDOVERRIDE) of the System Control and Status Register (SCSR).

**Counter clock**

Set the watchdog timer period relative to OSCCLK/512.

This parameter corresponds to bits 2–0 (WDPS) of the Watchdog Control Register (WDCR).

**Timer period ((1/Counter clock)*256) in seconds**

This field displays the timer period in seconds. This value automatically updates when you change the **Counter clock** parameter.

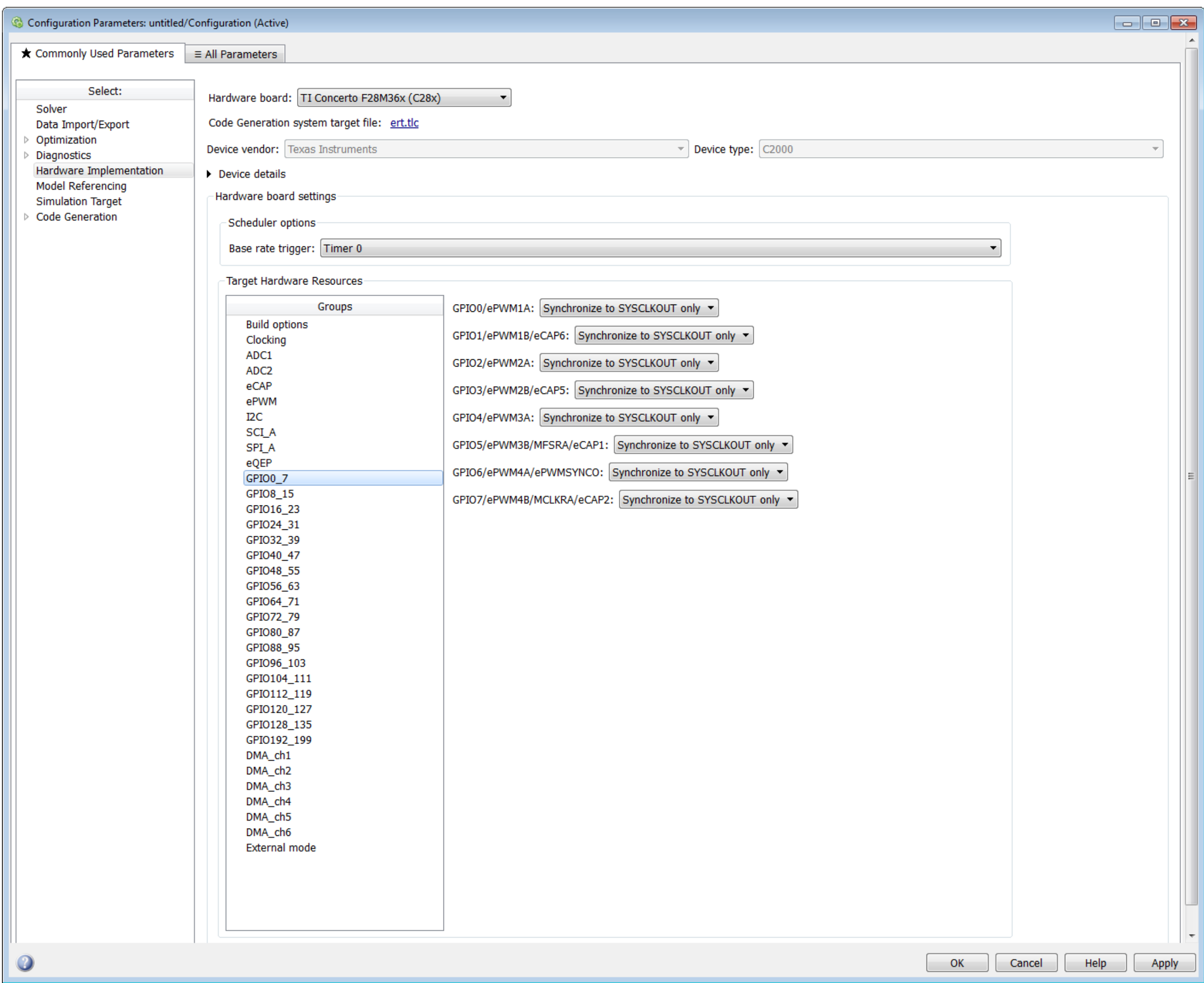
**Time out event**

Configure the watchdog to reset the processor or generate an interrupt when the software fails to reset the watchdog counter:

- Select `Chip reset` to generate a signal that resets the processor (WDRST signal) and disable the watchdog interrupt signal (WDINT signal).
- Select `Raise WD Interrupt` to generate a watchdog interrupt signal (WDINT signal) and disable the reset processor signal (WDRST signal). This signal can be used to wake the device from an IDLE or STANDBY low-power mode.

This parameter corresponds to bit 1 (WDENINT) of the System Control and Status Register (SCSR).

## C28x-GPIO



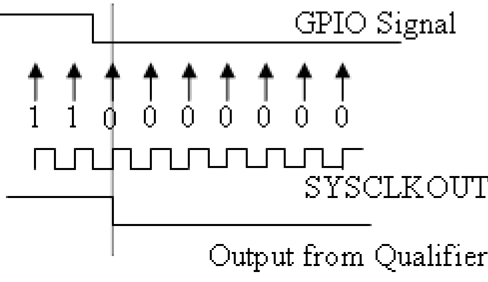

**GPIO**Use the GPIO pins for digital input or output by connecting to one of the three peripheral I/O ports.

The range of GPIO pins for different processors is given below:

| Processors | GPIO Pin Values |
|---|---|
| C281x | GPIOA, GPIOB, GPIOD, GPIOE, GPIOF, and GPIOG |
| F2803x | GPIO0–7, GPIO8–15, GPIO16–23, GPIO24–31, GPIO32–39, GPIO40–44 |
| F2806x | GPIO0–7, GPIO8–15, GPIO16–23, GPIO24–31, GPIO32–39, GPIO40–44, GPIO50–55, GPIO56–58 |
| F2823x, F2833x, and C2834x | GPIO0–7, GPIO8–15, GPIO16–23, GPIO24–31, GPIO32–39, GPIO40–47, GPIO48–55, GPIO56–63 |
| C2801x, F2802x, F28044, F280x | GPIO0–7, GPIO8–15, GPIO16–23, GPIO24–31, GPIO32–34 |
| F28M35x (C28x) | GPIO0–7, GPIO8–15, GPIO16–23, GPIO24–31, GPIO32–39, GPIO40–47, GPIO48_55, GPIO56–63, GPIO68–71, GPIO128–135 |
| F28M36x (C28x) | GPIO0–7, GPIO8–15, GPIO16–23, GPIO24–31, GPIO40–47, GPIO48–55, GPIO56–63, GPIO64–71, GPIO72–79, GPIO80–87, GPIO88–95, GPIO96–103, GPIO104–111, GPIO112–119, GPIO120–127, GPIO128–135, GPIO192–199. |

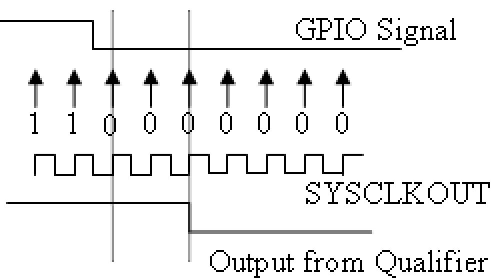
Each pin selected for input offers four signal qualification types:

- `Sync to SYSCLKOUT only` — This setting is the default for all pins at reset. Using this qualification type, the input signal is synchronized to the system clock SYSCLKOUT. The following figure shows the input signal measured on each tick of the system clock, and the resulting output from the qualifier.

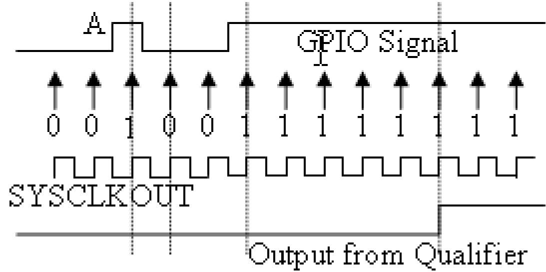



- `Qualification using 3 samples` — This setting requires three consecutive cycles of the same value for the output value to change. The following figure shows that, in the third cycle, the GPIO value changes to `0`, but the qualifier output is still `1`

because it waits for three consecutive cycles of the same GPIO value. The next three cycles all have a value of `0`, and the output from the qualifier changes to `0` immediately after the third consecutive value is received.



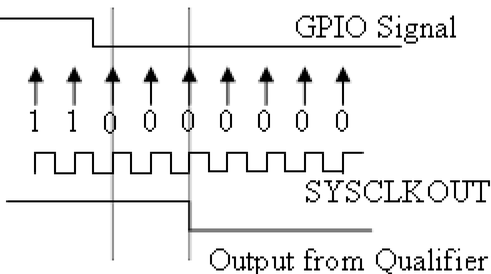

- `Qualification using 6 samples` — This setting requires six consecutive cycles of the same GPIO input value for the output from the qualifier to change. In the following figure, the glitch **A** does not alter the output signal. When the glitch occurs, the counting begins, but the next measurement is low again, so the count is ignored. The output signal does not change until six consecutive samples of the high signal are measured.



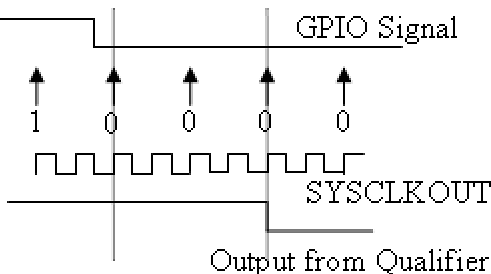

**Qualification sampling period prescaler**

Visible only when a setting for **Qualification type for GPIO [pin#]** is selected. The qualification sampling period prescaler, with possible values of `0` to `255`, calculates the frequency of the qualification samples or the number of system clock ticks per sample. The formula for calculating the qualification sampling frequency is SYSCLKOUT/(2 * Prescaler), except for zero. When **Qualification sampling period prescaler=**`0`, a sample is taken every SYSCLKOUT clock tick. For example, a prescale setting of `0` means that a sample is taken on each SYSCLKOUT tick.

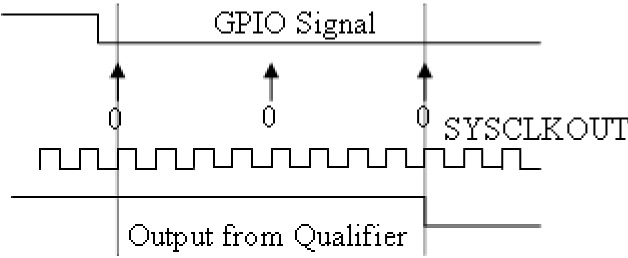
The following figure shows the SYSCLKOUT ticks, a sample taken every clock tick, and the **Qualification type** set to `Qualification using 3 samples`. In this case, the **Qualification sampling period prescaler=**`0`:




In the next figure **Qualification sampling period prescaler=**`1`. A sample is taken every two clock ticks, and the **Qualification type** is set to `Qualification using 3 samples`. The output signal changes much later than if **Qualification sampling period prescaler=**`0`.




In the following figure, **Qualification sampling period prescaler=**`2`. Thus , a sample is taken every four clock ticks, and the **Qualification type** is set to `Qualification using 3 samples`.




- `Asynchronous`

  Using this qualification type, the signal is synchronized to an asynchronous event initiated by software (CPU) via control register bits.

**Qualification sampling period**

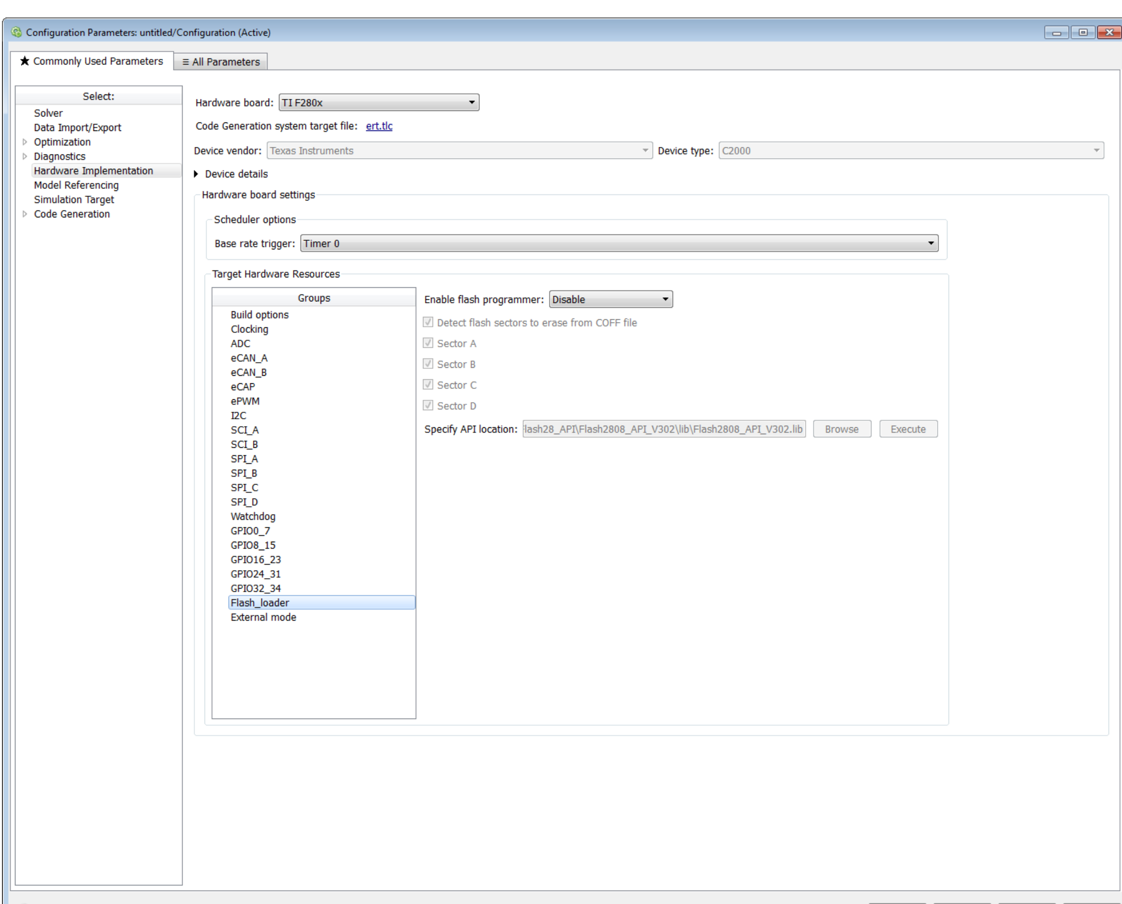
Enter the qualification sampling period.

**GPIOA, GPIOB, GPIOD, GPIOE input qualification sampling period**

**GPIO# Pull Up Disabled**

Select this check box to disable the GPIO pull up register. This option is available only for TI Concerto F28M35x/F28M36x processors.

# C28x-Flash_loader




You can use Flash_loader to:

- Automatically program generated code to flash memory on the target when you build the code.
- Manually erase, program, or verify specific flash memory sectors.

To use this feature, download and install the TI Flash API plugin from the TI Web site.

For more information, consult the *_API_Readme.pdf file included in the *TI Flash API* downloadable zip file.

**Enable Flash Programmer**

Enable the flash programmer by selecting a task for it to perform when you click **Execute** or build the software.

To program the flash memory when you build the software, select `Erase, Program, Verify`.

**Detect Flash sectors to erase from COFF file**

When enabled, the flash programmer erases all of the flash sectors defined by the COFF file.

**Sector A, Sector B, Sector C...**

When **Detect Flash sectors to erase from COFF file** is disabled, you can select the specific sector to erase.

**Specify API location**

Specify the folder path of the TI flash API executable you downloaded and installed on your computer.

Use **Browse** to locate the file or enter the path in the text box.

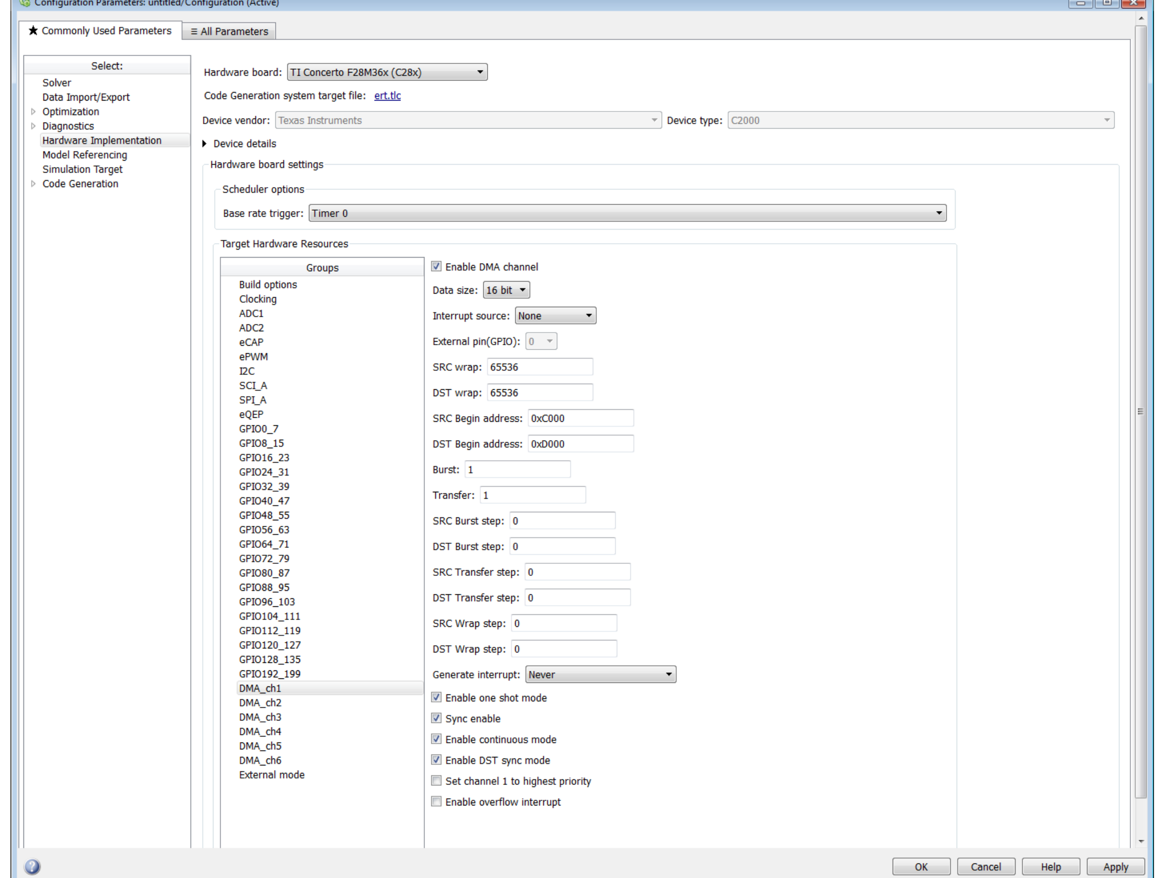
For example,

```
C:\TI\controlSUITE\libs\utilities\flash_api\2806x\v100\lib\2806x_BootROM_API_TABLE_Symbols_fpu32.lib
```

**Execute**

Click this button to initiate the task selected in **Enable Flash Programmer**.

# C28x-DMA_ch[#]




The Direct Memory Access module transfers data directly between peripherals and memory using a dedicated bus, increasing overall system performance.

You can individually enable and configure each DMA channel.

The DMA module services are event driven. Using the Interrupt source and **External pin (GPIO)** parameters, you can configure a wide range of peripheral interrupt event triggers.

For more information, consult the *TMS320x2833x, 2823x/ TMS320F28M35x/ TMS320F28M36x Direct Memory Access (DMA) Module Reference Guide*, Literature Number: SPRUFB8A/ SPRUH22F/ SPRUHE8B.

Also consult the *Increasing Data Throughput using the TMS320F2833x DSC DMA* training presentation (requires login), both available from the TI Web site.

**Enable DMA channel**

Enable this parameter to edit the configuration of a specific DMA channel.

This parameter does not have a corresponding bit or register.

**Data size**

Select the size of the data bit transfer: `16 bit` or `32 bit`.

The DMA read/write data buses are 32 bits wide. 32-bit transfers have twice the data throughput of a 16-bit transfer.

When providing DMA service to McBSP, set **Data size** to `16 bit`.

The following parameters are based on a 16-bit word size. If you set **Data size** to `32 bit`, double the value of the following parameters:

- Size: Burst
- Source: Burst step
- Source: Transfer step
- Source: Wrap step
- Destination: Burst step
- Destination: Transfer step
- Destination: Wrap step

**Data size** corresponds to bit 14 (DATASIZE) in the Mode Register (MODE).

**Interrupt source**

Select the peripheral interrupt that triggers a DMA burst for the specified channel.

Select SEQ1INT or SEQ2INT to configure the ADC interrupt as interrupt source.

Select XINT1, XINT2, or XINT13 to configure GPIO pin 0 to 31 as an external interrupt source. Select XINT3 to XINT7 to configure GPIO pin 32 to 63 as an external interrupt source.

For more information about configuring XINT, consult the following references:

- *TMS320x2833x, 2823x External Interface (XINTF) User's Guide*, Literature Number: SPRU949, available on the TI Web site.
- *TMS320x2833x System Control and Interrupts*, Literature Number: SPRUFB0, available on the TI Web site.
- *TMS320F28M35x/ TMS320F28M36x*, Literature Number: SPRUH22F/ SPRUHE8B available on the TI Web site.
- The C280x/C2802x/C2803x/C2805x/C2806x/C2833x/C2834x/F28M3x/F2807x/F2837xD/F2837xS/F28004x GPIO Digital Input and C280x/C2802x/C2803x/C2805x/C2806x/C2833x/C2834x/F28M3x/F2807x/F2837xD/F2837xS/F28004x GPIO Digital Output block reference sections.

Drop-down menu items from TINT0 to MREVTB may require manual configuration.

Select ePWM1SOCA through ePWM6SOCB to configure the ePWM interrupt as an interrupt source. Note that not all revisions of the TMS320F2833x silicon provide ePWM interrupts as sources for DMA transfers. For more information about silicon revisions consult the following reference:

*TMS320x2833x, 2823x Silicon Errata/ TMS320F28M35x/ TMS320F28M36x*, Literature Number: SPRZ272/ SPRUH22F/ SPRUHE8B, available on the TI Web site.

The **Interrupt source** parameter corresponds to bit 4-0 (PERINTSEL) in the Mode Register (MODE).

**External pin(GPIO)**

When you set **Interrupt source** is set to an external interface (XINT[#]), specify the GPIO pin number from which the interrupt originates.

This parameter corresponds to the GPIO XINTn, XNMI Interrupt Select (GPIOXINTnSEL, GPIOXNMISEL) Registers.

For more information, consult the *TMS320x2833x / TMS320F28M35x/ TMS320F28M36x System Control and Interrupts Reference Guide*, Literature Number SPRUFB0/ SPRUH22F/ SPRUHE8B available from the TI Web site.

**SRC wrap**

Specify the number of bursts before returning the current source address pointer to the **Source Begin Address** value. To disable wrapping, enter a value for **SRC wrap** that is greater than the **Transfer** value.

This parameter corresponds to bits 15-0 (SRC_WRAP_SIZE) in the Source Wrap Size Register (SRC_WRAP_SIZE).

**DST wrap**

Specify the number of bursts before returning the current destination address pointer to the **Destination Begin Address** value. To disable wrapping, enter a value for **DST wrap** that is greater than the **Transfer** value.

This parameter corresponds to bits 15-0 (DST_WRAP_SIZE) in the Destination Wrap Size Register (DST_WRAP_SIZE).

**SRC Begin address**

Set the starting address for the current source address pointer. The DMA module points to this address at the beginning of a transfer and returns to it as specified by the **SRC wrap** parameter.

This parameter corresponds to bits 21-0 (BEGADDR) in the Active Source Begin Register (SRC_BEG_ADDR).

**DST Begin address**

Set the starting address for the current destination address pointer. The DMA module points to this address at the beginning of a transfer and returns to it as specified by the **DST wrap** parameter.

This parameter corresponds to bits 21-0 (BEGADDR) in the Active Destination Begin Register (DST_BEG_ADDR).

**Burst**

Specify the number of 16-bit words in a burst, from 1 to 32. The DMA module must complete a burst before it can service the next channel.

Set the **Burst** value for the peripheral the DMA module is servicing. For the ADC, the value equals the number of ADC registers used, up to 16. For multichannel buffered serial ports (McBSP), which lack FIFOs, the value is 1.

For RAM, the value can range from 1 to 32.

This parameter corresponds to bits 4-0 (BURSTSIZE) in the Burst Size Register (BURST_SIZE).

---

**Note** This parameter is based on a 16-bit word size. If you set **Data size** to `32 bit`, double the value of this parameter.

---

### Transfer

Specify the number of bursts in a transfer, from 1 to 65536.

This parameter corresponds to bits 15-0 (TRANSFERSIZE) in the Transfer Size Register (TRANSFER_SIZE).

### SRC Burst step

Set the number of 16-bit words by which to increment or decrement the current address pointer before the next burst. Enter a value from –4096 (decrement) to 4095 (increment).

To disable incrementing or decrementing the address pointer, set **Burst step** to 0. For example, because McBSP does not use FIFO, configure DMA to maintain the sequence of the McBSP data by moving each word of the data individually.

Accordingly, when you use DMA to transmit or receive McBSP data, set **Burst size** to 1 word and **Burst step** to 0.

This parameter corresponds to bits 15-0 (SRCBURSTSTEP) in the Source Burst Step Size Register (SRC_BURST_STEP).

---

**Note** This parameter is based on a 16-bit word size. If you set **Data size** to `32 bit`, double the value of this parameter.

---

### DST Burst step

Set the number of 16-bit words by which to increment or decrement the current address pointer before the next burst. Enter a value from –4096 (decrement) to 4095 (increment).

To disable incrementing or decrementing the address pointer, set **Burst step** to 0. For example, because McBSP does not use FIFO, configure DMA to maintain the sequence of the McBSP data by moving each word of the data individually. Accordingly, when you use DMA to transmit or receive McBSP data, set **Burst size** to 1 word and **Burst step** to 0.

This parameter corresponds to bits 15-0 (DSTBURSTSTEP) in the Destination Burst Step Size Register (DST_BURST_STEP).

**Note** This parameter is based on a 16-bit word size. If you set **Data size** to `32 bit`, double the value of this parameter.

### SRC Transfer step

Set the number of 16-bit words by which to increment or decrement the current address pointer before the next transfer. Enter a value from –4096 (decrement) to 4095 (increment).

To disable incrementing or decrementing the address pointer, set **Transfer step** to 0.

This parameter corresponds to bits 15-0 (SRCTRANSFERSTEP) Source Transfer Step Size Register (SRC_TRANSFER_STEP).

If DMA is configured to perform memory wrapping (if **SRC wrap** is enabled) the corresponding source **Transfer step** does not alter the results.

**Note** This parameter is based on a 16-bit word size. If you set **Data size** to `32 bit`, double the value of this parameter.

### DST Transfer step

Set the number of 16-bit words by which to increment or decrement the current address pointer before the next transfer. Enter a value from –4096 (decrement) to 4095 (increment).

To disable incrementing or decrementing the address pointer, set **Transfer step** to 0.

This parameter corresponds to bits 15-0 (DSTTRANSFERSTEP) Destination Transfer Step Size Register (DST_TRANSFER_STEP).

If DMA is configured to perform memory wrapping (if **DST wrap** is enabled) the corresponding destination **Transfer step** does not alter the results.

**Note** This parameter is based on a 16-bit word size. If you set **Data size** to `32 bit`, double the value of this parameter.

### SRC Wrap step

Set the number of 16-bit words by which to increment or decrement the SRC_BEG_ADDR address pointer when a wrap event occurs. Enter a value from –4096 (decrement) to 4095 (increment).

This parameter corresponds to bits 15-0 (WRAPSTEP) in the Source Wrap Step Size Registers (SRC_WRAP_STEP).

---

**Note** This parameter is based on a 16-bit word size. If you set **Data size** to `32 bit`, double the value of this parameter.

---

### DST Wrap step

Set the number of 16-bit words by which to increment or decrement the DST_BEG_ADDR address pointer when a wrap event occurs. Enter a value from –4096 (decrement) to 4095 (increment).

This parameter corresponds to bits 15-0 (WRAPSTEP) in the Destination Wrap Step Size Registers (DST_WRAP_STEP).

---

**Note** This parameter is based on a 16-bit word size. If you set **Data size** to `32 bit`, double the value of this parameter.

---

### Generate interrupt

Enable this parameter to have the DMA channel send an interrupt to the CPU via the PIE at the beginning or end of a data transfer.

This parameter corresponds to bit 15 (CHINTE) and bit 9 (CHINTMODE) in the Mode Register (MODE).

### Enable one shot mode

Enable this parameter to have the DMA channel complete an entire *transfer* in response to an interrupt event trigger.

This option allows a single DMA channel and peripheral to dominate resources, and may streamline processing, but it also creates the potential for resource conflicts and delays.

Disable this parameter to have DMA complete one *burst* per channel per interrupt.

### Sync enable

When **Interrupt source** is set to SEQ1INT, enable this parameter to reset the DMA wrap counter when it receives the ADCSYNC signal from SEQ1INT. This way, the wrap counter and the ADC channels remain synchronized with each other.

If **Interrupt source** is not set to SEQ1INT, **Sync enable** does not alter the results.

This parameter corresponds to bit 12 (SYNCE) of the Mode Register (MODE).

### Enable continuous mode

Select this parameter to leave the DMA channel enabled upon completing a transfer. The channel will wait for the next interrupt event trigger.

Clear this parameter to disable the DMA channel upon completing a transfer. The DMA module disables the DMA channel by clearing the RUNSTS bit in the CONTROL register when it completes the transfer. To use the channel again, first reset the RUN bit in the CONTROL register.

### Enable DST sync mode

When **Sync enable** is enabled, enabling this parameter resets the destination wrap counter (DST_WRAP_COUNT) when the DMA module receives the SEQ1INT interrupt/ADCSYNC signal.

Disabling this parameter resets the source wrap counter (SCR_WRAP_COUNT) when the DMA module receives the SEQ1INT interrupt/ADCSYNC signal.

This parameter is associated with bit 13 (SYNCSEL) in the Mode Register (MODE).

### Set channel 1 to highest priority

This parameter is only available for DMA_ch1.

Enable this setting when DMA channel 1 is configured to handle high-bandwidth data, such as ADC data, and the other DMA channels are configured to handle lower-priority data.

When enabled, the DMA module services each enabled channel sequentially until it receives a trigger from channel 1.

Upon receiving the trigger, DMA interrupts its service to the current channel at the end of the current word, services the channel 1 burst that generated the trigger, and then continues servicing the current channel at the beginning of the next word.

Disable this channel to give each DMA channel equal priority, or if DMA channel 1 is the only enabled channel.

When disabled, the DMA module services each enabled channel sequentially.

This parameter corresponds to bit 0 (CH1PRIORITY) in the Priority Control Register 1 (PRIORITYCTRL1).

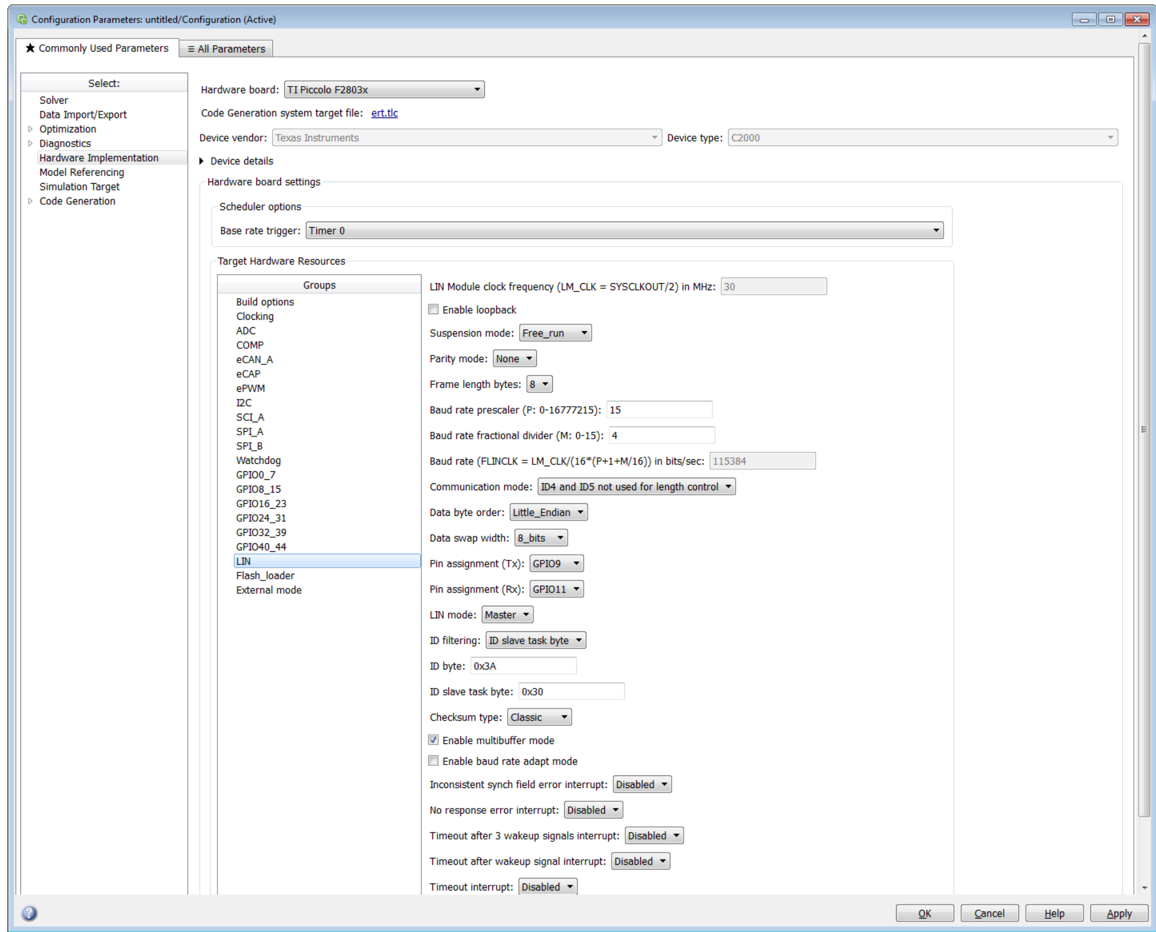
**Enable overflow interrupt**

Enable this parameter to have the DMA channel send an interrupt to the CPU via PIE if the DMA module receives a peripheral interrupt while a previous interrupt from the same peripheral is waiting to be serviced.

This parameter is typically used for debugging during the development phase of a project.

The **Enable overflow interrupt** parameter corresponds to bit 7 (OVRINTE) of the Mode Register (MODE), and involves the Overflow Flag Bit (OVRFLG) and Peripheral Interrupt Trigger Flag Bit (PERINTFLG).

# C28x-LIN




For detailed information on the LIN module, see *TMS320F2803x Piccolo Local Interconnect Network (LIN) Module*, Literature Number SPRUGE2, available at the Texas Instruments Web site.

The following options configure all LIN Transmit and LIN Receive blocks within a model.

**LIN Module clock frequency (LM_CLK = SYSCLKOUT/2) in MHz**

Displays the frequency of the LIN module clock in MHz.

**Enable loopback**

To enable LIN loopback testing, select this option. While this option is enabled, the LIN module does the following:

- Internally redirects the LINTX output to the LINRX input.
- Puts the external LINTX pin into high state.
- Puts the external LINRX pin into a high impedance state.

The default is disabled (unchecked).

**Suspension mode**

Use this option to configure how the LIN state machine behaves while you debug the program on an emulator. If you select `Hard_abort`, entering LIN debug mode halts the transmissions and counters.

The transmissions and counters resume when you exit LIN debug mode. If you select `Free_run`, entering LIN debug mode allows the current transmit and receive functions to complete.

The default is `Free_run`.

**Parity mode**

Use this option to configure parity checking:

- To disable parity checking, select `None`.
- To enable odd parity checking, select `Odd`.
- To enable even parity checking, select `Even`.

The default is `None`.

In order for **ID parity error interrupt** in the LIN Receive block to generate interrupts, also enable **Parity mode**.

**Frame length bytes**

Set the number of data bytes in the response field, from 1 to 8 bytes.

The default is `8` bytes.

**Baud rate prescaler (P: 0-16777215)**

To set the LIN baud rate manually, enter a prescaler value, from 0 to 16777215. Click **Apply** to update the **Baud rate** display.

The default is `15`.

For more information, consult the “Baud Rate” topic in the TI document, *TMS320F2803x Piccolo Local Interconnect Network (LIN) Module*, Literature Number SPRUGE2.

**Baud rate fractional divider (M: 0–15)**

To set the LIN baud rate manually, enter a fractional divider value, from 0 to 15. Click **Apply** to update the **Baud rate** display.

The default is `4`.

For more information, consult the “Baud Rate” topic in the TI document, *TMS320F2803x Piccolo Local Interconnect Network (LIN) Module*, Literature Number SPRUGE2.

**Baud rate (FLINCLK = LM_CLK/(16*(P+1+M/16)) in bits/sec**

This field displays the baud rate. For more information, see “Setting the LIN baud rate”.

**Communication mode**

Enable or disable the LIN module from using the ID-field bits ID4 and ID5 for length control.

The default is `ID4 and ID5 not used for length control`

**Data byte order**

Set the “endianness” of the LIN message data bytes to `Little_Endian` or `Big_Endian`.

The default is `Little_Endian`.

**Data swap width**

Select `8_bits` or `16_bits`. If you set **Data byte order** to `Big_Endian`, the only available option for **Data swap width** is `8_bits`.

**Pin assignment (Tx)**

Map the LINTX output to a specific GPIO pin.

The default is `GPIO9`.

**Pin assignment (Rx)**

Map the LINRX input to a specific GPIO pin.

The default is `GPIO11`.

**LIN mode**

Put the LIN module in `Master` or `Slave` mode. The default is `Slave`.

In master mode, the LIN node can transmit queries and commands to slaves. In slave mode, the LIN module responds to queries or commands from a master node.

This option corresponds to the CLK_MASTER field in the SCI Global Control Register (SCIGCR1).

**ID filtering**

Select which type of mask filtering comparison the LIN module performs, `ID byte` or `ID slave task byte`.

If you select `ID byte`, the module uses the RECID and ID-BYTE fields in the LINID register to detect a match. If you select this option and enter 0xFF for LINMASK, the LIN module does not report matches.

I you select `ID slave task`. the module uses the RECID an ID-SlaveTask byte to detect a match. If you select this option and enter 0xFF for LINMASK, the LIN module reports matches.

The default is `ID slave task byte`.

**ID byte**

If you set **ID filtering** to `ID byte`, use this option to set the ID BYTE, also known as the “LIN mode message ID”.

In master mode, the CPU writes this value to initiate a header transmission. In slave mode, the LIN module uses this value to perform message filtering.

The default is `0x3A`.

**ID slave task byte**

If you set **ID filtering** to `ID slave task byte`, use this option to set the ID-SlaveTask BYTE. The LIN node compares this byte with the Received ID and determines whether to send a transmit or receive response.

The default is `0x30`.

**Checksum type**

Use this option to select the type of checksum. If you select `Classic`, the LIN node generates the checksum field from the data fields in the response.

If you select `Enhance`, the LIN node generates the checksum field from both the ID field in the header and data fields in the response. LIN 1.3 supports classic checksums only. LIN 2.0 supports both classic and enhanced checksums.

The default is `Classic`.

**Enable multibuffer mode**

When you enable (select) this check box, the LIN node uses transmit and receive buffers instead of just one register. This setting affects various other LIN registers, such as: checksums, framing errors, transmitter empty flags, receiver ready flags, transmitter ready flags.

The default is enabled (checked).

**Enable baud rate adapt mode**

The dialog box displays this option when you set **LIN mode** to `Slave`.

If you enable this option, the slave node automatically adjusts its baud rate to match that of the master node. For this feature to work, first set the **Baud rate prescaler** and **Baud rate fractional divider**.

If you disable this option, the LIN module sets a static baud rate based on the **Baud rate prescaler** and **Baud rate fractional divider**.

The default is disabled (unchecked).

**Inconsistent synch field error interrupt**

The dialog box displays this option when you set **LIN mode** to `Slave`.

If you enable this option, the slave node generates interrupts when it detects irregularities in the synch field. This option is only relevant if you enable **Enable adapt mode**.

The default is `Disabled`.

**No response error interrupt**

The dialog box displays this option when you set **LIN mode** to `Slave`.

If you enable this option, the LIN module generates an interrupt if it does not receive a complete response from the master node within a timeout period.

The default is `Disabled`.

**Timeout after 3 wakeup signals interrupt**

The dialog box displays this option when you set **LIN mode** to `Slave`.

When enabled, the slave node generates an interrupt when it sends three wakeup signals to the master node and does not receive a header in response. (The slave waits 1.5 seconds before sending another series of wakeup signals.)

This interrupt typically indicates the master node is having a problem recovering from low-power or sleep mode.

The default is `Disabled`.

**Timeout after wakeup signal interrupt**

The dialog box displays this option when you set **LIN mode** to `Slave`.

When enabled, the slave node generates an interrupt when it sends a wakeup signal to the master node and does not receive a header in response. (The slave waits 150 milliseconds before sending another series of wakeup signals.)

This interrupt typically indicates the master node is delayed recovering from low-power or sleep mode.

The default is `Disabled`.

**Timeout interrupt**

The dialog box displays this option when you set **LIN mode** to `Slave`.

When enabled, the slave node generates an interrupt after 4 seconds of inactivity on the LIN bus.

The default is `Disabled`.

**Wakeup interrupt**

The dialog box displays this option when you set **LIN mode** to `Slave`.

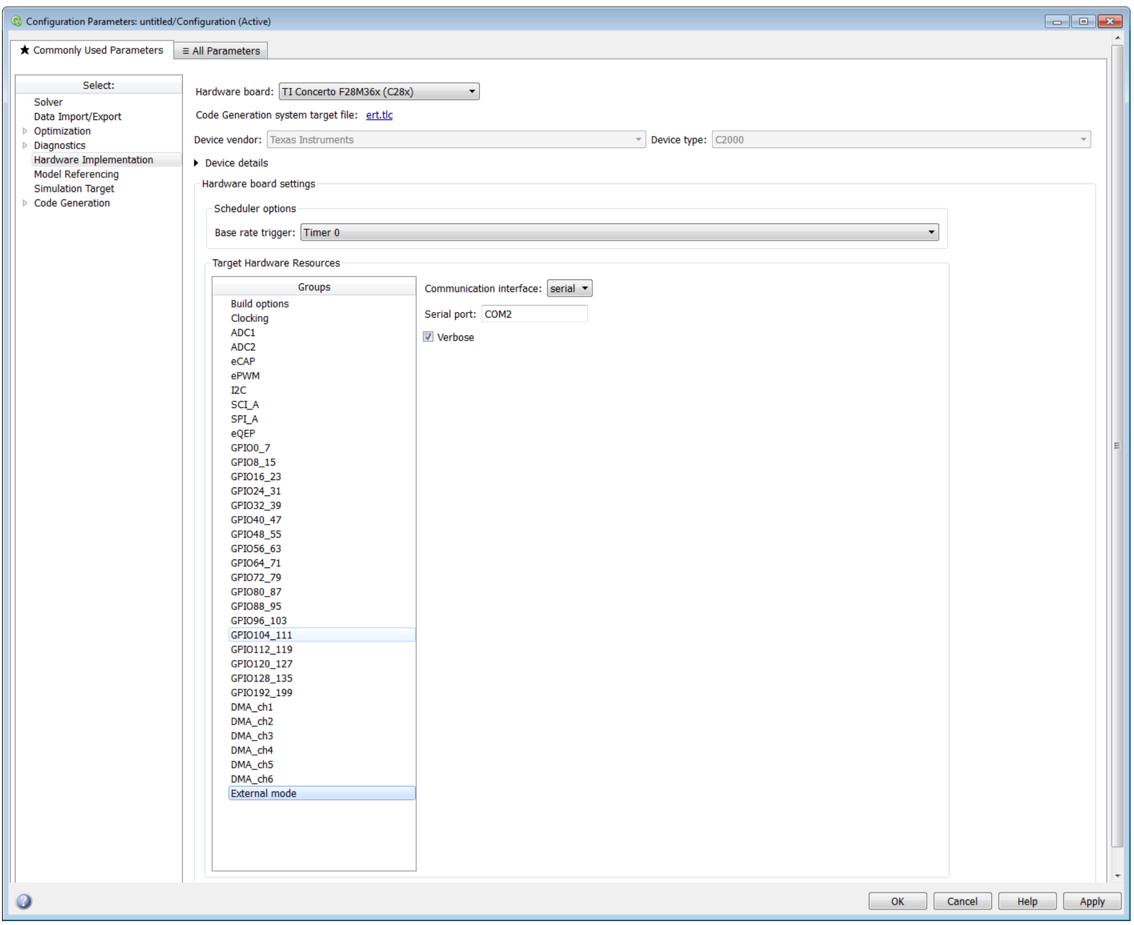
When you enable this option:

- In low-power mode, a LIN slave node generates a wakeup interrupt when it detects the falling edge of a wake-up pulse or a low level on the LINRX pin.
- A LIN slave node that is “awake” generates a wakeup interrupt if it receives a request to enter low-power mode while it is receiving.

- A LIN slave node that is “awake” does not generate a wakeup interrupt if it receives a wakeup pulse.

The default is `Disabled`.

## External mode




Helps you set the external mode settings for your model.

**Communication interface**

Use the ‘serial’ option to run your model in the External mode with serial communication.

**Serial port**

Enter the COMPort used by the target hardware.

To know the comport used by the target hardware on your computer, see “External Mode over Serial Communication” (Embedded Coder Support Package for Texas Instruments C2000 Processors).

**Verbose**

Select this check box to view the External Mode execution progress and updates in the Diagnostic Viewer or in the MATLAB command window.

## Execution profiling

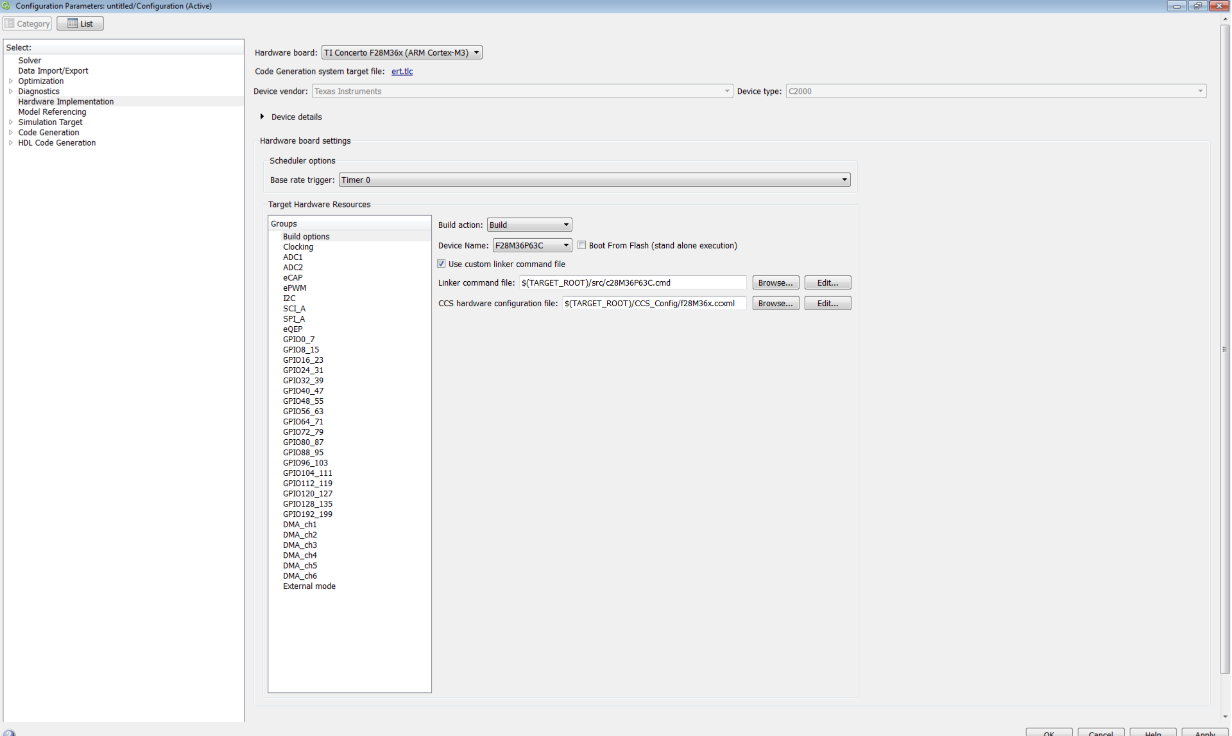
**Number of profiling samples to collect**

Enter the number of profiling samples to collect. You can use real-time execution profiling to check whether the generated code meets the real-time performance requirements. Execution Profiling results can also be used to take actions to enhance design of your system.

# Hardware Implementation Pane: Texas Instruments Concerto

**In this section...**

"Hardware Implementation Pane Overview" on page 13-233
"M3x-Scheduler options" on page 13-233
"C28x / ARM Cortex-M3 - Build options" on page 13-235
"M3x-Clocking" on page 13-238
"M3x-GPIO A–D" on page 13-240
"M3x-UART0–4" on page 13-241
"M3x-Ethernet" on page 13-243
"M3x-PIL" on page 13-244
"External mode" on page 13-246
"C28x-Clocking" on page 13-247
"C28x-ADC" on page 13-250
"C28x-eCAP" on page 13-253
"C28x-ePWM" on page 13-256
"C28x-I2C" on page 13-259
"C28x-SCI_A, C28x-SCI_B, C28x-SCI_C" on page 13-265
"C28x-SPI_A, C28x-SPI_B, C28x-SPI_C, C28x-SPI_D" on page 13-268
"C28x-eQEP" on page 13-271
"C28x-GPIO" on page 13-273
"C28x-DMA_ch[#]" on page 13-278
"External mode" on page 13-287



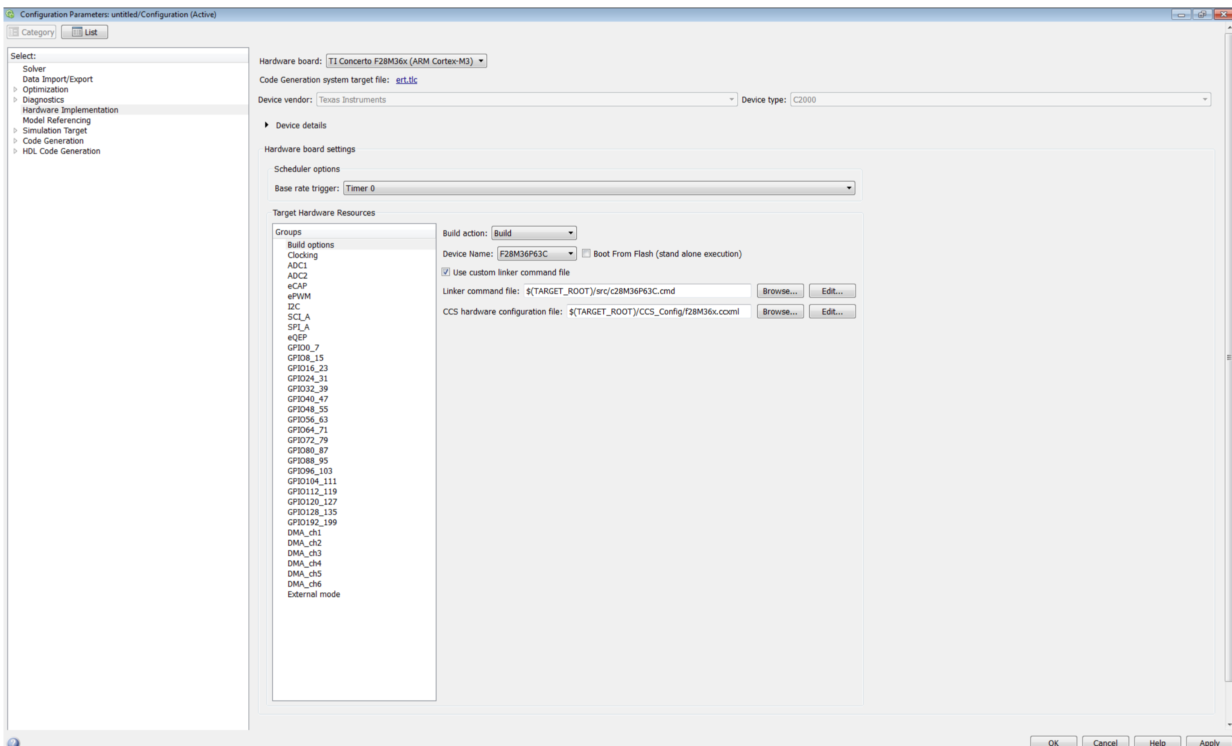

## Hardware Implementation Pane Overview

Configure the parameters for properties of the physical hardware, such as peripherals.

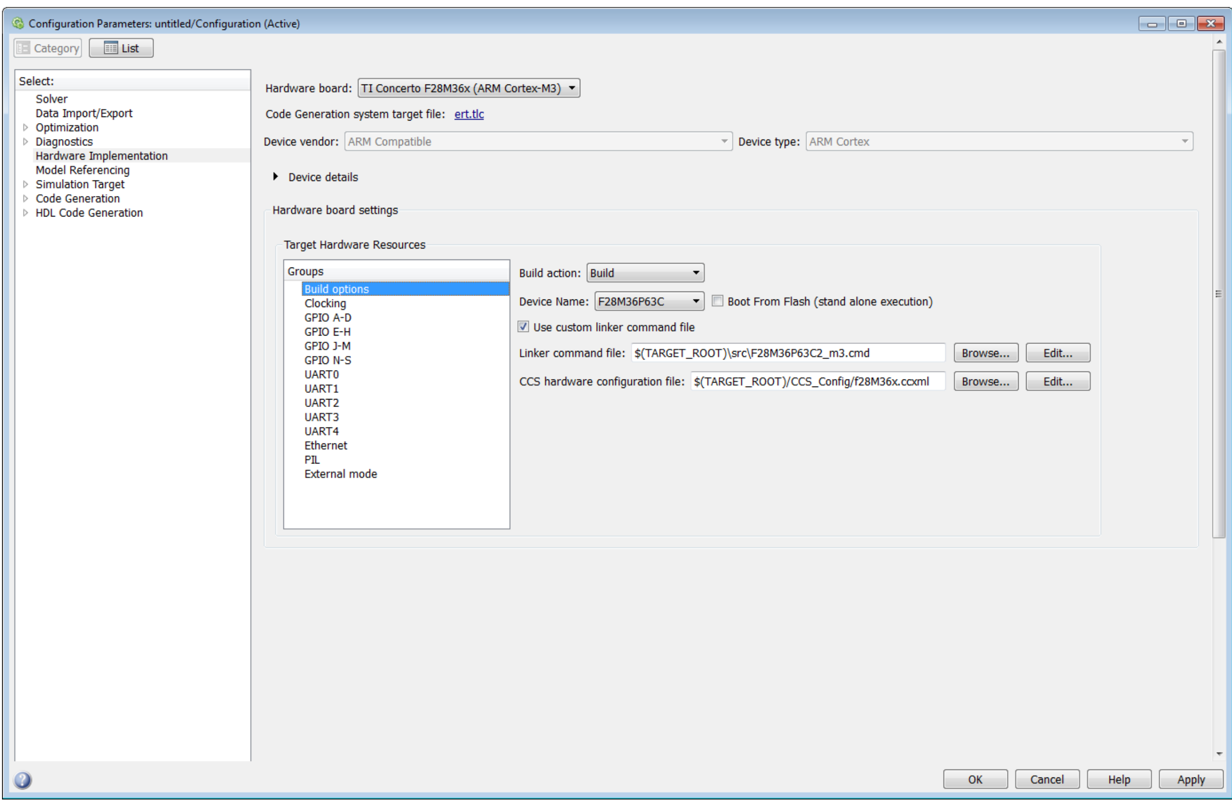
## M3x-Scheduler options

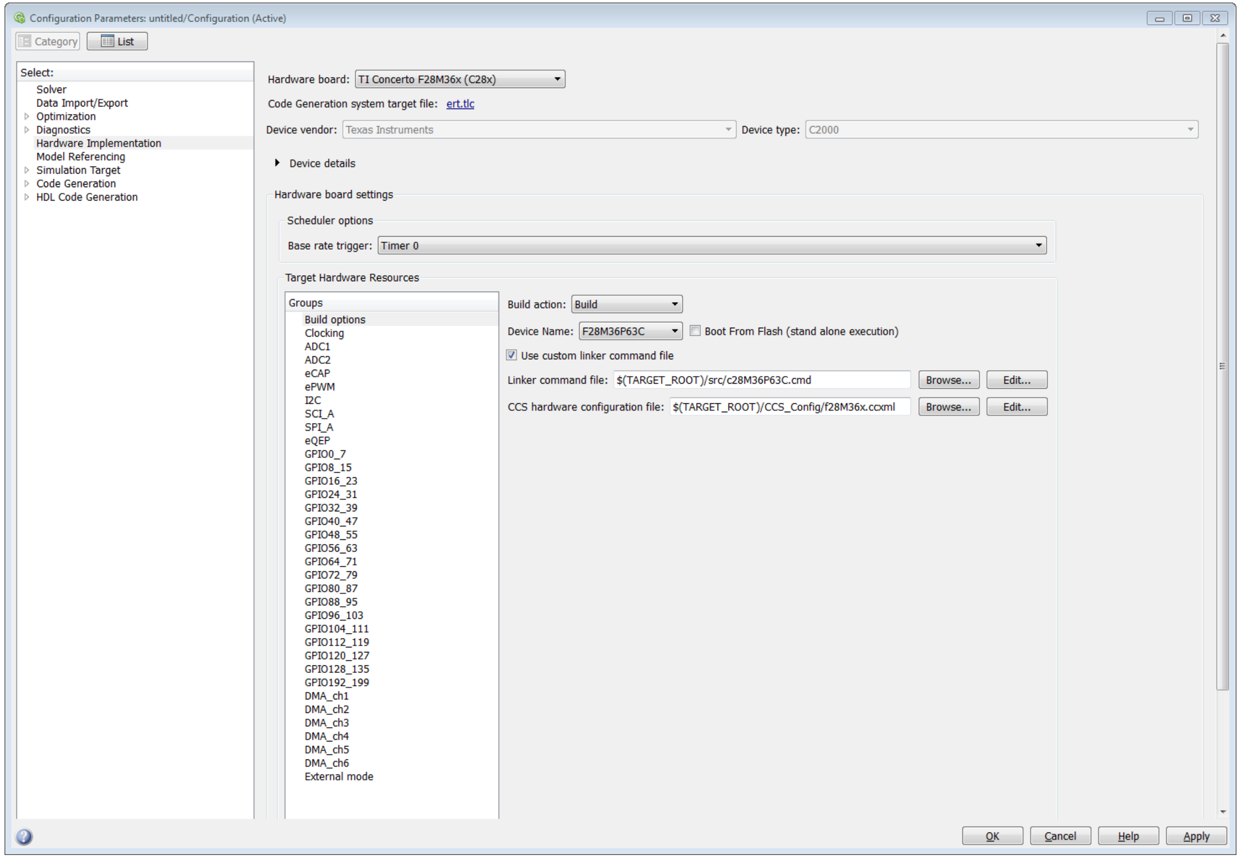
### Scheduler interrupt source

Select the source of the scheduler interrupt.


Configuration Parameters: untitled/Configuration (Active)
Category
List
Select:
Solver
Data Import/Export
Optimization
Diagnostics
Hardware Implementation
Model Referencing
Simulation Target
Code Generation
HDL Code Generation
Hardware board: TI Concerto F28M36x (ARM Cortex-M3)
Code Generation system target file: ert.tlc
Device vendor: Texas Instruments
Device type: C2000
Device details
Hardware board settings
Scheduler options
Base rate trigger: Timer 0
Target Hardware Resources
Groups
Build options
Clocking
ADC1
ADC2
eCAP
ePWM
I2C
SCI_A
SPI_A
eQEP
GPIO0_7
GPIO8_15
GPIO16_23
GPIO24_31
GPIO32_39
GPIO40_47
GPIO48_55
GPIO56_63
GPIO64_71
GPIO72_79
GPIO80_87
GPIO88_95
GPIO96_103
GPIO104_111
GPIO112_119
GPIO120_127
GPIO128_135
GPIO192_199
DMA_ch1
DMA_ch2
DMA_ch3
DMA_ch4
DMA_ch5
DMA_ch6
External mode
Build action: Build
Device Name: F28M36P63C
Boot From Flash (stand alone execution)
Use custom linker command file
Linker command file: $(TARGET_ROOT)/src/c28M36P63C.cmd
CCS hardware configuration file: $(TARGET_ROOT)/CCS_Config/f28M36x.ccxml
Browse...
Edit...
OK
Cancel
Help
Apply

# C28x / ARM Cortex-M3 - Build options






Use the build options to specify how the build process should take place during code generation.

### Build action

The option to specify if you want only 'build' or 'build, load, and run' action during the build process. The `build, load and run` option is supported only for TI Code Composer Studio v5 (C2000), v6 (C2000), and v5(ARM)/v6(ARM) tool chain.

If you select `build, load, and run` option, you must provide the required CCS hardware configuration file.

**Device name**

The option to select a particular device from the selected processor family in the Target hardware parameter on the Code Generation pane.

**Boot From Flash (stand alone execution)**

The option to specify if the application has to load to the flash. If you do not select this option, the application loads to the RAM.

**Use custom linker command file**

The option to indicate that the custom linker command file must be used during the build action. Select this option, if you have your own custom linker file, which you can specify in Linker command file parameter. If you do not select this option, based on the device you have selected, a default custom linker command file will be used.

**Linker command file**

The path to memory description file that is required during linking. For each family of TI processor selected under ‘Target Hardware’, one linker command file will be selected automatically.

For different variant of processor, you can select from the ‘src’ folder inside the Support Package installation path. You can also create custom linker command file and select the file path using **Browse**.

**CCS hardware configuration file**

The Code Composer Studio file required for downloading the application on the hardware. Select one of the .ccxml files from the folder ‘CCS_Config’ folder under Support Package installation folder.

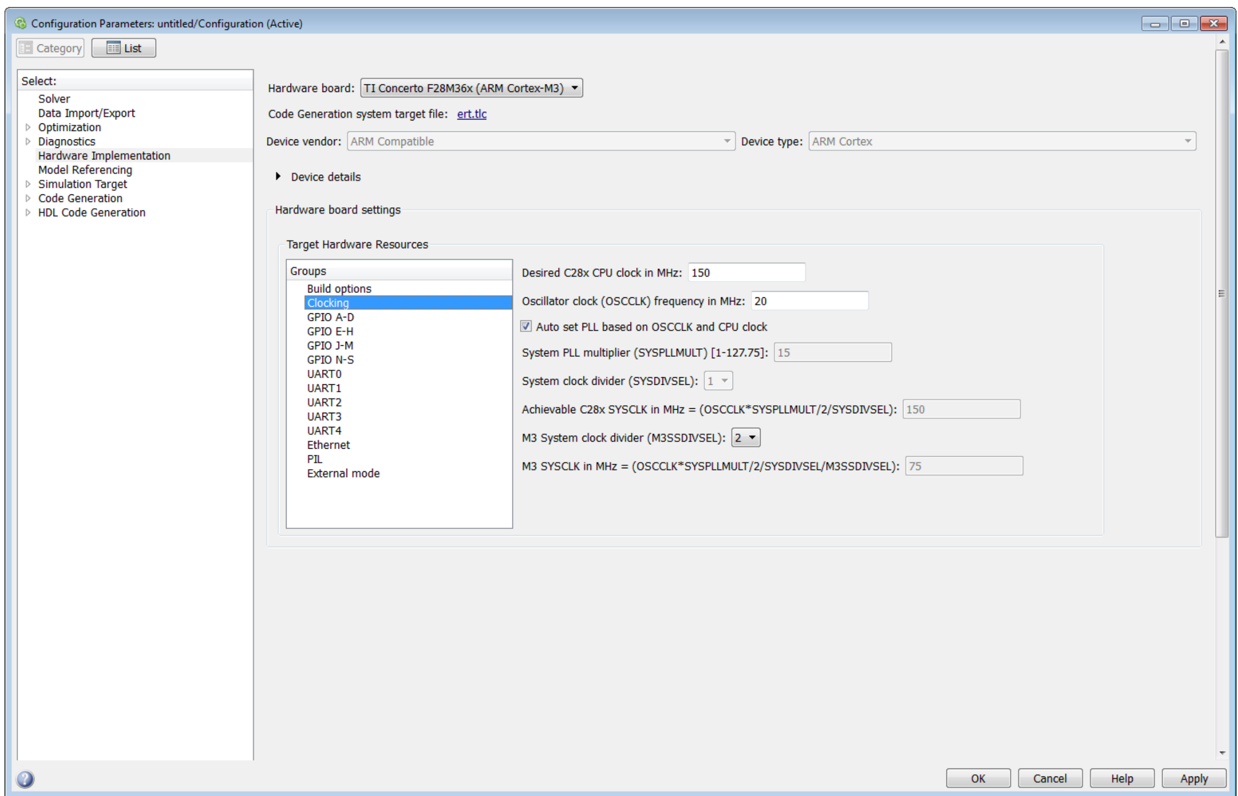
Instead, you can also create your own .ccxml file.

Select the file you created using **Browse**.

The .ccxml files provided with the support package are as follows:

- f28M35x.ccxml – Texas Instruments XDS100v2 USB Emulator_0
- f28M36x.ccxml – Texas Instruments XDS100v2 USB Emulator_0

# M3x-Clocking




### Desired C28x CPU clock in MHz

Specify the expected C28x CPU clock frequency and match the same in your C28x Model. The C28x CLOCK is the same as PLLSYSCLK. The M3 Clock is a factor of M3SSDIVSEL divided by the PLLSYSCLK.

### Oscillator clock (OSCCLK) frequency in MHz

Specify the frequency of the crystal oscillator used in the board. In case of Concerto the crystal oscillator is external to the processor.

### Auto set PLL based on OSCCLK and CPU clock

The option that helps you to set the PLL control register value automatically. When you select this check box, the values in the SYSPLLMULT, SYSDIVSEL , and the

Achievable C28x SYSCLK in MHz parameters are automatically calculated based on the **Desired C28x CPU Clock** value entered on the Board.

**System PLL multiplier (SYSPLLMULT)[1–127.75]**

Specify the system PLL multiplier. You can specify a value in this parameter if **Auto set PLL based on OSCCLK and CPU clock** is not selected. The PLL multiplier is a 9 bit field with 7 bits of the SYSPLLMULT register comprising of the integer portion and the remaining 2 bits for the fractional portion. You can enter a value in the range between 0 to 127.75 with multiples of 0.25 for fractional portion of the value.

**System clock divider (SYSDIVSEL)**

If you select the **Auto set PLL based on OSCCLK and CPU clock** check box, the auto calculated clock divider value achieves the specified CPU Clock value based on the Oscillator clock frequency. Otherwise, you can select a value for Clock divider (SYSDIVSEL).

**Achievable C28x SYSCLK in MHz = (OSCCLK * SYSPLLMULT/ 2/ SYSDIVSEL)**

The auto calculated feedback value that matches most closely to the desired CPU Clock value on the board, based on the values of OSCCLK, SYSPLLMULT, and the SYSDIVSEL.

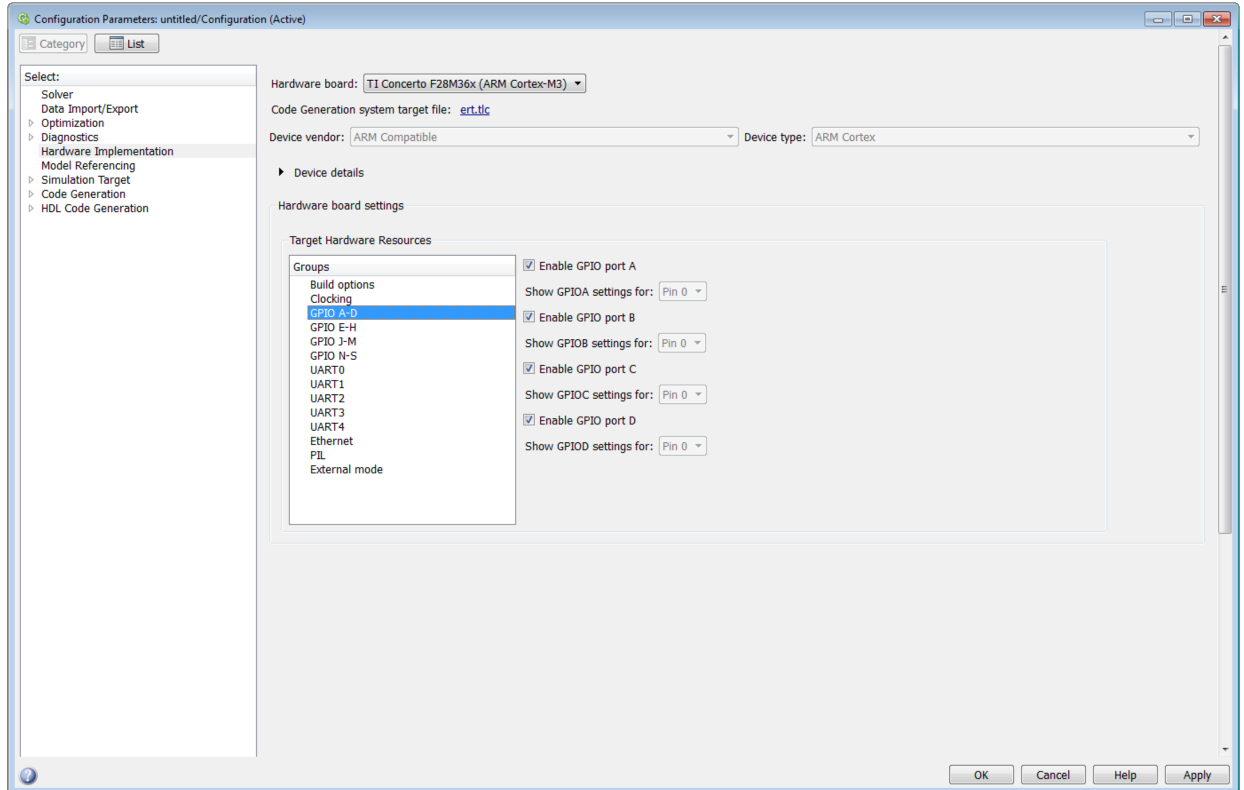
**M3 System clock divider (M3SSDIVSEL)**

Select a value from the options for M3 system clock divider. The C28 CLKIN clock is divided by the selected value to generate the M3 CPU clock.

**M3 SYSCLK in MHz = (OSCCLK * SYSPLLMULT/ 2/ SYSDIVSEL/ M3SSDIVSEL)**

This is the achievable M3 system clock frequency. This is calculated based on the values of OSCCLK, SYSPLLMULT, SYSDIVSEL, and M3SSDIVSEL.

## M3x-GPIO A-D




### Enable GPIO port A

Select this option to enable GPIO port A.

### Show GPIOA settings for

Select GPIO pins from port A for which you want to set the CPU core and the pin type.

### Select the CPU core which controls Pin #

Select the CPU core for the selected GPIO pin.

- Auto detect M3 usage, otherwise set to C28x — This default option detects, if you have used the selected GPIO pin for the M3 block in your model.
- M3 — Select this option to assign the GPIO pin to the M3 CPU.

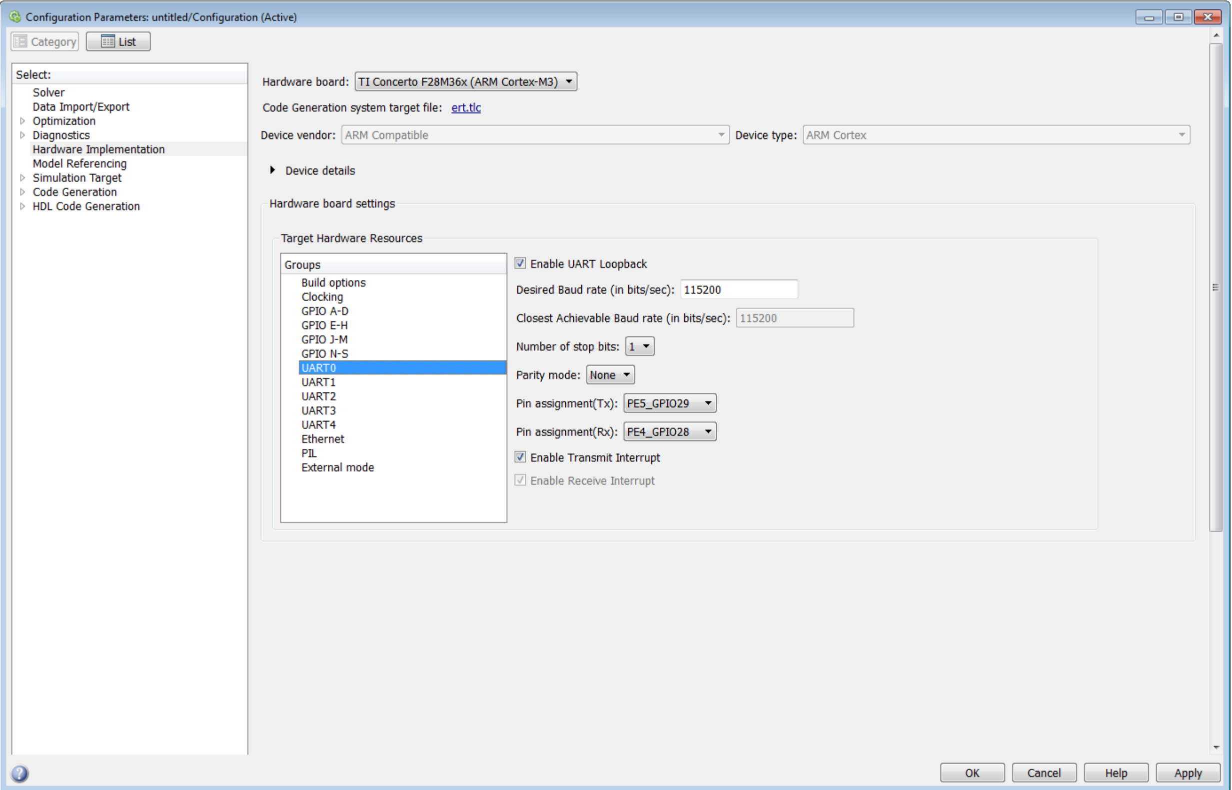
- C28x — Select this option to assign the GPIO pin to the C28x CPU.

### Select the pin type for Pin 0

Select the pull-up and the open-drain options for the selected GPIO pin.

**Note** The above parameter descriptions are also applicable for all the GPIO ports.

## M3x-UART0–4




### Enable UART Loopback

Select this check box to enable data transmission from Tx to Rx buffer. However, selecting this option does not ensure that the data is present in the GPIO MUX.

**Enable M3 UART4 to C28 SCI-A Loopback**

Select this check box to enable data transmission from M3 UART4 to C28 SCI-A. This option is available only for UART4 parameter.

**Desired Baud rate (in bits/sec)**

Specify the desired baud rate of the data transmission.

**Closest Achievable Baud rate (in bits/sec)**

The value in this parameter is calculated based on the desired baud rate that you specify and the system clock frequency. This baud rate is used for the data transfer.

**Number of stop bits**

Select the number of stop bits used to indicate the end of a byte data transmission. The options available:

- `1` — Select this option to indicate there is 1 stop bit at the end of a byte data transmission.
- `2` — Select this option to indicate there are 2 stop bits at the end of a byte data transmission.

**Parity mode**

Select a parity mode that is added at the end of a binary data for error detection.

The options available are:

- `Odd` — Select this option to indicate that odd parity is used for data transmission.

  In odd parity mode, the parity bit is set to ‘1’ if the sum of bits with the value ‘1’ is even and the parity bit is set to ‘0’, if the sum of bits with the value ‘1’ is odd.
- `Even` — Select this option to indicate that even parity is used for data transmission.

  In even parity mode, the parity bit is set to ‘1’, if the sum of bits with the value ‘1’ is odd and the parity bit is set to ‘0’, if the sum of bits with the value ‘1’ is even.
- `One` — Select this option to indicate that the parity bit is always ‘1’.
- `Zero` — Select this option to indicate that the parity bit is always ‘0’.

**Pin assignment(Tx)**

Select a GPIO pin as the UART pin for data transmission. By default, the GPIO29 is hardwired as the Tx GPIO to the FTDI chip.

**Pin assignment(Rx)**

Select a GPIO pin as UART pin for data reception. By default, the GPIO28 is hardwired as the Rx GPIO to the FTDI chip.

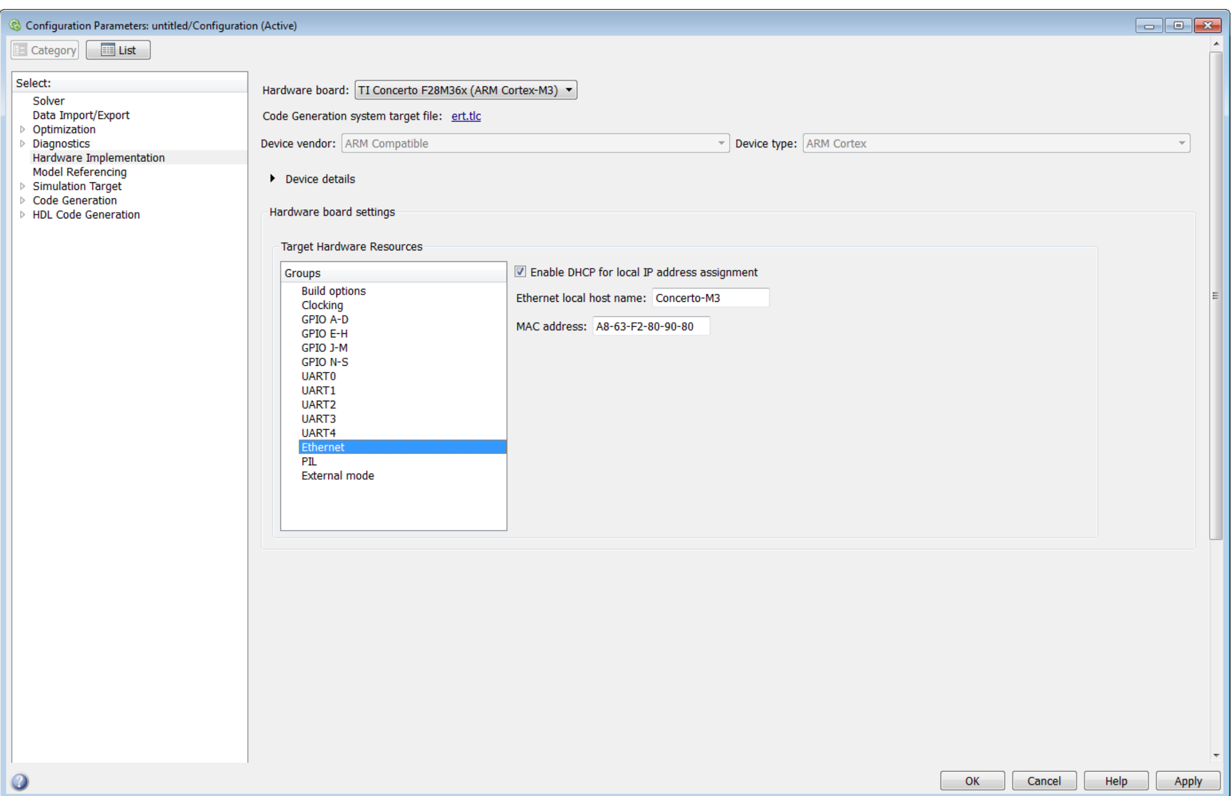
**Enable Transmit Interrupt**

Select this check box to enable the transmit interrupt.

**Enable Receive Interrupt**

Select this check box to enable the receive interrupt.

## M3x-Ethernet



**Enable DHCP for local IP address assignment**

Select this check box to configure the board to get an IP address from the local DHCP server on the network.

**Local IP address**

Enter the IP address of the board.

**Subnet mask**

Enter the subnet mask for the board. A subnet mask divides an IP address into network address and a host address.

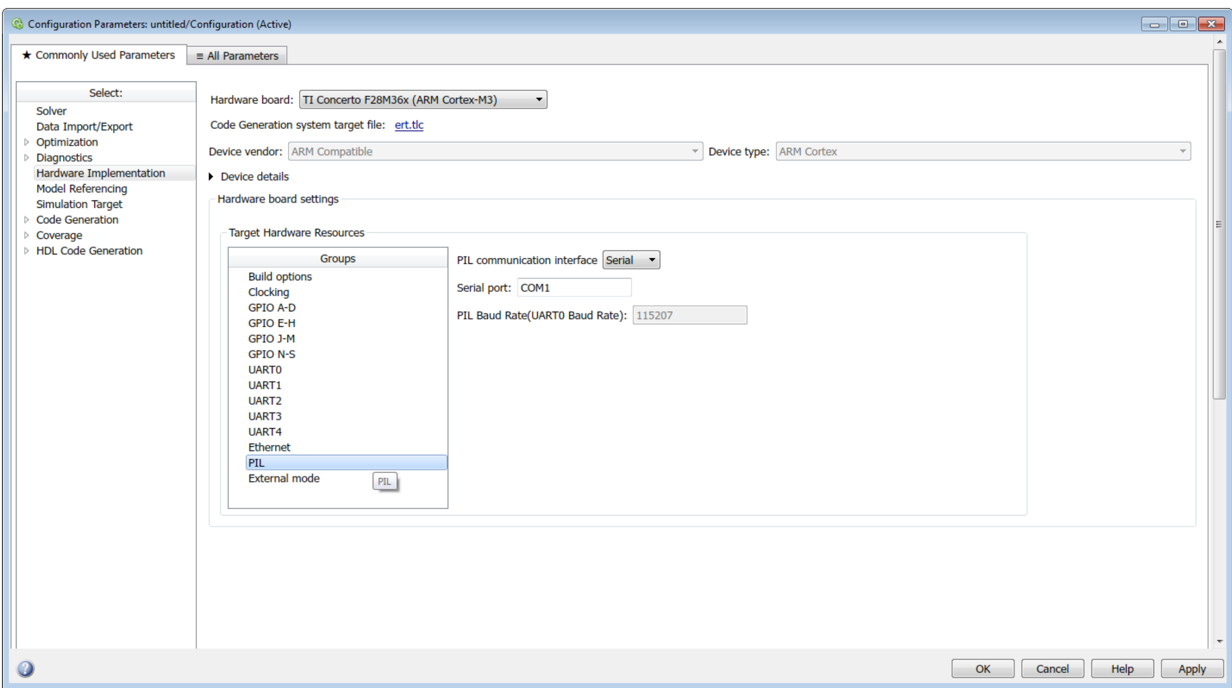
**Ethernet local host name**

Enter the local host name.

**MAC address**

Enter the MAC address.

## M3x-PIL



**PIL communication interface**

Select the communication interface for PIL. The available options are: `Serial` and `TCP/IP`.

**Serial port**

Enter the serial port used by the target hardware.

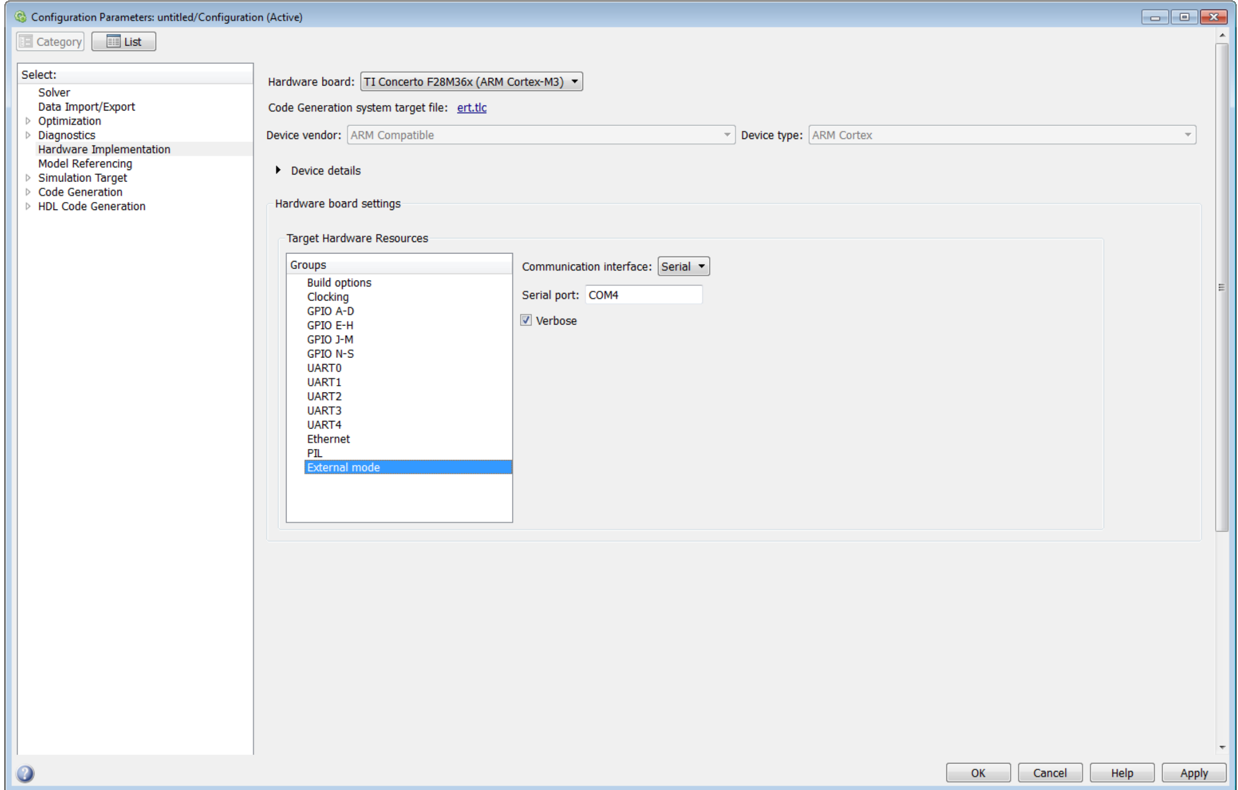
**PIL Baud Rate (UART) Baud rate)**

This is the PIL baud rate used by the target. This is based on the baud rate that you specify in the **Desired Baud rate (in bits/sec)** parameter for UART0.

**Ethernet port**

This is the Ethernet port used for PIL communication. This parameter appears only when you select `TCP/IP` option in **PIL communication interface** parameter.

## External mode




### Communication interface

Use the ‘serial’ option to run your model in the External mode with serial communication.

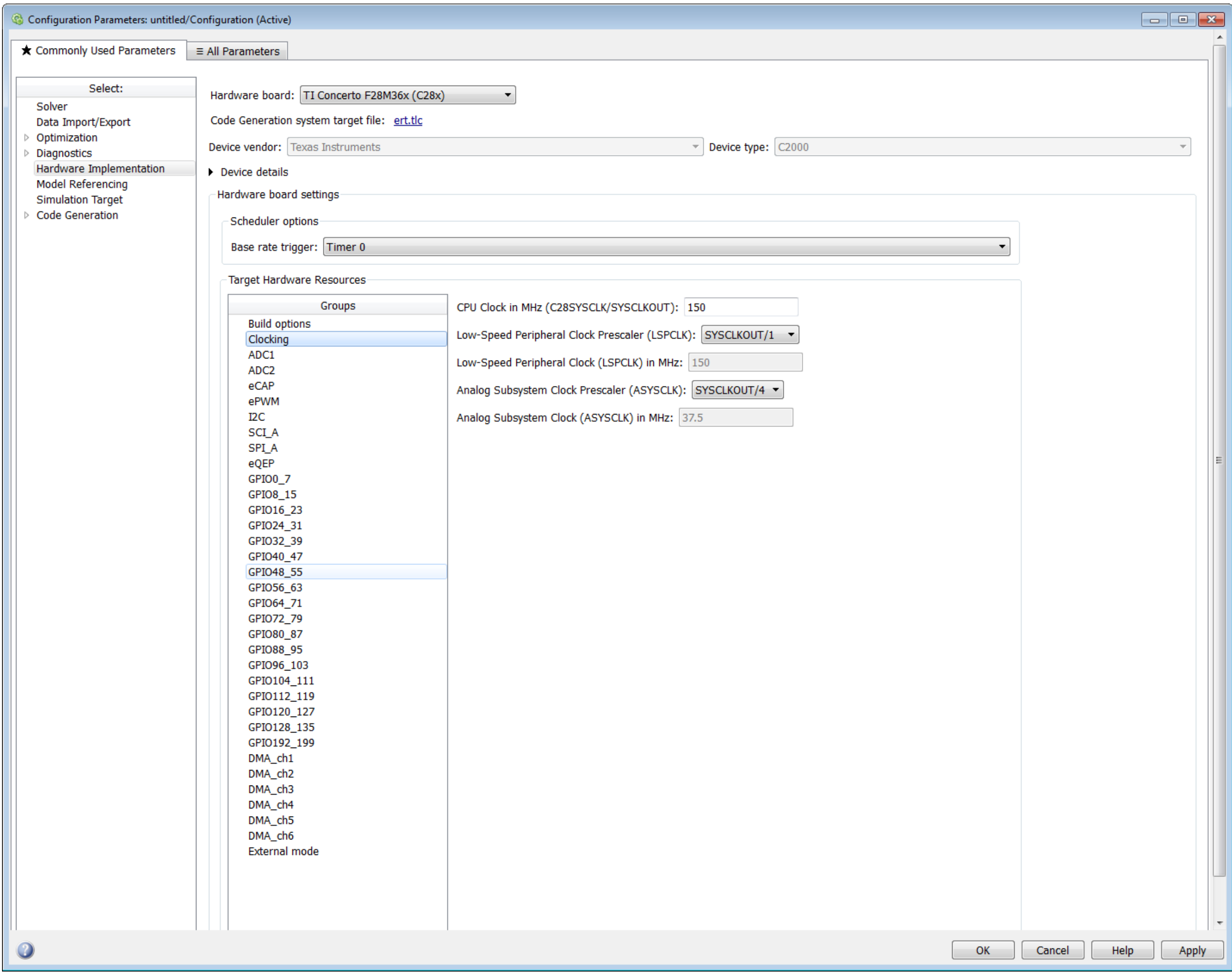
### Serial port

Enter the serial port used by the target hardware.

### Verbose

Select this check box to view the External Mode execution progress and updates in the Diagnostic Viewer or in the MATLAB command window.

## C28x-Clocking




Use the clocking options to help you achieve the CPU Clock rate specified on the board. The default clocking values run the CPU clock (CLKIN) at its maximum frequency. The parameters use the external oscillator frequency on the board (OSCCLK) that is recommended by the processor vendor.

You can get feedback on the closest achievable SYSCLKOUT value with the specified Oscillator clock frequency by selecting the **Auto set PLL based on OSCCLK and CPU**

**clock** check box. Alternatively, you can manually specify the PLL value for the SYSCLKOUT value calculation.

Change the clocking values if:

- You want to change the CPU frequency.
- The external oscillator frequency differs from the value recommended by the manufacturer.

To determine the CPU frequency (CLKIN), use the following equation:

CLKIN = (OSCCLK * PLLCR) / (DIVSEL or CLKINDIV)

- CLKIN is the frequency at which the CPU operates, also known as the CPU clock.
- OSCCLK is the frequency of the oscillator.
- **PLLCR** is the PLL Control Register value.
- **CLKINDIV** is the Clock in Divider.
- **DIVSEL** is the Divider Select.

The availability of the DIVSEL or CLKINDIV parameters changes depending on the processor that you select. If neither parameter is available, use the following equation:

CLKIN = (OSCCLK * PLLCR) / 2

In case of Concerto C28x processor, make sure to match the Achievable C28x SYSCLK in MHz with the value entered here.

In the **CPU clock** parameter, enter the resulting CPU clock frequency (CLKIN).

For more information, see the "PLL-Based Clock Module" section in the Texas Instruments *Reference Guide* for your processor.

**Use internal oscillator**

Use the internal zero pin oscillator on the CPU. This parameter is enabled by default.

**Oscillator clock (OSCCLK) frequency in MHz**

The oscillator frequency that is used in the processor. This parameter is not available for TI Concerto F28M35x/ F28M36x processors.

**Auto set PLL based on OSCCLK and CPU clock**

The option that helps you set the PLL control register value automatically. When you select this check box, the values in the PLLCR, DIVSEL , and the Closest achievable

SYSCLKOUT in MHz parameters are automatically calculated based on the **CPU Clock** value entered on the Board. This parameter is not available for TI Concerto F28M35x/ F28M36x processors.

**PLL control register (PLLCR)**

If you select the **Auto set PLL based on OSCCLK and CPU clock** check box, the auto calculated control register value achieves the specified CPU Clock value, based on the Oscillator clock frequency. Otherwise, you can select a value for PLL control register. This parameter is not available for TI Concerto F28M35x/ F28M36x processors.

**Clock divider (DIVSEL)**

If you select the **Auto set PLL based on OSCCLK and CPU clock** check box, the auto calculated clock divider value achieves the specified CPU Clock value based on the Oscillator clock frequency. Otherwise, you can select a value for Clock divider (DIVSEL). This parameter is not available for TI Concerto F28M35x/ F28M36x processors.

**Closest achievable SYSCLKOUT in MHz = (OSCCLK*PLLCR)/DIVSELClosest achievable SYSCLKOUT in MHz = (OSCCLK*PLLCR)/CLKINDIV**

The auto calculated feedback value that matches most closely to the desired CPU Clock value on the board, based on the values of OSCCLK, PLLCR, and the DIVSEL. This parameter is not available for TI Concerto F28M35x/ F28M36x processors.

**Low-Speed Peripheral Clock Prescaler (LSPCLK)**

The value by which to scale the LSPCLK. This value is based on the SYSCLKOUT.

**Low-Speed Peripheral Clock (LSPCLK) in MHz**

This value is calculated based on LSPCLK Prescaler. Example: SPI uses a LSPCLK.

**High-Speed Peripheral Clock Prescaler (HSPCLK)**

The value by which to scale the HSPCLK. This value is based on the SYSCLKOUT.

**High-Speed Peripheral Clock (HSPCLK) in MHZ**

This value is calculated based on HSPCLK Prescaler. Example: ADC uses a HSPCLK.

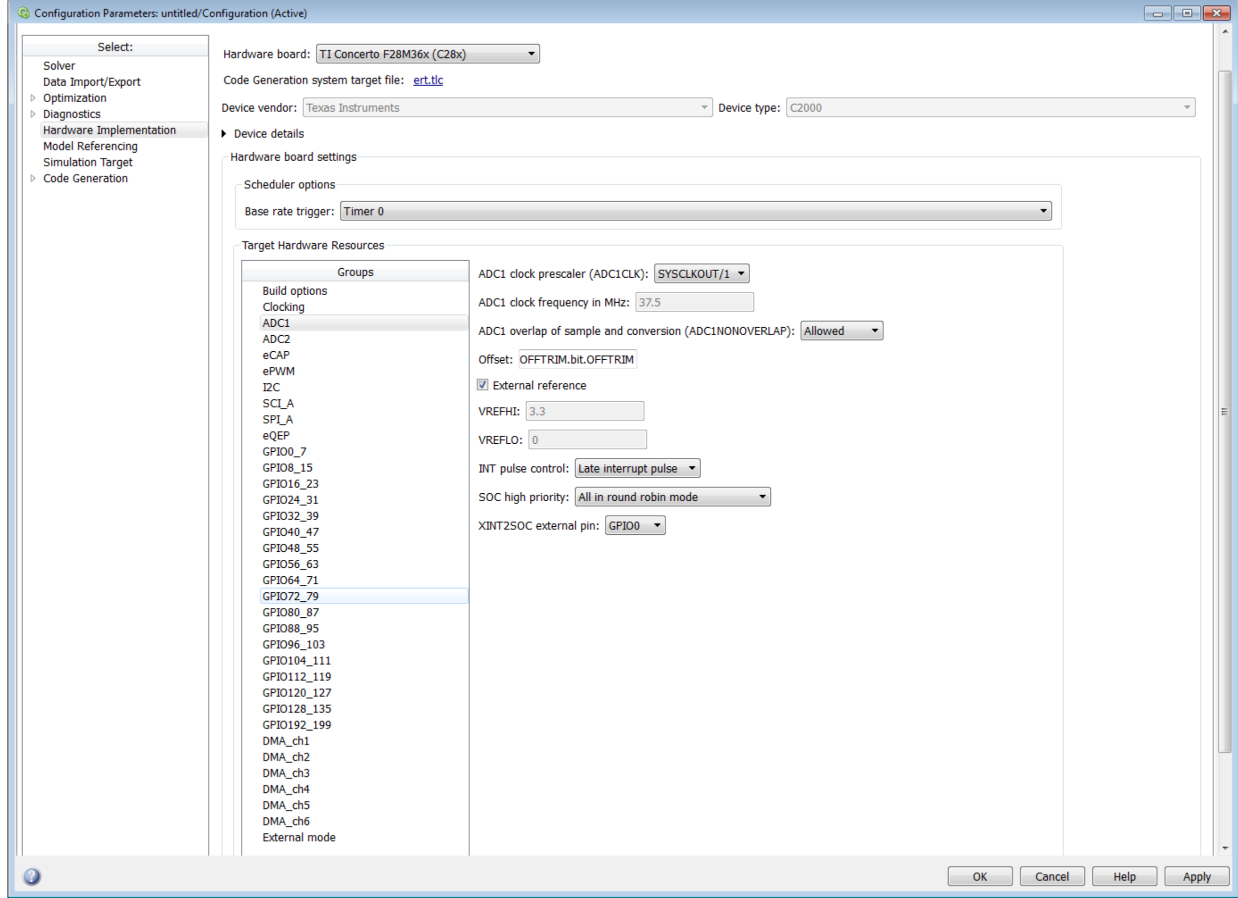
**Analog Common Interface Bus Clock (ACIB)**

The value by which to scale the bus clock. This option is available only for TI Concerto F28M35x/ F28M36x processors.

**Analog Common Interface Bus Clock (ACIB) in MHz**

This value is calculated based on the ACIB value. This option is available only for TI Concerto F28M35x/ F28M36x processors.

## C28x-ADC




The high-speed peripheral clock (HSPCLK) controls the internal timing of the ADC module. The ADC derives the operating clock speed from the HSPCLK speed in several prescaler stages. For more information about configuring these scalers, refer to "Configuring ADC Parameters for Acquisition Window Width".

You can set the following parameters for the ADC clock prescaler:

**Select the CPU core which controls ADC_x module**

Select the CPU core to control the ADC module.

**ADC clock prescaler (ADCCLK)**

The option to select the ADCCLK divider for processors c2802x, c2803x, c2806x, F28M3x, F2807x, or F2837x.

**ADC clock frequency in MHz**

The clock frequency for ADC. This is a read-only field and the value in this field is based on the value you select in **ADC clock prescaler (ADCCLK)**.

**ADC overlap of sample and conversion (ADC#NONOVERLAP)**

The option to enable or disable overlap of sample and conversion.

**ADC clock prescaler (ADCLKPS)**

The HSPCLK speed is divided by this 4-bit value as the first step in deriving the core clock speed of the ADC. The default value is 3.

**ADC Core clock prescaler (CPS)**

After dividing the HSPCLK speed by the **ADC clock prescaler (ADCLKPS)** value, setting the **ADC clock prescaler (ADCLKPS)** parameter to 1, the default value, divides the result by 2.

**ADC Module clock (ADCCLK = HSPCLK/ADCLKPS*2)/(CPS+1)) in MHz**

The clock to the ADC module and indicates the ADC operating clock speed.

**Acquisition window prescaler (ACQ_PS)**

This value does not directly alter the core clock speed of the ADC. It serves to determine the width of the sampling or acquisition period. The higher the value, the wider is the sampling period. The default value is 4.

**Acquisition window size ((ACQ_PS+1)/ADCCLK) in micro seconds/channel**

Acquisition window size determines for what time duration the sampling switch is closed. The width of SOC pulse is ADCTRL1[11:8] + 1 times the ADCLK period.

**Offset**

Enter the offset value.

**Use external reference 2.048VExternal reference**

By default, an internally generated band gap voltage reference supplies the ADC logic. However, depending on application requirements, you can enable the external reference so the ADC logic uses an external voltage reference instead. Select the check box to use a 2.048V external voltage reference.

**Use external reference**

By default, an internally generated band gap voltage reference supplies the ADC logic. However, depending on application requirements, you can enable the external

reference so the ADC logic uses an external voltage reference instead. Select the check box to use an external voltage reference.

**Continuous mode**

When the ADC generates an end of conversion (EOC) signal, generate an ADCINT# interrupt whether the previous interrupt flag has been acknowledged or not.

**ADC offset correction (OFFSET_TRIM: -256 to 255)**

The 280x ADC supports offset correction via a 9-bit value that it adds or subtracts before the results are available in the ADC result registers. Timing for results is not affected. The default value is `0`.

**VREFHIVREFLO**

When you disable the **Use external reference 2.048V** or **External reference** option, the ADC logic uses a fixed 0-volt to 3.3-volt input range and the software disables **VREFHI** and **VREFLO**. To interpret the ADC input as a ratiometric signal, select the **External reference** option. Then set values for the high voltage reference (**VREFHI**) and the low voltage reference (**VREFLO**). **VREFHI** uses the external ADCINA0 pin, and **VREFLO** uses the internal GND.

**INT pulse control**

Use this option to configure when the ADC sets ADCINTFLG ADCINTx relative to the SOC and EOC Pulses. Select `Late interrupt pulse` or `Early interrupt pulse`.

**SOC high priority**

Use this option to enable and configure **SOC high priority mode** . In `all in round robin mode`, the default selection, the ADC services each SOC interrupt in a numerical sequence.

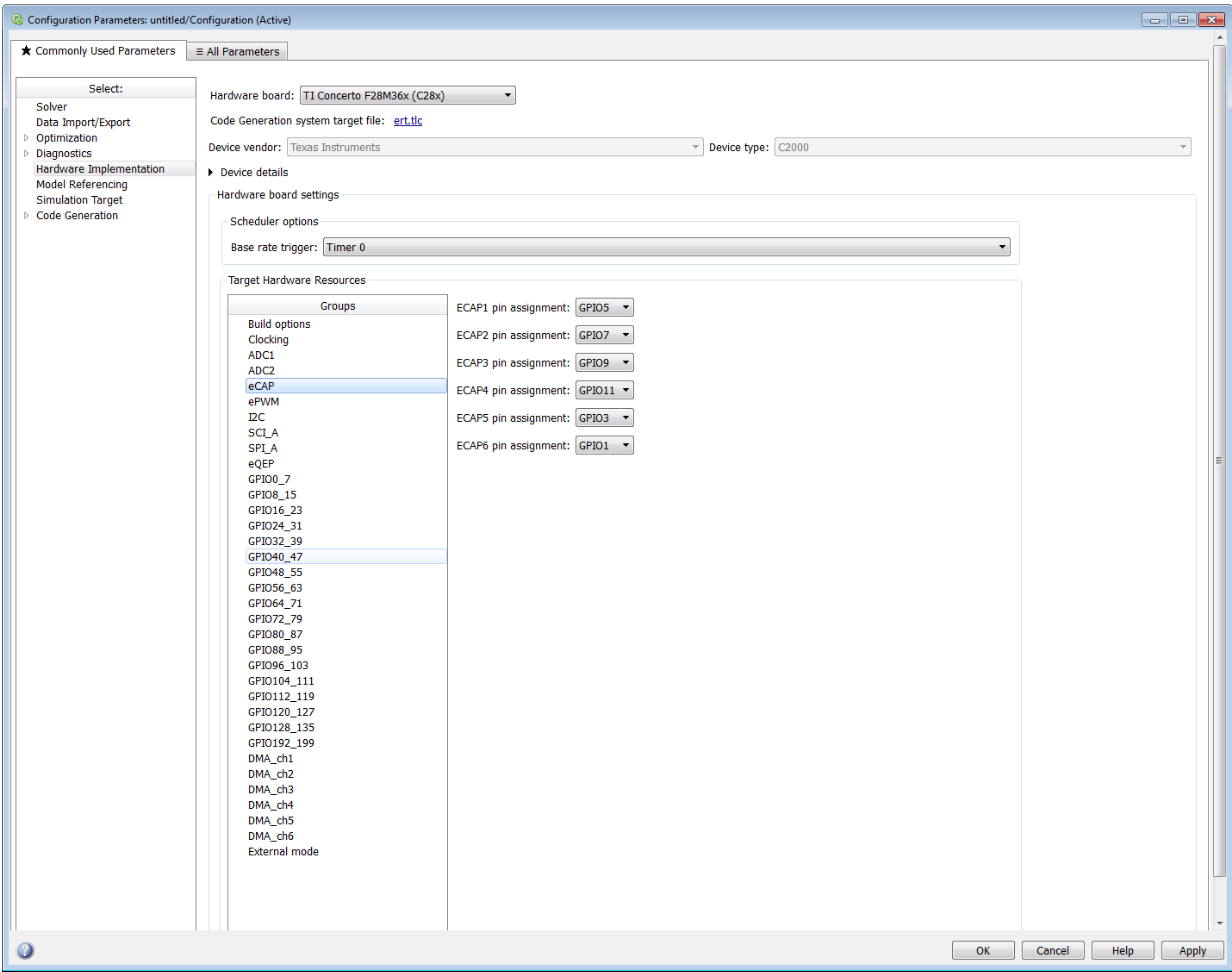
Choose one of the `high priority` selections to assign high priority to one or more of the SOCs. In this mode, the ADC operates in round robin mode until it receives a high priority SOC interrupt. The ADC finishes servicing the current SOC, services the high priority SOCs, and then returns to the next SOC in the round robin sequence.

For example, the ADC is servicing SOC8 when it receives a high priority interrupt on SOC1. The ADC completes servicing SOC8, services SOC1, and then services SOC9.

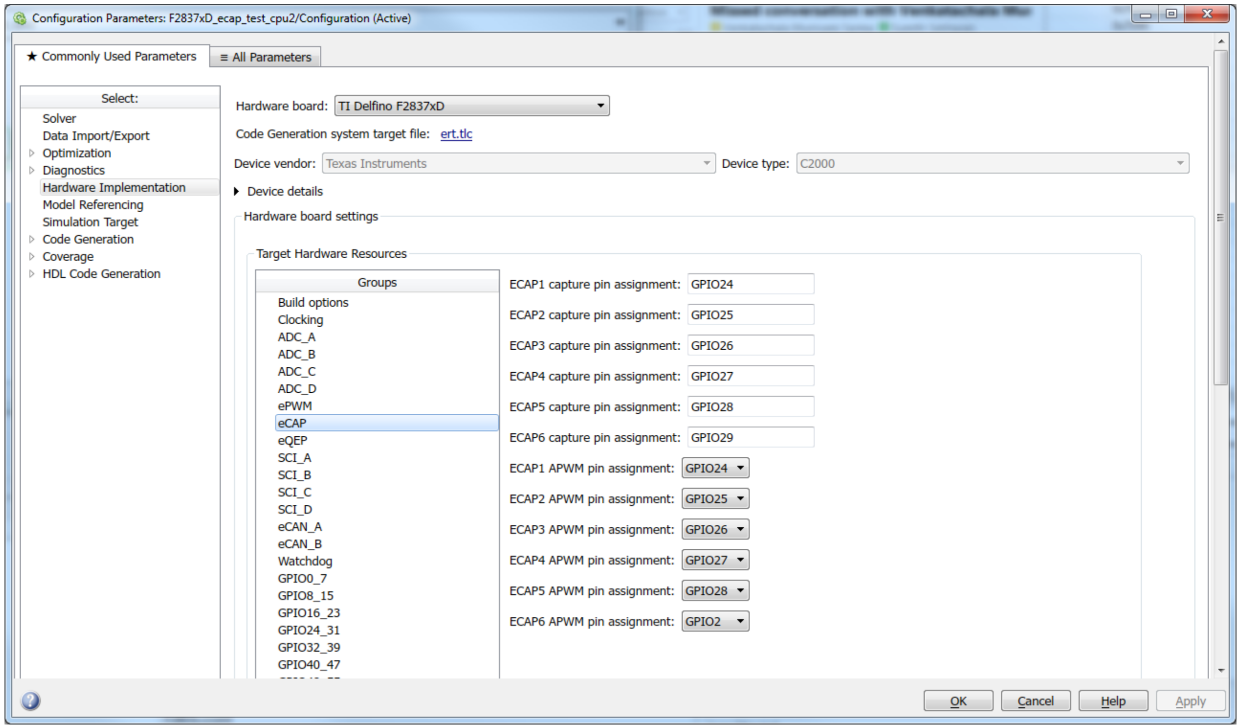
**XINT2SOC external pin**

Select the pin to which the ADC sends the XINT2SOC pulse.

## C28x-eCAP




The following screen shows the eCAP parameters for F2837xS, F2837xD and F2807x processors.




Assigns eCAP pins to GPIO pins.

**ECAP1 pin assignment**

Select a GPIO pin for ECAP1 module.

**ECAP2 pin assignment**

Select a GPIO pin for ECAP2 module.

**ECAP3 pin assignment**

Select a GPIO pin for ECAP3 module.

**ECAP4 pin assignment**

Select a GPIO pin for ECAP4 module.

**ECAP5 pin assignment**

Select a GPIO pin for ECAP5 module.

**ECAP6 pin assignment**

Select a GPIO pin for ECAP6 module.

The parameters described below are only for F2837xS, F2837xD, and F2807x processors.

**ECAP1 capture pin assignment**

Select GPIO pin for ECAP1 module when using in eCAP (capture) operating mode.

**ECAP2 capture pin assignment**

Select a GPIO pin for ECAP2 module.

**ECAP3 capture pin assignment**

Select GPIO pin for ECAP3 module when using in eCAP (capture) operating mode.

**ECAP4 capture pin assignment**

Select GPIO pin for ECAP4 module when using in eCAP (capture) operating mode.

**ECAP5 capture pin assignment**

Select GPIO pin for ECAP5 module when using in eCAP (capture) operating mode.

**ECAP6 capture pin assignment**

Select GPIO pin for ECAP6 module when using in eCAP (capture) operating mode.

**ECAP1 APWM pin assignment**

Select GPIO pin for ECAP1 module when using in APWM operating mode.

**ECAP2 APWM pin assignment**

Select GPIO pin for ECAP2 module when using in APWM operating mode.

**ECAP3 APWM pin assignment**

Select GPIO pin for ECAP3 module when using in APWM operating mode.

**ECAP4 APWM pin assignment**

Select GPIO pin for ECAP4 module when using in APWM operating mode.

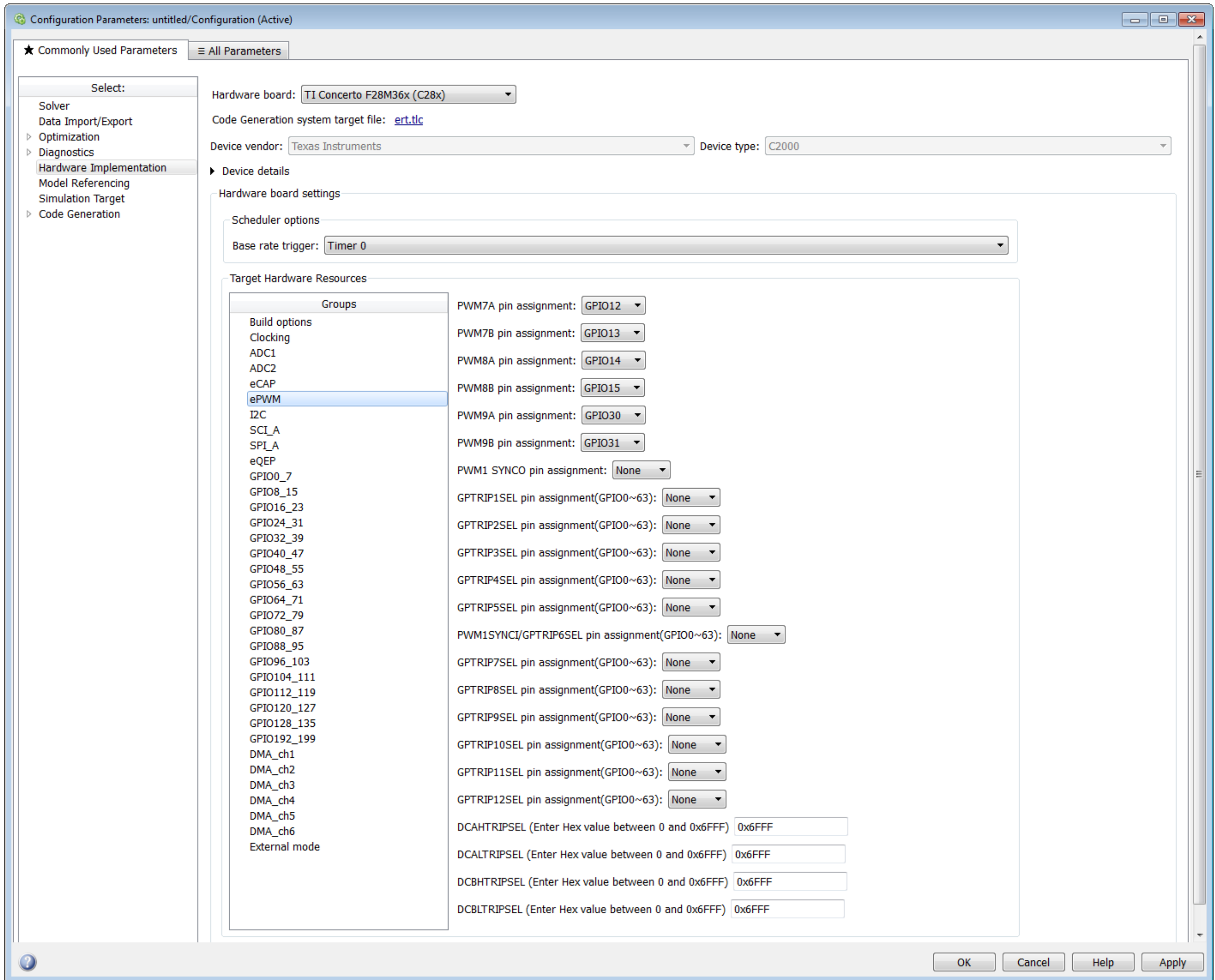
**ECAP5 APWM pin assignment**

Select GPIO pin for ECAP5 module when using in APWM operating mode.

**ECAP6 APWM pin assignment**

Select GPIO pin for ECAP6 module when using in APWM operating mode.

# C28x-ePWM




Assigns ePWM signals to GPIO pins.

### EPWM clock divider (EPWMCLKDIV)

This parameter appears only for F2807x and F2837x processors. This parameter allows you to select the ePWM clock divider.

**TZ1 pin assignment**

Assigns the trip-zone input 1 (TZ1) to a GPIO pin. Choices are `None` (the default), `GPIO12`, and `GPIO15`.

**TZ2 pin assignment**

Assigns the trip-zone input 2 (TZ2) to a GPIO pin. Choices are `None` (the default), `GPIO16`, and `GPIO28`.

**TZ3 pin assignment**

Assigns the trip-zone input 3 (TZ3) to a GPIO pin. Choices are `None` (the default), `GPIO17`, and `GPIO29`.

**TZ4 pin assignment**

Assigns the trip-zone input 4 (TZ4) to a GPIO pin. Choices are `None` (the default), `GPIO17`, and `GPIO28`.

**TZ5 pin assignment**

Assigns the trip-zone input 5 (TZ5) to a GPIO pin. Choices are `None` (the default), `GPIO16`, and `GPIO28`.

**TZ6 pin assignment**

Assigns the trip-zone input 6 (TZ6) to a GPIO pin. Choices are `None` (the default), `GPIO17`, and `GPIO29`.

---

**Note** The TZ# pin assignments are available only for TI F280x processors.

---

**SYNCI pin assignment**

Assigns the ePWM external sync pulse input (SYNCI) to a GPIO pin. Choices are `None` (the default), `GPIO6`, and `GPIO32`.

**SYNCO pin assignment**

---

**Note** SYNCI and SYNCO pin assignments are available for TI F28044, TI F280x, TI Delfino F2833x, TI Delfino F2834x, TI Piccolo F2802x, TI Piccolo F2803x, TI Piccolo F2806 processors.

---

Assigns the ePWM external sync pulse output (SYNCO) to a GPIO pin. Choices are `None` (the default), `GPIO6`, and `GPIO33`.

**PWM#A, PWM#B, PWM#C pin assignment**

The PWM # A, PWM # B, PWM # C pin assignment.

**GPTRIP#SEL pin assignment(GPIO0~63)**

Assigns the ePWM trip-zone input to a GPIO pin.

---

**Note** The GPTRIP#SEL pin assignment is available only for TI Concerto F28M35x/F28M36x processors.

---

**PWM1SYNCI/ GPTRIP6SEL pin assignment**

Assigns the ePWM sync pulse input (SYNCI) to a GPIO pin.

---

**Note** The PWM1SYNCI/GPTRIP#SEL pin assignments are available only for TI Concerto F28M35x/F28M36x processors.

---

**DCAHTRIPSEL (Enter Hex value between 0 and 0x6FFF) / DCBHTRIPSEL (Enter Hex value between 0 and 0x6FFF)**

Assigns the Digital Compare A High Trip Input to a GPIO pin.

---

**Note** DCAHTRIPSel pin assignment is available only for TI Concerto F28M35x/F28M36x processors.

---

**DCALTRIPSEL (Enter Hex value between 0 and 0x6FFF) / DCBLTRIPSEL (Enter Hex value between 0 and 0x6FFF)**

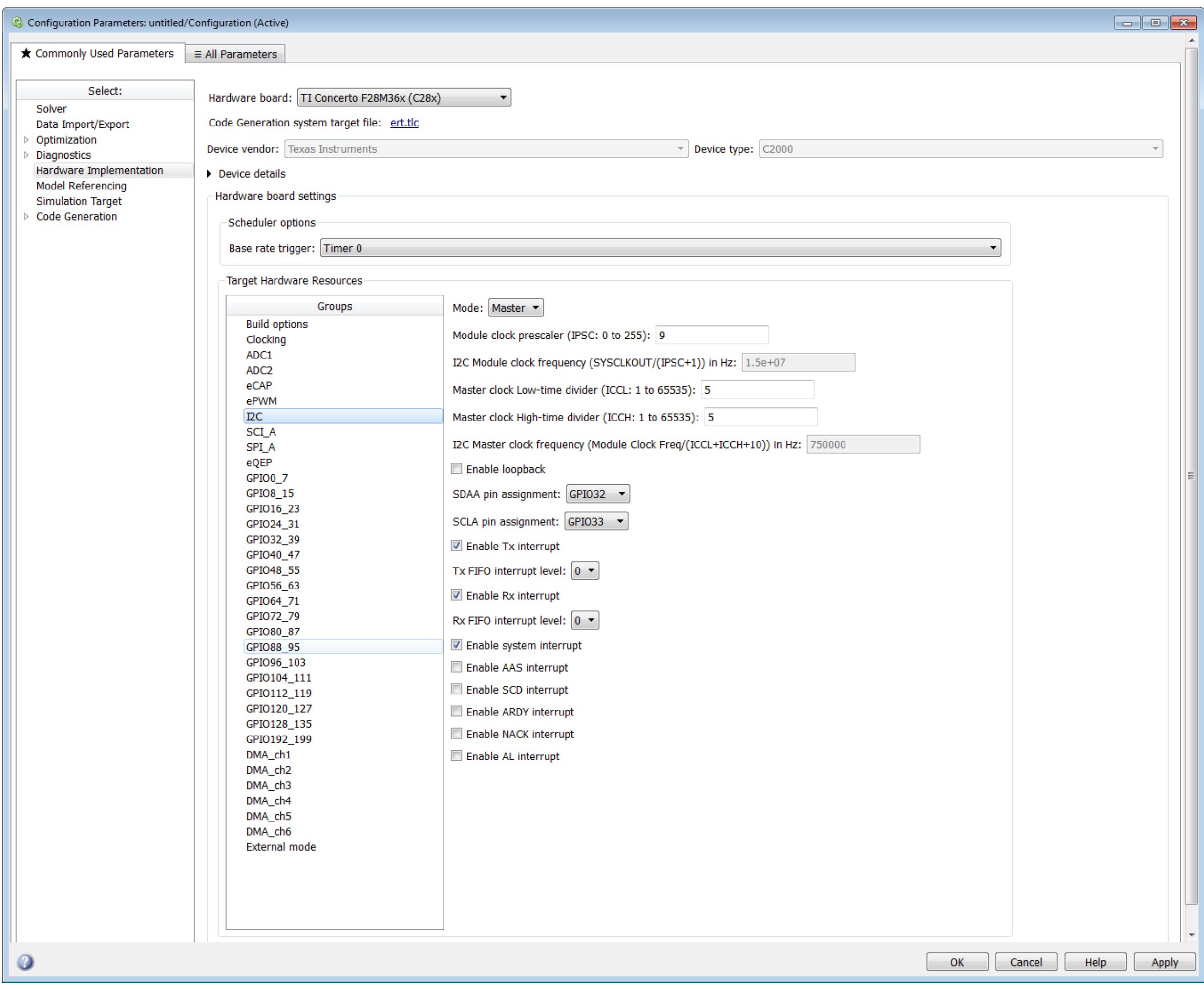
Assigns the Digital Compare A High Trip Input to a GPIO pin.

---

**Note** The DCALTRIPSEL pin assignment is available only for TI Concerto F28M35x/F28M36x processors.

---

## C28x-I2C




Report or set Inter-Integrated Circuit parameters. For more information, consult the *TMS320x280x/ TMS320F28M35x/ TMS320F28M36x Inter-Integrated Circuit Module Reference Guide*, Literature Number: SPRU721A/ SPRUH22F/ SPRUHE8B available on the Texas Instruments Web site.

### Mode

Configure the I2C module as `Master` or `Slave`.

If a module is an I2C master, it:

Initiates communication with slave nodes by sending the slave address and requesting data transfer to or from the slave.

Outputs the **Master clock frequency** on the serial clock line (SCL) line.

If a module is an I2C slave, it:

- Synchronizes itself with the serial clock line (SCL) line.
- Responds to communication requests from the master.

When **Mode** is `Slave`, you can configure the **Addressing format**, **Address register**, and **Bit count** parameters.

The **Mode** parameter corresponds to bit 10 (MST) of the I2C Mode Register (I2CMDR).

### Addressing format

If **Mode** is `Slave`, determine the addressing format of the I2C master, and set the I2C module to the same mode:

- `7-Bit Addressing`, the normal address mode.
- `10-Bit Addressing`, the expanded address mode.
- `Free Data Format`, a mode that does not use addresses. (If you **Enable loopback**, the `Free data format` is not supported.)

The **Addressing format** parameter corresponds to bit 3 (FDF) and bit 8 (XA) of the I2C Mode Register (I2CMDR).

### Own address register

If **Mode** is `Slave`, enter the 7-bit (0–127) or 10-bit (0–1023) address this I2C module uses while it is a slave.

This parameter corresponds to bits 9–0 (OAR) of the I2C Own Address Register (I2COAR).

**Bit count**

If **Mode** is `Slave`, set the number of bits in each data *byte* the I2C module transmits and receives. This value must match that of the I2C master.

This parameter corresponds to bits 2–0 (BC) of the I2C Mode Register (I2CMDR).

**Module clock prescaler (IPSC: 0 to 255):**

If **Mode** is `Master`, configure the module clock frequency by entering a value 0–255.

*Module clock frequency = I2C input clock frequency* / (*Module clock prescaler* + 1)

The I2C specifications require a module clock frequency between 7 MHz and 12 MHz.

The *I2C input clock frequency* depends on the DSP input clock frequency and the value of the PLL Control Register divider (PLLCR). For more information on setting the PLLCR, consult the documentation for your specific Digital Signal Controller.

This **Module clock prescaler (IPSC: 0 to 255):** corresponds to bits 7–0 (IPSC) of the I2C Prescaler Register (I2CPSC).

**I2C Module clock frequency (SYSCLKOUT / (IPSC+1)) in Hz:**

This field displays the frequency the I2C module uses internally. To set this value, change the **Module clock prescaler**.

For more information about this value, consult the “Formula for the Master Clock Period” section in the *TMS320x280x Inter-Integrated Circuit Module Reference Guide*, Literature Number: SPRU721, on the Texas Instruments Web site.

**I2C Master clock frequency (Module Clock Freq/(ICCL+ICCH+10)) in Hz:**

This field displays the master clock frequency.

For more information about this value, consult the “Clock Generation” section in the *TMS320x280x/ TMS320F28M35x/ TMS320F28M36x Inter-Integrated Circuit Module Reference Guide*, Literature Number: SPRU721/ SPRUH22F/ SPRUHE8B, available on the Texas Instruments Web site.

**Master clock Low-time divider (ICCL: 1 to 65535):**

When **Mode** is `Master`, this divider determines the duration of the low state of the SCL line on the I2C-bus.

The low-time duration of the master clock = Tmod x (ICCL + d).

For more information, consult the “Formula for the Master Clock Period” section in the *TMS320x280x/ TMS320F28M35x/ TMS320F28M36x Inter-Integrated Circuit*

*Module Reference Guide*, Literature Number: SPRU721A/ SPRUH22F/ SPRUHE8B, available on the Texas Instruments Web site.

This parameter corresponds to bits 15–0 (ICCL) of the Clock Low-Time Divider Register (I2CCLKL).

**Master clock High-time divider (ICCH: 1 to 65535):**

When **Mode** is `Master`, this divider determines the duration of the high state on the serial clock pin (SCL) of the I2C-bus.

The high-time duration of the master clock = Tmod x (ICCL + d).

For more information about this value, consult the “Formula for the Master Clock Period” section in the *TMS320x280x/ TMS320F28M35x/ TMS320F28M36x Inter-Integrated Circuit Module Reference Guide*, Literature Number: SPRU721A, SPRUH22f, SPRUHE8B, available on the Texas Instruments Web site.

This parameter corresponds to bits 15–0 (ICCH) of the Clock High-Time Divider Register (I2CCLKH).

**Enable loopback**

When **Mode** is `Master`, enable or disable digital loopback mode. In digital loopback mode, I2CDXR transmits data over an internal path to I2CDRR, which receives the data after a configurable delay.

The delay, measured in DSP cycles, equals (I2C input clock frequency/module clock frequency) x 8.

While **Enable loopback** is enabled, free data format addressing is not supported.

This parameter corresponds to bit 6 (DLB) of the I2C Mode Register (I2CMDR).

**SDAA pin assignment**

Select a GPIO pin as I2C data bidirectional port.

This parameter is available only for TI Concerto F28M36x (C28x) processors.

**SCLA pin assignment**

Select the GPIO pin as I2C clock bidirectional port.

This parameter is available only for TI Concerto F28M36x (C28x) processors.

**Enable Tx interrupt**

This parameter corresponds to bit 5 (TXFFIENA) of the I2C Transmit FIFO Register (I2CFFTX).

**Tx FIFO interrupt level**

This parameter corresponds to bits 4–0 (TXFFIL4-0) of the I2C Transmit FIFO Register (I2CFFTX).

**Enable Rx interrupt**

This parameter corresponds to bit 5 (RXFFIENA) of the I2C Receive FIFO Register (I2CFFRX).

**Rx FIFO interrupt level**

This parameter corresponds to bit 4–0 (RXFFIL4-0) of the I2C Receive FIFO Register (I2CFFRX).

**Enable system interrupt**

Select this parameter to display and individually configure the following five Basic I2C Interrupt Request parameters in the Interrupt Enable Register (I2CIER):

- Enable AAS interrupt
- Enable SCD interrupt
- Enable ARDY interrupt
- Enable NACK interrupt
- Enable AL interrupt

**Enable AAS interrupt**

Enable the addressed-as-slave interrupt.

When enabled, the I2C module generates an interrupt (AAS bit = 1) upon receiving one of the following:

- Its **Own address register**
- A general call (all zeros)
- A data byte is in free data format

When enabled, the I2C module clears the interrupt (AAS = 0) upon receiving one of the following:

- Multiple START conditions (7-bit addressing mode only)

- A slave address that is different from **Own address register** (10-bit addressing mode only)
- A NACK or a STOP condition

This parameter corresponds to bit 6 (AAS) of the Interrupt Enable Register (I2CIER).

**Enable SCD interrupt**

Enable stop condition detected interrupt.

When enabled, the I2C module generates an interrupt (SCD bit = 1) when the CPU detects a stop condition on the I2C bus.

When enabled, the I2C module clears the interrupt (SCD = 0) upon one of the following events:

- The CPU reads the I2CISRC while it indicates a stop condition
- A reset of the I2C module
- Someone manually clears the interrupt

This parameter corresponds to bit 5 (SCD) of the Interrupt Enable Register (I2CIER).

**Enable ARDY interrupt**

Enable register-access-ready interrupt enable bit.

When enabled, the I2C module generates an interrupt (ARDY bit = 1) when the previous address, data, and command values in the I2C module registers have been used and new values can be written to the I2C module registers.

This parameter corresponds to bit 2 (ARDY) of the Interrupt Enable Register (I2CIER).

**Enable NACK interrupt**

Enable no acknowledgment interrupt enable bit.

When enabled, the I2C module generates an interrupt (NACK bit = 1) when the module is a transmitter in master or slave mode and it receives a NACK condition.

This parameter corresponds to bit 1 (NACK) of the Interrupt Enable Register (I2CIER).

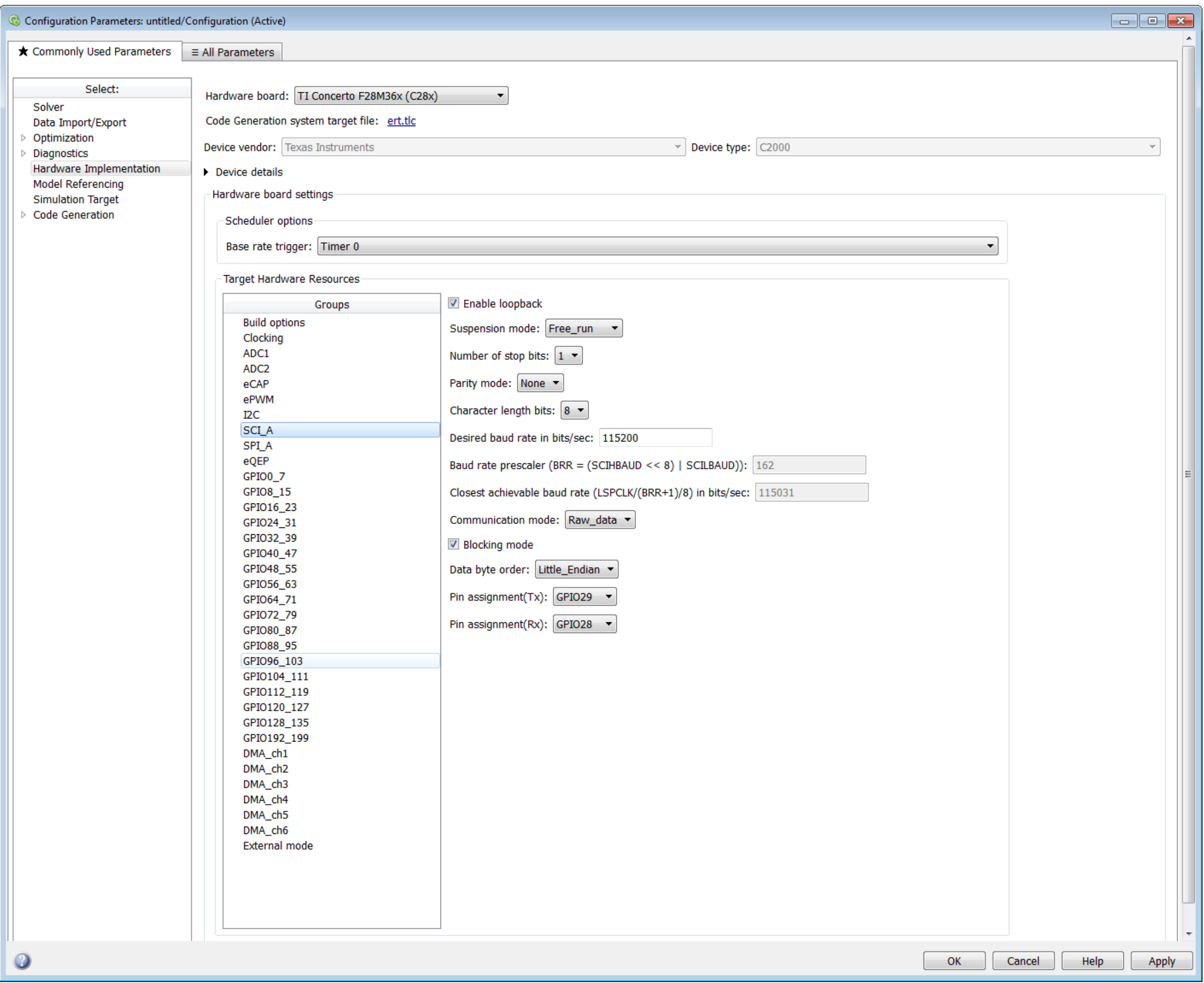
**Enable AL interrupt**

Enable arbitration-lost interrupt.

When enabled, the I2C module generates an interrupt (AL bit = 1) when the I2C module is operating as a master transmitter and looses an arbitration contest with another master transmitter.

This parameter corresponds to bit 0 (AL) of the Interrupt Enable Register (I2CIER).

## C28x-SCI_A, C28x-SCI_B, C28x-SCI_C



The serial communications interface parameters you can set for module A. These parameters are:

**Enable loopback**

Select this parameter to enable the loopback function for self-test and diagnostic purposes only. When this function is enabled, a C28x DSP Tx pin is internally connected to its Rx pin and can transmit data from its output port to its input port to check the integrity of the transmission.

**Baud rate**

Baud rate for transmitting and receiving data. Select from `115200` (the default), `57600`, `38400`, `19200`, `9600`, `4800`, `2400`, `1200`, `300`, and `110`.

**Suspension mode**

Type of suspension to use when debugging your program with Code Composer Studio. When your program encounters a breakpoint, the suspension mode determines whether to perform the program instruction. Available options are `Hard_abort`, `Soft_abort`, and `Free_run`. `Hard_abort` stops the program immediately. `Soft_abort` stops when the current receive/transmit sequence is complete. `Free_run` continues running regardless of the breakpoint.

**Number of stop bits**

Select whether to use `1` or `2` stop bits.

**Parity mode**

Type of parity to use. Available selections are `None`, `Odd` parity, or `Even` parity. `None` disables parity. `Odd` sets the parity bit to one if you have an odd number of ones in your bytes, such as 00110010. `Even` sets the parity bit to one if you have an even number of ones in your bytes, such as 00110011.

**Character length bits**

Length in bits of each transmitted or received character, set to 8 bits.

**Desired baud rate in bits/sec**

The desired baud rate specified by the user.

**Baud rate prescaler (BRR = (SCIHBAUD << 8) | SCILBAUD))**

The baud rate prescaler.

**Closest achievable baud rate (LSPCLK/(BRR+1)/8) in bits/sec**

The closest achievable baud rate calculated based on LSPCLK and BRR.

**Communication mode**

Select `Raw_data` or `Protocol` mode. Raw data is unformatted and sent whenever the transmitting side is ready to send, whether the receiving side is ready or not. Without a wait state, deadlock conditions do not occur. Data transmission is asynchronous. With this mode, it is possible the receiving side could miss data, but if the data is noncritical, using raw data mode can avoid blocking processes.

When you select protocol mode, some handshaking between host and processor occurs. The transmitting side sends $SND to indicate it is ready to transmit. The receiving side sends back $RDY to indicate it is ready to receive. The transmitting side then sends data and, when the transmission is completed, it sends a checksum.

Advantages to using protocol mode include:

- Avoids deadlock
- Determines whether data is received without errors (checksum)
- Determines whether data is received by processor
- Determines time consistency; each side waits for its turn to send or receive

---

**Note** Deadlocks can occur if one SCI Transmit block tries to communicate with more than one SCI Receive block on different COM ports when both are blocking (using protocol mode). Deadlocks cannot occur on the same COM port.

---

**Blocking mode**

If this option is set to True, system waits until data is available to read (when data length is reached). If this option is set to False, system checks FIFO periodically (in polling mode) for data to read. If data is present, the block reads and outputs the contents. When data is not present, the block outputs the last value and continues.

**Data byte order**

Select `Little Endian` or `Big Endian`, to match the endianness of the data being moved.

**Pin assignment (Tx)**

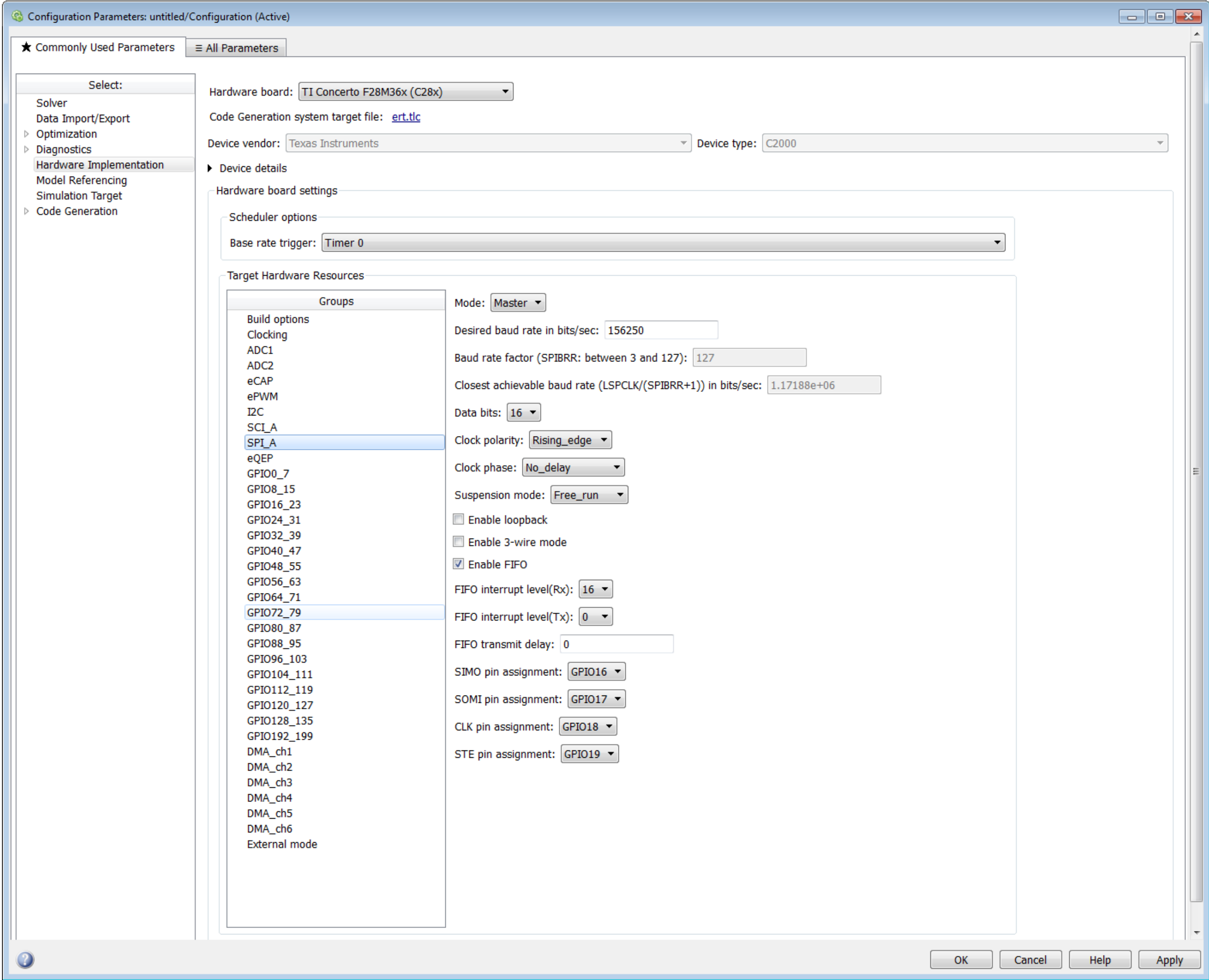
Assigns the SCI transmit pin to use with the SCI module.

**Pin assignment (Rx)**

Assigns the SCI receive pin to use with the SCI module.

**Note** The SCI_B and SCI_C are available only for TI F280x processors.

## C28x-SPI_A, C28x-SPI_B, C28x-SPI_C, C28x-SPI_D




The serial peripheral interface parameters you can set for the A module. These parameters are:

**Mode**

Set to `Master` or `Slave`.

**Desired baud rate in bits/sec**

The desired baud rate specified by the user.

**Baud rate factor (SPIBRR: between 3 and 127)**

To set the **Baud rate factor**, search for “Baud Rate Determination” and “SPI Baud Rate Register (SPIBRR) Bit Descriptions” in *TMS320x28xx, 28xxx DSP Serial Peripheral Interface (SPI) Reference Guide*, Literature Number: SPRU059, available on the Texas Instruments Web Site.

**Closest achievable baud rate (LSPCLK/(SPIBRR+1)) in bits/sec**

The closest achievable baud rate calculated based on LSPCLK and SPIBRR.

**Data bits**

Length in bits from `1` to `16` of each transmitted or received character. For example, if you select `8`, the maximum data that can be transmitted using SPI is $2^{8-1}$. If you send data greater than this value, the buffer overflows.

**Clock polarity**

Select `Rising_edge` or `Falling_edge`.

**Clock phase**

Select `No_delay` or `Delay_half_cycle`.

**Suspension mode**

Type of suspension to use when debugging your program with Code Composer Studio. When your program encounters a breakpoint, the selected suspension mode determines whether to perform the program instruction. Available options are `Hard_abort`, `Soft_abort`, and `Free_run`. `Hard_abort` stops the program immediately. `Soft_abort` stops when the current receive or transmit sequence is complete. `Free_run` continues running regardless of the breakpoint.

**Enable loopback**

Select this option to enable the loopback function for self-test and diagnostic purposes only. When this function is enabled, the Tx pin on a C28x DSP is internally connected to its Rx pin and can transmit data from its output port to its input port to check the integrity of the transmission.

**Enable 3-wire mode**

Enable SPI communication over three pins instead of the normal four pins.

**Enable FIFO**

Set true or false.

**FIFO interrupt level (Rx)**

Set level for receive FIFO interrupt.

**FIFO interrupt level (Tx)**

Set level for transmit FIFO interrupt.

**FIFO transmit delay**

Enter FIFO transmit delay (in processor clock cycles) to pause between data transmissions. Enter an integer.

**CLK pin assignment**

Assigns the SPI something (CLK) to a GPIO pin. Choices are `None` (default), `GPIO14`, or `GPIO26`.

CLK pin assignment is not available for TI Concerto F28M35x/F28M36x processors.

**SOMI pin assignment**

Assigns the SPI value (SOMI) to a GPIO pin. Choices are `None` (default), `GPIO13`, or `GPIO25`.

**STE pin assignment**

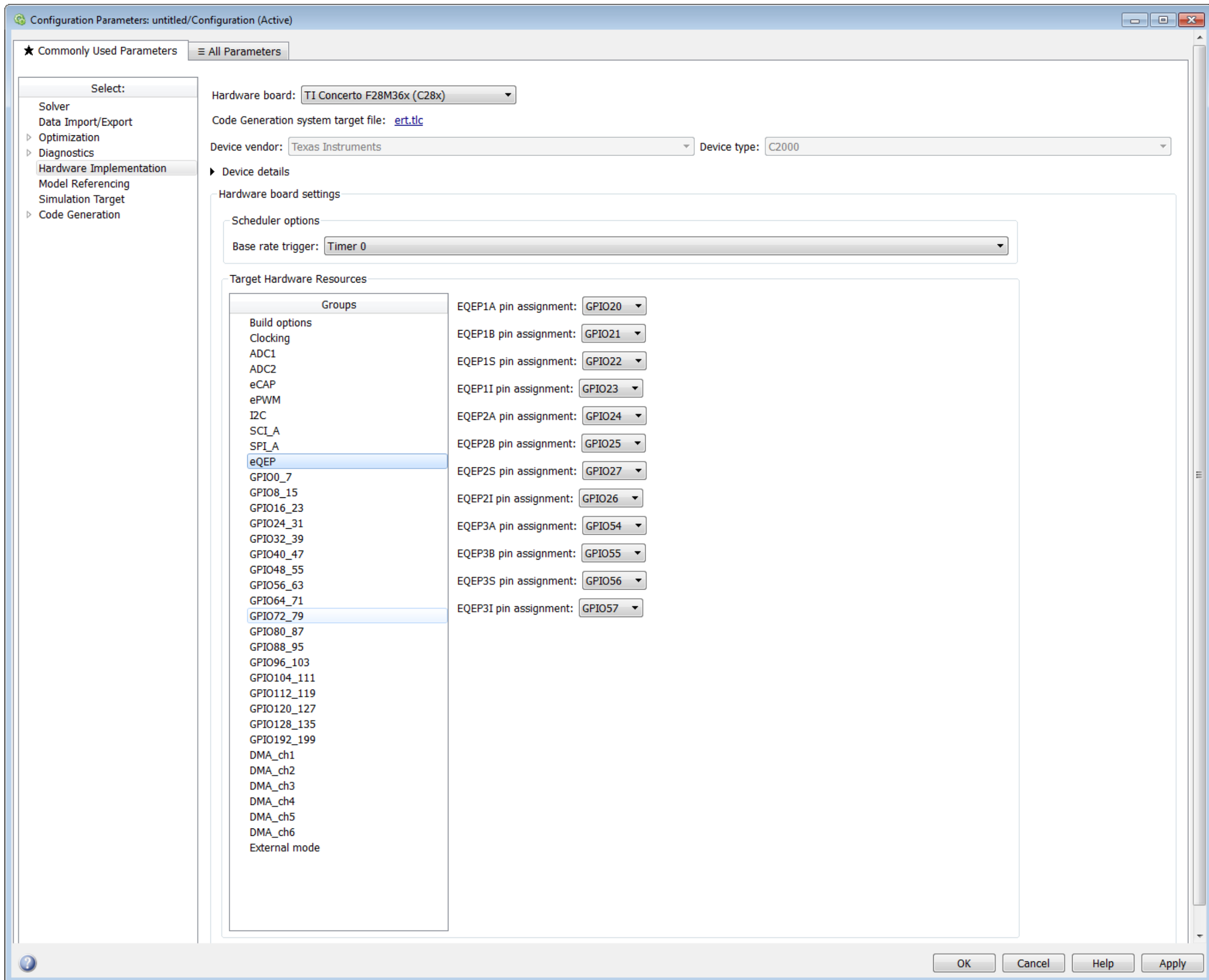
Assigns the SPI value (STE) to a GPIO pin. Choices are `None` (default), `GPIO15`, or `GPIO27`.

STE pin assignment is not available for TI Concerto F28M35x/ F28M36x processors.

**SIMO pin assignment**

Assigns the SPI something (SIMO) to a GPIO pin. Choices are `None` (default), `GPIO12`, or `GPIO24`.

# C28x-eQEP




Assigns eQEP pins to GPIO pins.

### EQEP1A pin assignment

Select an option from the list—`None`, `GPIO20`, `GPIO50`, `GPIO64`, or `GPIO106`.

**EQEP1B pin assignment**

Select an option from the list—None, GPIO21, GPIO51, GPIO65, or GPIO107.

**EQEP1S pin assignment**

Select an option from the list—None, GPIO22, GPIO52, GPIO66, or GPIO108.

**EQEP1I pin assignment**

Select an option from the list—None, GPIO23, GPIO53, GPIO67, or GPIO109.

**EQEP2A pin assignment**

Select an option from the list—None, GPIO24, GPIO54, GPIO66, or GPIO110. The pin numbers shown in the list vary based on the processor selected.

**EQEP2B pin assignment**

Select an option from the list—None, GPIO25, GPIO55, GPIO67, or GPIO111. The pin numbers shown in the list vary based on the processors selected.

**EQEP2S pin assignment**

Select an option from the list—None, GPIO27, GPIO31, GPIO57, GPIO67, or GPIO113.

**EQEP2I pin assignment**

Select an option from the list—None, GPIO26, GPIO30, GPIO56, GPIO64, or GPIO112.

**EQEP3A pin assignment**

Select an option from the list—None, GPIO54, or GPIO112. This parameter is available only for TI Concerto F28M36x (C28x) processors.

**EQEP3B pin assignment**

Select an option from the list—None, GPIO55, or GPIO113. This parameter is available only for TI Concerto F28M36x (C28x) processors.

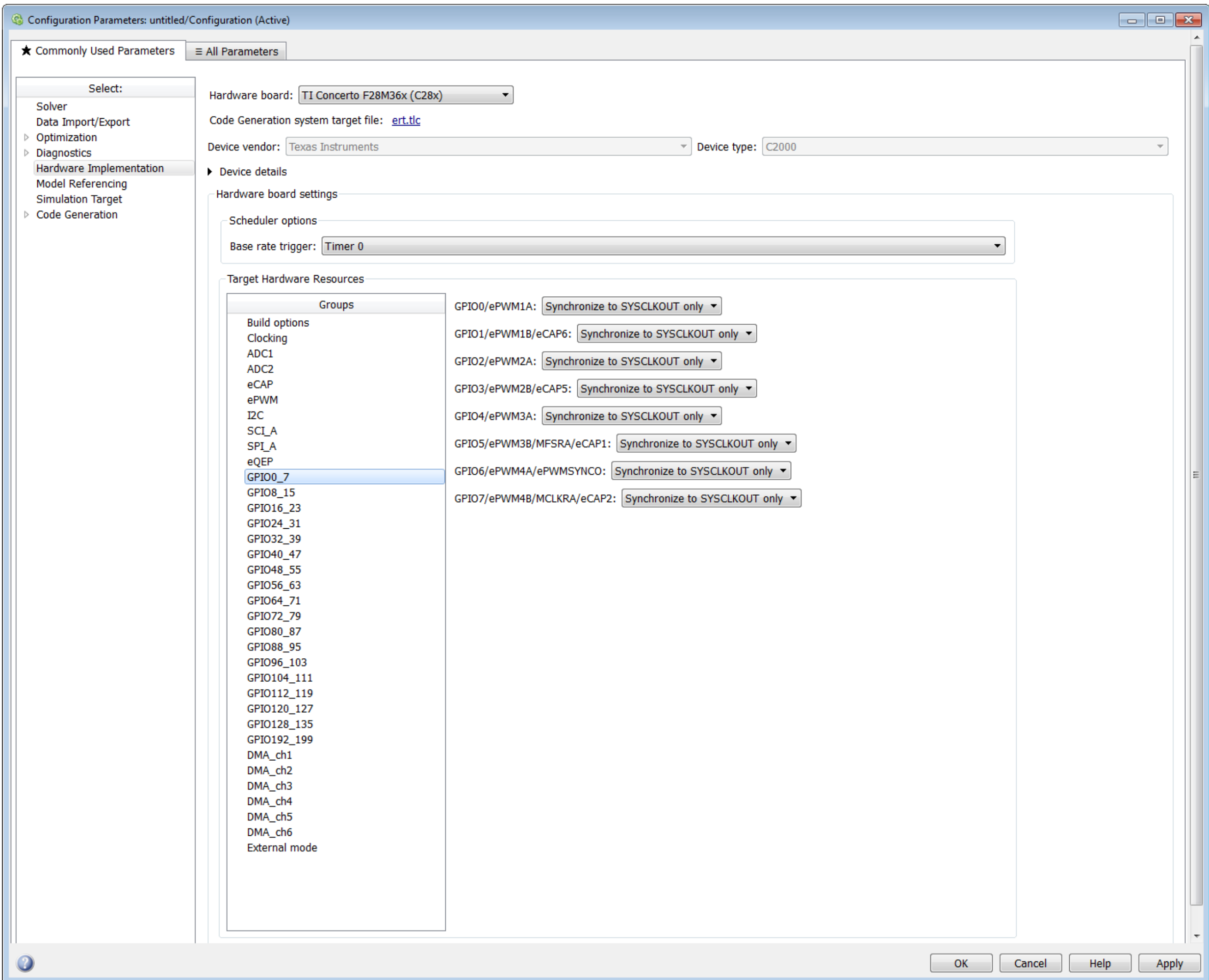
**EQEP3S pin assignment**

Select an option from the list—None, GPIO56, or GPI110.

**EQEP3I pin assignment**

Select an option from the list—None, GPIO57, or GPIO111.

## C28x-GPIO



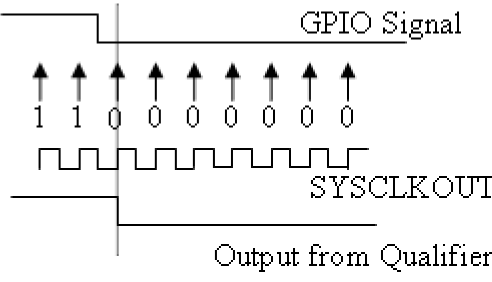

**GPIO**Use the GPIO pins for digital input or output by connecting to one of the three peripheral I/O ports.

The range of GPIO pins for different processors is given below:

| Processors | GPIO Pin Values |
|---|---|
| C281x | GPIOA, GPIOB, GPIOD, GPIOE, GPIOF, and GPIOG |
| F2803x | GPIO0–7, GPIO8–15, GPIO16–23, GPIO24–31, GPIO32–39, GPIO40–44 |
| F2806x | GPIO0–7, GPIO8–15, GPIO16–23, GPIO24–31, GPIO32–39, GPIO40–44, GPIO50–55, GPIO56–58 |
| F2823x, F2833x, and C2834x | GPIO0–7, GPIO8–15, GPIO16–23, GPIO24–31, GPIO32–39, GPIO40–47, GPIO48–55, GPIO56–63 |
| C2801x, F2802x, F28044, F280x | GPIO0–7, GPIO8–15, GPIO16–23, GPIO24–31, GPIO32–34 |
| F28M35x (C28x) | GPIO0–7, GPIO8–15, GPIO16–23, GPIO24–31, GPIO32–39, GPIO40–47, GPIO48_55, GPIO56–63, GPIO68–71, GPIO128–135 |
| F28M36x (C28x) | GPIO0–7, GPIO8–15, GPIO16–23, GPIO24–31, GPIO40–47, GPIO48–55, GPIO56–63, GPIO64–71, GPIO72–79, GPIO80–87, GPIO88–95, GPIO96–103, GPIO104–111, GPIO112–119, GPIO120–127, GPIO128–135, GPIO192–199. |

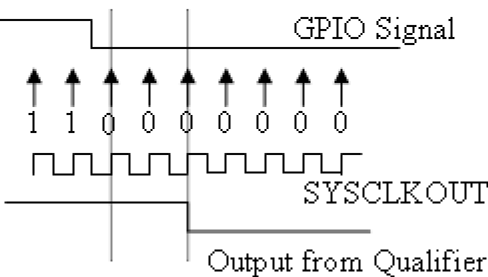
Each pin selected for input offers four signal qualification types:

- `Sync to SYSCLKOUT only` — This setting is the default for all pins at reset. Using this qualification type, the input signal is synchronized to the system clock SYSCLKOUT. The following figure shows the input signal measured on each tick of the system clock, and the resulting output from the qualifier.

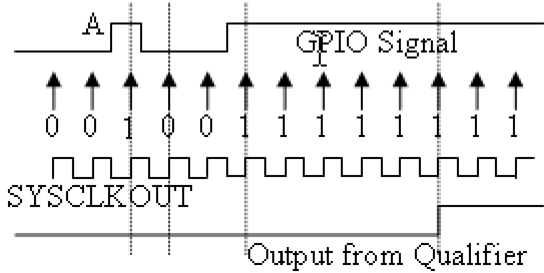



- `Qualification using 3 samples` — This setting requires three consecutive cycles of the same value for the output value to change. The following figure shows that, in the third cycle, the GPIO value changes to `0`, but the qualifier output is still `1`

because it waits for three consecutive cycles of the same GPIO value. The next three cycles all have a value of `0`, and the output from the qualifier changes to `0` immediately after the third consecutive value is received.



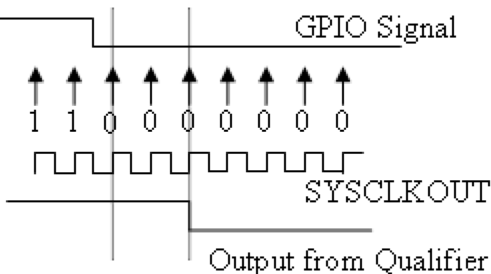

- `Qualification using 6 samples` — This setting requires six consecutive cycles of the same GPIO input value for the output from the qualifier to change. In the following figure, the glitch **A** does not alter the output signal. When the glitch occurs, the counting begins, but the next measurement is low again, so the count is ignored. The output signal does not change until six consecutive samples of the high signal are measured.



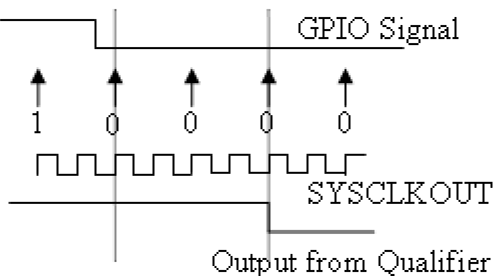

**Qualification sampling period prescaler**

Visible only when a setting for **Qualification type for GPIO [pin#]** is selected. The qualification sampling period prescaler, with possible values of `0` to `255`, calculates the frequency of the qualification samples or the number of system clock ticks per sample. The formula for calculating the qualification sampling frequency is SYSCLKOUT/(2 * Prescaler), except for zero. When **Qualification sampling period prescaler=**`0`, a sample is taken every SYSCLKOUT clock tick. For example, a prescale setting of `0` means that a sample is taken on each SYSCLKOUT tick.

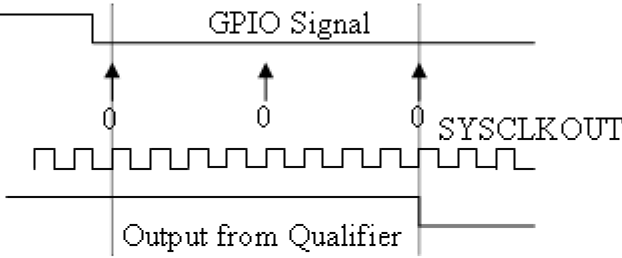
The following figure shows the SYSCLKOUT ticks, a sample taken every clock tick, and the **Qualification type** set to `Qualification using 3 samples`. In this case, the **Qualification sampling period prescaler**=`0`:




In the next figure **Qualification sampling period prescaler**=`1`. A sample is taken every two clock ticks, and the **Qualification type** is set to `Qualification using 3 samples`. The output signal changes much later than if **Qualification sampling period prescaler**=`0`.




In the following figure, **Qualification sampling period prescaler**=`2`. Thus , a sample is taken every four clock ticks, and the **Qualification type** is set to `Qualification using 3 samples`.




- `Asynchronous`

  Using this qualification type, the signal is synchronized to an asynchronous event initiated by software (CPU) via control register bits.

**Qualification sampling period**

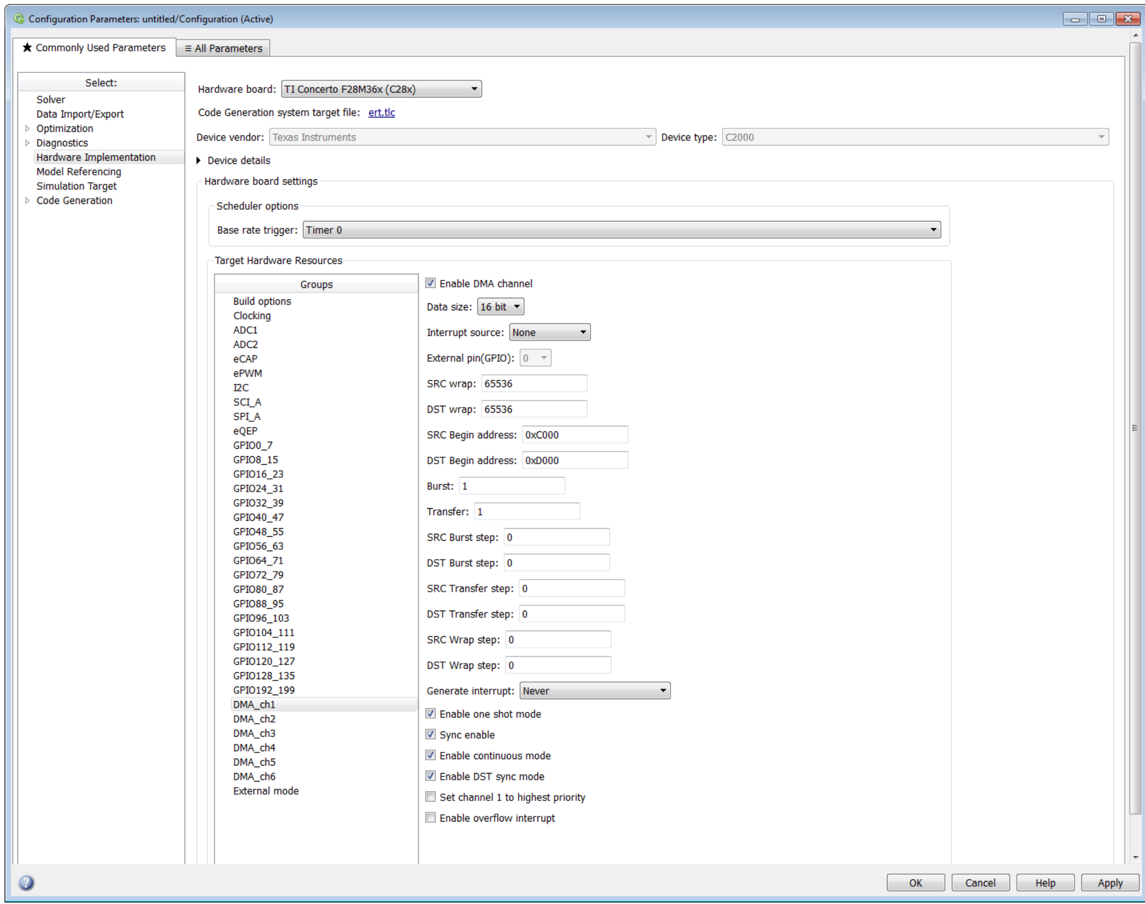
Enter the qualification sampling period.

**GPIOA, GPIOB, GPIOD, GPIOE input qualification sampling period**

**GPIO# Pull Up Disabled**

Select this check box to disable the GPIO pull up register. This option is available only for TI Concerto F28M35x/F28M36x processors.

## C28x-DMA_ch[#]




The Direct Memory Access module transfers data directly between peripherals and memory using a dedicated bus, increasing overall system performance.

You can individually enable and configure each DMA channel.

The DMA module services are event driven. Using the Interrupt source and **External pin (GPIO)** parameters, you can configure a wide range of peripheral interrupt event triggers.

For more information, consult the *TMS320x2833x, 2823x/ TMS320F28M35x/ TMS320F28M36x Direct Memory Access (DMA) Module Reference Guide*, Literature Number: SPRUFB8A/ SPRUH22F/ SPRUHE8B.

Also consult the *Increasing Data Throughput using the TMS320F2833x DSC DMA* training presentation (requires login), both available from the TI Web site.

**Enable DMA channel**

Enable this parameter to edit the configuration of a specific DMA channel.

This parameter does not have a corresponding bit or register.

**Data size**

Select the size of the data bit transfer: `16 bit` or `32 bit`.

The DMA read/write data buses are 32 bits wide. 32-bit transfers have twice the data throughput of a 16-bit transfer.

When providing DMA service to McBSP, set **Data size** to `16 bit`.

The following parameters are based on a 16-bit word size. If you set **Data size** to `32 bit`, double the value of the following parameters:

- Size: Burst
- Source: Burst step
- Source: Transfer step
- Source: Wrap step
- Destination: Burst step
- Destination: Transfer step
- Destination: Wrap step

**Data size** corresponds to bit 14 (DATASIZE) in the Mode Register (MODE).

**Interrupt source**

Select the peripheral interrupt that triggers a DMA burst for the specified channel.

Select SEQ1INT or SEQ2INT to configure the ADC interrupt as interrupt source.

Select XINT1, XINT2, or XINT13 to configure GPIO pin 0 to 31 as an external interrupt source. Select XINT3 to XINT7 to configure GPIO pin 32 to 63 as an external interrupt source.

For more information about configuring XINT, consult the following references:

- *TMS320x2833x, 2823x External Interface (XINTF) User's Guide*, Literature Number: SPRU949, available on the TI Web site.
- *TMS320x2833x System Control and Interrupts*, Literature Number: SPRUFB0, available on the TI Web site.
- *TMS320F28M35x/ TMS320F28M36x*, Literature Number: SPRUH22F/ SPRUHE8B available on the TI Web site.
- The C280x/C2802x/C2803x/C2805x/C2806x/C2833x/C2834x/F28M3x/F2807x/F2837xD/F2837xS/F28004x GPIO Digital Input and C280x/C2802x/C2803x/C2805x/C2806x/C2833x/C2834x/F28M3x/F2807x/F2837xD/F2837xS/F28004x GPIO Digital Output block reference sections.

Drop-down menu items from TINT0 to MREVTB may require manual configuration.

Select ePWM1SOCA through ePWM6SOCB to configure the ePWM interrupt as an interrupt source. Note that not all revisions of the TMS320F2833x silicon provide ePWM interrupts as sources for DMA transfers. For more information about silicon revisions consult the following reference:

*TMS320x2833x, 2823x Silicon Errata/ TMS320F28M35x/ TMS320F28M36x*, Literature Number: SPRZ272/ SPRUH22F/ SPRUHE8B, available on the TI Web site.

The **Interrupt source** parameter corresponds to bit 4-0 (PERINTSEL) in the Mode Register (MODE).

**External pin(GPIO)**

When you set **Interrupt source** is set to an external interface (XINT[#]), specify the GPIO pin number from which the interrupt originates.

This parameter corresponds to the GPIO XINTn, XNMI Interrupt Select (GPIOXINTnSEL, GPIOXNMISEL) Registers.

For more information, consult the *TMS320x2833x / TMS320F28M35x/ TMS320F28M36x System Control and Interrupts Reference Guide*, Literature Number SPRUFB0/ SPRUH22F/ SPRUHE8B available from the TI Web site.

**SRC wrap**

Specify the number of bursts before returning the current source address pointer to the **Source Begin Address** value. To disable wrapping, enter a value for **SRC wrap** that is greater than the **Transfer** value.

This parameter corresponds to bits 15-0 (SRC_WRAP_SIZE) in the Source Wrap Size Register (SRC_WRAP_SIZE).

**DST wrap**

Specify the number of bursts before returning the current destination address pointer to the **Destination Begin Address** value. To disable wrapping, enter a value for **DST wrap** that is greater than the **Transfer** value.

This parameter corresponds to bits 15-0 (DST_WRAP_SIZE) in the Destination Wrap Size Register (DST_WRAP_SIZE).

**SRC Begin address**

Set the starting address for the current source address pointer. The DMA module points to this address at the beginning of a transfer and returns to it as specified by the **SRC wrap** parameter.

This parameter corresponds to bits 21-0 (BEGADDR) in the Active Source Begin Register (SRC_BEG_ADDR).

**DST Begin address**

Set the starting address for the current destination address pointer. The DMA module points to this address at the beginning of a transfer and returns to it as specified by the **DST wrap** parameter.

This parameter corresponds to bits 21-0 (BEGADDR) in the Active Destination Begin Register (DST_BEG_ADDR).

**Burst**

Specify the number of 16-bit words in a burst, from 1 to 32. The DMA module must complete a burst before it can service the next channel.

Set the **Burst** value for the peripheral the DMA module is servicing. For the ADC, the value equals the number of ADC registers used, up to 16. For multichannel buffered serial ports (McBSP), which lack FIFOs, the value is 1.

For RAM, the value can range from 1 to 32.

This parameter corresponds to bits 4-0 (BURSTSIZE) in the Burst Size Register (BURST_SIZE).

---

**Note** This parameter is based on a 16-bit word size. If you set **Data size** to `32 bit`, double the value of this parameter.

---

**Transfer**

Specify the number of bursts in a transfer, from 1 to 65536.

This parameter corresponds to bits 15-0 (TRANSFERSIZE) in the Transfer Size Register (TRANSFER_SIZE).

**SRC Burst step**

Set the number of 16-bit words by which to increment or decrement the current address pointer before the next burst. Enter a value from –4096 (decrement) to 4095 (increment).

To disable incrementing or decrementing the address pointer, set **Burst step** to 0. For example, because McBSP does not use FIFO, configure DMA to maintain the sequence of the McBSP data by moving each word of the data individually.

Accordingly, when you use DMA to transmit or receive McBSP data, set **Burst size** to 1 word and **Burst step** to 0.

This parameter corresponds to bits 15-0 (SRCBURSTSTEP) in the Source Burst Step Size Register (SRC_BURST_STEP).

---

**Note** This parameter is based on a 16-bit word size. If you set **Data size** to `32 bit`, double the value of this parameter.

---

**DST Burst step**

Set the number of 16-bit words by which to increment or decrement the current address pointer before the next burst. Enter a value from –4096 (decrement) to 4095 (increment).

To disable incrementing or decrementing the address pointer, set **Burst step** to 0. For example, because McBSP does not use FIFO, configure DMA to maintain the sequence of the McBSP data by moving each word of the data individually. Accordingly, when you use DMA to transmit or receive McBSP data, set **Burst size** to 1 word and **Burst step** to 0.

This parameter corresponds to bits 15-0 (DSTBURSTSTEP) in the Destination Burst Step Size Register (DST_BURST_STEP).

**Note** This parameter is based on a 16-bit word size. If you set **Data size** to `32 bit`, double the value of this parameter.

### SRC Transfer step

Set the number of 16-bit words by which to increment or decrement the current address pointer before the next transfer. Enter a value from –4096 (decrement) to 4095 (increment).

To disable incrementing or decrementing the address pointer, set **Transfer step** to 0.

This parameter corresponds to bits 15-0 (SRCTRANSFERSTEP) Source Transfer Step Size Register (SRC_TRANSFER_STEP).

If DMA is configured to perform memory wrapping (if **SRC wrap** is enabled) the corresponding source **Transfer step** does not alter the results.

**Note** This parameter is based on a 16-bit word size. If you set **Data size** to `32 bit`, double the value of this parameter.

### DST Transfer step

Set the number of 16-bit words by which to increment or decrement the current address pointer before the next transfer. Enter a value from –4096 (decrement) to 4095 (increment).

To disable incrementing or decrementing the address pointer, set **Transfer step** to 0.

This parameter corresponds to bits 15-0 (DSTTRANSFERSTEP) Destination Transfer Step Size Register (DST_TRANSFER_STEP).

If DMA is configured to perform memory wrapping (if **DST wrap** is enabled) the corresponding destination **Transfer step** does not alter the results.

**Note** This parameter is based on a 16-bit word size. If you set **Data size** to `32 bit`, double the value of this parameter.

### SRC Wrap step

Set the number of 16-bit words by which to increment or decrement the SRC_BEG_ADDR address pointer when a wrap event occurs. Enter a value from –4096 (decrement) to 4095 (increment).

This parameter corresponds to bits 15-0 (WRAPSTEP) in the Source Wrap Step Size Registers (SRC_WRAP_STEP).

**Note** This parameter is based on a 16-bit word size. If you set **Data size** to `32 bit`, double the value of this parameter.

### DST Wrap step

Set the number of 16-bit words by which to increment or decrement the DST_BEG_ADDR address pointer when a wrap event occurs. Enter a value from –4096 (decrement) to 4095 (increment).

This parameter corresponds to bits 15-0 (WRAPSTEP) in the Destination Wrap Step Size Registers (DST_WRAP_STEP).

**Note** This parameter is based on a 16-bit word size. If you set **Data size** to `32 bit`, double the value of this parameter.

### Generate interrupt

Enable this parameter to have the DMA channel send an interrupt to the CPU via the PIE at the beginning or end of a data transfer.

This parameter corresponds to bit 15 (CHINTE) and bit 9 (CHINTMODE) in the Mode Register (MODE).

### Enable one shot mode

Enable this parameter to have the DMA channel complete an entire *transfer* in response to an interrupt event trigger.

This option allows a single DMA channel and peripheral to dominate resources, and may streamline processing, but it also creates the potential for resource conflicts and delays.

Disable this parameter to have DMA complete one *burst* per channel per interrupt.

**Sync enable**

When **Interrupt source** is set to SEQ1INT, enable this parameter to reset the DMA wrap counter when it receives the ADCSYNC signal from SEQ1INT. This way, the wrap counter and the ADC channels remain synchronized with each other.

If **Interrupt source** is not set to SEQ1INT, **Sync enable** does not alter the results.

This parameter corresponds to bit 12 (SYNCE) of the Mode Register (MODE).

**Enable continuous mode**

Select this parameter to leave the DMA channel enabled upon completing a transfer. The channel will wait for the next interrupt event trigger.

Clear this parameter to disable the DMA channel upon completing a transfer. The DMA module disables the DMA channel by clearing the RUNSTS bit in the CONTROL register when it completes the transfer. To use the channel again, first reset the RUN bit in the CONTROL register.

**Enable DST sync mode**

When **Sync enable** is enabled, enabling this parameter resets the destination wrap counter (DST_WRAP_COUNT) when the DMA module receives the SEQ1INT interrupt/ADCSYNC signal.

Disabling this parameter resets the source wrap counter (SCR_WRAP_COUNT) when the DMA module receives the SEQ1INT interrupt/ADCSYNC signal.

This parameter is associated with bit 13 (SYNCSEL) in the Mode Register (MODE).

**Set channel 1 to highest priority**

This parameter is only available for DMA_ch1.

Enable this setting when DMA channel 1 is configured to handle high-bandwidth data, such as ADC data, and the other DMA channels are configured to handle lower-priority data.

When enabled, the DMA module services each enabled channel sequentially until it receives a trigger from channel 1.

Upon receiving the trigger, DMA interrupts its service to the current channel at the end of the current word, services the channel 1 burst that generated the trigger, and then continues servicing the current channel at the beginning of the next word.

Disable this channel to give each DMA channel equal priority, or if DMA channel 1 is the only enabled channel.

When disabled, the DMA module services each enabled channel sequentially.

This parameter corresponds to bit 0 (CH1PRIORITY) in the Priority Control Register 1 (PRIORITYCTRL1).

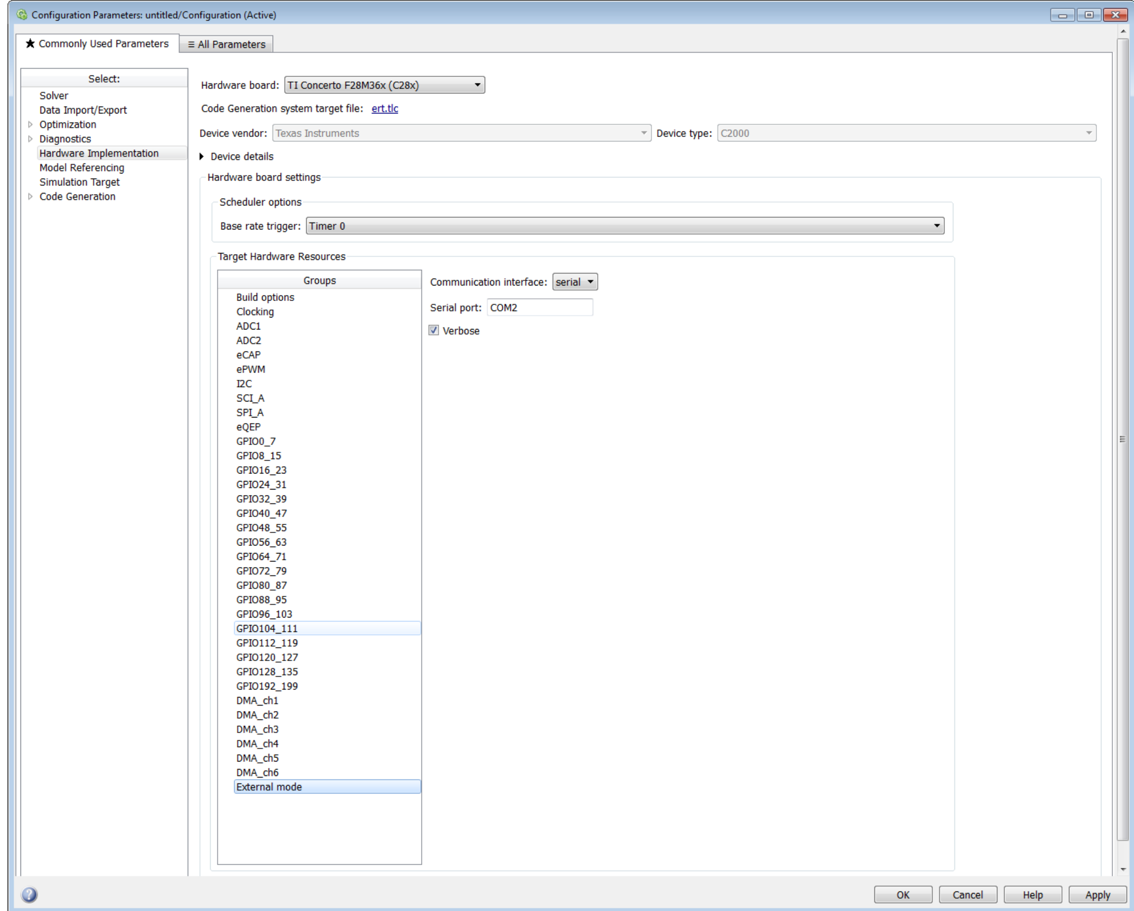
**Enable overflow interrupt**

Enable this parameter to have the DMA channel send an interrupt to the CPU via PIE if the DMA module receives a peripheral interrupt while a previous interrupt from the same peripheral is waiting to be serviced.

This parameter is typically used for debugging during the development phase of a project.

The **Enable overflow interrupt** parameter corresponds to bit 7 (OVRINTE) of the Mode Register (MODE), and involves the Overflow Flag Bit (OVRFLG) and Peripheral Interrupt Trigger Flag Bit (PERINTFLG).

# External mode




Helps you set the external mode settings for your model.

### Communication interface

Use the ‘serial’ option to run your model in the External mode with serial communication.

### Serial port

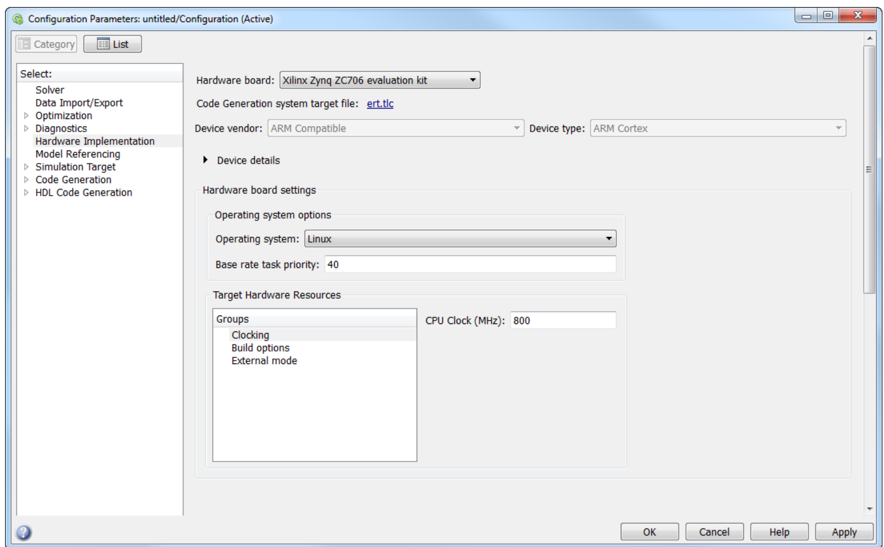
Enter the COMPort used by the target hardware.

To know the comport used by the target hardware on your computer, see “External Mode over Serial Communication” (Embedded Coder Support Package for Texas Instruments C2000 Processors).

**Verbose**

Select this check box to view the External Mode execution progress and updates in the Diagnostic Viewer or in the MATLAB command window.

# Hardware Implementation Pane: Xilinx Zynq ZC702/ ZC706 Evaluation Kits, ZedBoard




**In this section...**

"Hardware Implementation Pane Overview" on page 13-289
"Operating system settings" on page 13-289
"Clocking" on page 13-290
"Build Options" on page 13-290
"External mode" on page 13-291

## Hardware Implementation Pane Overview

Configure the parameters for properties of the physical hardware, such as peripherals.

## Operating system settings

### Operating System

Select the operating system you are using on your target hardware

### Base Rate Task Priority

This parameter sets the static priority of the base rate task. By default, the priority is `40`.

## Clocking

### CPU Clock (MHz)

Specify the CPU clock frequency of the real ARM Cortex processor on the target hardware.

## Build Options

### Build action

Defines how the code generator responds when you press Ctrl+B to build your model.

#### Settings

**Default:** `Build, load and run`

`Build, load and run`

With this option, pressing **Ctrl+B** or clicking **Build Model**:

1. Generates code from the model.
2. Compiles and links the code into an executable with libraries.
3. Loads the executable and libraries into the QEMU emulator.
4. Runs the executable in the QEMU emulator.

`Build`

With this option, pressing **Ctrl+B** or clicking **Build Model**:

1. Generates code from the model.
2. Compiles and links the code into an executable with libraries.

This option does not load and run the executable in the QEMU emulator.

#### Command-Line Information

**Parameter:** `buildAction`
**Type:** character vector

**Value:** `Build` | `Build_load_and_run` |
**Default:** `Build_load_and_run`

#### Recommended Settings

| Application | Setting |
| --- | --- |
| Debugging | `Build_load_and_run` |
| Traceability | No impact |
| Efficiency | No impact |
| Safety precaution | No impact |

#### See Also

For more information, refer to the "Code Generation Pane: Coder Target" topic.

For more information about PIL and its uses, refer to the "Verifying Generated Code via Processor-in-the-Loop" topic.

## External mode

**Communication interface**
: Select the transport layer external mode uses to exchange data between the host computer and the target hardware.

**IP address**
: Specify the IP address of the target hardware. By default, this value is an environment variable, `$(SOCFPGA_IPADDRESS)`, which reuses the IP address from the most recent connection to the target hardware.

**Verbose**
: Display verbose messages in the MATLAB Command Window.

# Recommended Settings Summary for Model Configuration Parameters

The following tables summarizes the impact of each configuration parameter on debugging, traceability, efficiency, and safety considerations, and indicates the factory default configuration settings for the ERT target. The Simulink Coder configuration parameters are documented in “Recommended Settings Summary for Model Configuration Parameters” (Simulink Coder). For additional details, click the links in the Configuration Parameter column.

### Mapping of Application Requirements to the Optimization Pane

| Configuration Parameter | Debugging | Traceability | Efficiency | Safety precaution | Factory Default |
|---|---|---|---|---|---|
| **Application lifespan (days)** (Simulink) | No impact | No impact | Optimal finite value | `inf` | `auto` |
| **Optimize using the specified minimum and maximum values** (Simulink Coder) | Off | Off | On | No recommendatio n | Off |
| **Remove root level I/O zero initialization** (Simulink Coder) | No impact | No impact | On (GUI) `off` (command line) (execution, ROM), No impact (RAM) | Off | Off |
| **Remove internal data zero initialization** (Simulink Coder) | No impact | No impact | On (GUI) `off` (command line) (execution, ROM), No impact (RAM) | Off | Off |
| **Remove code that protects against division arithmetic exceptions** (Simulink Coder) | No impact | No impact | On (execution, ROM) | Off | Off |

| Configuration Parameter | Debugging | Traceability | Efficiency | Safety precaution | Factory Default |
|---|---|---|---|---|---|
| **Pack Boolean data into bitfields** (Simulink Coder) | No impact | No Impact | Off (execution, ROM), On (RAM) | No impact | Off |
| **Pass reusable subsystem outputs as** (Simulink Coder) | No impact | No impact | `Structure reference` (ROM), `Individual arguments` (execution, RAM) | No impact | `Individual Arguments` |
| **Parameter structure** (Simulink Coder) | No impact | `Hierarchical` | `Non-Hierarchical` | No impact | `NonHierarchical` |

#### Mapping of Application Requirements to the Code Generation Pane: Memory Sections Parameters

| Configuration Parameter | Debugging | Traceability | Efficiency | Safety precaution | Factory Default |
|---|---|---|---|---|---|
| **Package** on page 10-3 | No impact | No impact | No impact | No impact | `---None---` |
| **Initialize/-Terminate** on page 10-6 | No impact | No impact | No impact | No impact | `Default` |
| **Execution** on page 10-8 | No impact | No impact | No impact | No impact | `Default` |
| **Shared utility** on page 10-10 | No impact | No impact | No impact | No impact | `Default` |
| **Constants** on page 10-12 | No impact | No impact | No impact | No impact | `Default` |
| **Inputs/Outputs** on page 10-14 | No impact | No impact | No impact | No impact | `Default` |
| **Internal data** on page 10-16 | No impact | No impact | No impact | No impact | `Default` |
| **Parameters** on page 10-18 | No impact | No impact | No impact | No impact | `Default` |
| **Validation results** on page 10-20 | No impact | No impact | No impact | No impact | No package selected. |

**Mapping of Application Requirements to the Code Generation Pane: Report Tab**

| Configuration Parameter | Debugging | Traceability | Efficiency | Safety precaution | Factory Default |
|---|---|---|---|---|---|
| **Code-to-model** (Simulink Coder) | On | On | No impact | No recommendation | On |
| **Model-to-code** (Simulink Coder) | On | On | No impact | No recommendation | On |
| **Generate model Web view** (Simulink Coder) | No impact | No impact | No impact | No impact | Off |
| **Eliminated / virtual blocks** (Simulink Coder) | On | On | No impact | No recommendation | On |
| **Traceable Simulink blocks** (Simulink Coder) | On | On | No impact | No recommendation | On |
| **Traceable Stateflow objects** (Simulink Coder) | On | On | No impact | No recommendation | On |
| **Traceable MATLAB functions** (Simulink Coder) | On | On | No impact | No recommendation | On |
| **Static code metrics** (Simulink Coder) | No impact | No impact | No impact | No impact | Off |

| Configuration Parameter | Debugging | Traceability | Efficiency | Safety precaution | Factory Default |
|---|---|---|---|---|---|
| **Summarize which blocks triggered code replacements** (Simulink Coder) | No impact | No impact | No impact | No impact | Off |

**Mapping of Application Requirements to the Code Generation Pane: Comments Tab**

| Configuration Parameter | Debugging | Traceability | Efficiency | Safety precaution | Factory Default |
|---|---|---|---|---|---|
| **Simulink block descriptions** (Simulink Coder) | On | On | No impact | No impact | On |
| **Simulink data object descriptions** (Simulink Coder) | On | On | No impact | No impact | On |
| **Custom comments (MPT objects only)** (Simulink Coder) | On | On | No impact | No impact | Off |
| **Custom comments function** (Simulink Coder) | Valid file name | Valid file name | No impact | No impact | '' |
| **Stateflow object descriptions** (Simulink Coder) | On | On | No impact | No impact | On |
| **Requirements in block comments** (Simulink Coder) | On | On | No impact | No recommendation | Off |

**Mapping of Application Requirements to the Code Generation Pane: Symbols Tab**

| Configuration Parameter | Debugging | Traceability | Efficiency | Safety precaution | Factory Default |
|---|---|---|---|---|---|
| **Global variables** (Simulink Coder) | No impact | Use default | No impact | No recommendation | `$R$N$M` |
| **Global types** (Simulink Coder) | No impact | Use default | No impact | No recommendation | `&N$R$M_T` |
| **Field name of global types** (Simulink Coder) | No impact | Use default | No impact | No recommendation | `$N$M` |
| **Subsystem methods** (Simulink Coder) | No impact | Use default | No impact | No recommendation | `$R$N$M$F` |
| **Subsystem method arguments** (Simulink Coder) | No impact | Use default | No impact | No recommendation | `rt$I$N$M` |
| **Local temporary variables** (Simulink Coder) | No impact | Use default | No impact | No recommendation | `$N$M` |
| **Local block output variables** (Simulink Coder) | No impact | Use default | No impact | No recommendation | `rtb_$N$M` |
| **Constant macros** (Simulink Coder) | No impact | Use default | No impact | No recommendation | `$R$N$M` |
| **Shared utilities identifier format** (Simulink Coder) | No impact | Use default | No impact | No recommendation | `$N$C` |

| Configuration Parameter | Debugging | Traceability | Efficiency | Safety precaution | Factory Default |
|---|---|---|---|---|---|
| **Minimum mangle length** (Simulink Coder) | No impact | 1 | No impact | No impact | 1 |
| **Maximum identifier length** (Simulink Coder) | Valid value | >30 | No impact | >30 | 31 |
| **System-generated identifiers** (Simulink Coder) | No impact | No impact | No impact | No impact | `Shortened` |
| **Generate scalar inlined parameters as** (Simulink Coder) | No impact | `Macros` | `Literals` | No impact | `Literals` |
| **Use the same reserved names as Simulation Target** (Simulink Coder) | No impact | No impact | No impact | No impact | Off |
| **Shared checksum length** (Simulink Coder) | No impact | No impact | No impact | No impact | 8 |
| **EMX array utlity functions identifier format** (Simulink Coder) | No impact | No impact | No impact | No recommendation | `emx$M$N` |
| **EMX array types identifier format** (Simulink Coder) | No impact | No impact | No impact | No recommendation | `emxArray_$M$N` |

| Configuration Parameter | Debugging | Traceability | Efficiency | Safety precaution | Factory Default |
|---|---|---|---|---|---|
| **Custom token text** (Simulink Coder) | No impact | Set a custom string and use $U in symbols | No impact | No impact | `''` |
| **#define naming** (Simulink Coder) | No impact | `Force uppercase` | No impact | No impact | `None` |
| **Parameter naming** (Simulink Coder) | No impact | `Force uppercase` | No impact | No impact | `None` |
| **Signal naming** (Simulink Coder) | No impact | `Force uppercase` | No impact | No impact | `None` |
| **MATLAB function** (Simulink Coder) | No impact | No impact | No impact | No impact | `''` |

**Mapping of Application Requirements to the Code Generation Pane: Interface Tab**

| Configuration Parameter | Debugging | Traceability | Efficiency | Safety precaution | Factory Default |
|---|---|---|---|---|---|
| **Support: floating-point numbers** (Simulink Coder) | No impact | No impact | Off (GUI), `'on'` (command-line) for integer only | No impact | On (GUI), `'off'` (command-line) |
| **Support complex numbers** (Simulink Coder) | No impact | No impact | Off for real only | No impact | On |
| **Support absolute time** (Simulink Coder) | No impact | No impact | Off | No recommendation | On |
| **Support continuous time** (Simulink Coder) | No impact | No impact | Off (execution, ROM), No impact (RAM) | No recommendation | Off |
| **Support non-inlined S-functions** (Simulink Coder) | No impact | No impact | Off | No recommendation | Off |
| **Support variable-size signals (Simulink Coder)** | No impact | No impact | No impact | No recommendation | Off |
| **Multiword type definitions** (Simulink Coder) | No impact | No impact | No impact | No recommendation | `System defined` |
| **Maximum word length** (Simulink Coder) | No impact | No impact | No impact | No recommendation | 256 for ERT targets<br>2048 for GRT targets |

| Configuration Parameter | Debugging | Traceability | Efficiency | Safety precaution | Factory Default |
|---|---|---|---|---|---|
| **Pass root-level I/O as** (Simulink Coder) | No impact | No impact | No impact | No impact | `Individual arguments` |
| **Use dynamic memory allocation for model initialization** (Simulink Coder) | No impact | No impact | No impact | No recommendation | Off |
| **Terminate function required** (Simulink Coder) | No impact | No impact | No impact | No recommendation | On |
| **Remove error status field in real-time model data structure** (Simulink Coder) | Off | No impact | On | No recommendation | Off |
| **Combine signal/ state structures** (Simulink Coder) | Off | No impact | No impact | On | No impact |
| **Parameter visibility** (Simulink Coder) | No impact | No impact | No impact | No recommendation | `private` |
| **Internal data visibility** (Simulink Coder) | No impact | No impact | No impact | No recommendation | `private` |
| **Parameter access** (Simulink Coder) | `Inlined method` | `Inlined method` | `Inlined method` | No recommendation | `None` |
| **Internal data access** (Simulink Coder) | `Inlined method` | `Inlined method` | `Inlined method` | No recommendation | `None` |

| Configuration Parameter | Debugging | Traceability | Efficiency | Safety precaution | Factory Default |
|---|---|---|---|---|---|
| **External I/O access** (Simulink Coder) | `Inlined method` | `Inlined method` | `Inlined method` | No recommendation | `None` |
| **Generate destructor** (Simulink Coder) | No impact | No impact | No impact | No recommendation | On |
| **Use dynamic memory allocation for model block instantiation** (Simulink Coder) | No impact | No impact | On | No recommendation | Off |

**Mapping of Application Requirements to the Code Generation Pane: Verification Tab**

| Configuration Parameter | Debugging | Traceability | Efficiency | Safety precaution | Factory Default |
|---|---|---|---|---|---|
| **Measure task execution time** on page 12-5 | On | On | Off | No recommendation | Off |
| **Measure function execution times** on page 12-7 | On | On | Off | No recommendation | Off |
| **Workspace variable** on page 12-9 | No impact | Valid MATLAB variable name | No impact | No impact | Off |
| **Save options** on page 12-11 | `All data` | `All data` | `Summary data only` | No impact | `Summary data only` |
| **Third-party tool** on page 12-13 | `BullseyeCoverage` or `LDRA Testbed` | `BullseyeCoverage` or `LDRA Testbed` | `None` (code coverage off) | No recommendation | `None` (code coverage off) |
| **Enable portable word sizes** on page 12-15 | On | On | Off | No impact | Off |
| **Enable source-level debugging for SIL** on page 12-17 | On | On | Off | No impact | Off |

**Mapping of Application Requirements to the Code Generation Pane: Code Style Tab**

| Configuration Parameter | Debugging | Traceability | Efficiency | Safety precaution | Factory Default |
|---|---|---|---|---|---|
| **Parentheses level** on page 8-5 | `Nominal (Optimize for readability)` | `Nominal (Optimize for readability)` | `Minimum (Rely on C/C++ operators for precedence)` | No recommendation | `Nominal (Optimize for readability)` |
| **Preserve operand order in expression** on page 8-7 | On | On | Off | No recommendation | Off |
| **Preserve condition expression in if statement** on page 8-9 | On | On | Off | No recommendation | Off |
| **Convert if-elseif-else patterns to switch-case statements** on page 8-11 | No impact | Off | On (execution, ROM), No impact (RAM) | No impact | On |
| **Preserve extern keyword in function declarations** on page 8-13 | No impact | No impact | No impact | No impact | On |
| **Preserve static keyword in function declarations** on page 8-15 | No impact | No impact | On (execution, ROM) | No impact | On |
| **Suppress generation of default cases for Stateflow switch statements if unreachable** on page 8-17 | On | On | Off | No recommendation | On |
| **Replace multiplications by powers of two with signed bitwise shifts** on page 8-19 | No impact | No impact | On | No impact | On |

| Configuration Parameter | Debugging | Traceability | Efficiency | Safety precaution | Factory Default |
|---|---|---|---|---|---|
| **Casting modes** on page 8-23 | Nominal | Nominal | Nominal | Standards Compliant | Nominal |
| **Indent style** on page 8-25 | K&R | K&R | K&R | K&R | K&R |
| **Indent size** on page 8-27 | 2 | 2 | 2 | 2 | 2 |

**Mapping of Application Requirements to the Code Generation Pane: Templates Tab**

| Configuration Parameter | Debugging | Traceability | Efficiency | Safety precaution | Factory Default |
|---|---|---|---|---|---|
| **Code templates: Source file (*.c) template** on page 11-5 | No impact | No impact | No impact | No impact | `ert_code_-template.cgt` |
| **Code templates: Header file (*.h) template** on page 11-7 | No impact | No impact | No impact | No impact | `ert_code_-template.cgt` |
| **Data templates: Source file (*.c) template** on page 11-9 | No impact | No impact | No impact | No impact | `ert_code_-template.cgt` |
| **Data templates: Header file (*.h) template** on page 11-11 | No impact | No impact | No impact | No impact | `ert_code_-template.cgt` |
| **File customization template** on page 11-13 | No impact | No impact | No impact | No impact | `example_file_-process.tlc` |
| **Generate an example main program** on page 11-15 | No impact | No impact | No impact | No impact | On |
| **Target operating system** on page 11-18 | No impact | No impact | No impact | No impact | `BareBoard-Example` |

**Mapping of Application Requirements to the Code Generation Pane: Code Placement Tab**

| Configuration Parameter | Debugging | Traceability | Efficiency | Safety precaution | Factory Default |
|---|---|---|---|---|---|
| **Data definition** on page 7-5 | No impact | Valid value | No impact | No impact | Auto |
| **Data definition filename** on page 7-7 | No impact | Valid value | No impact | No impact | `global.c` |
| **Data declaration** on page 7-9 | No impact | Valid value | No impact | No impact | Auto |
| **Data declaration filename** on page 7-11 | No impact | Valid value | No impact | No impact | `global.h` |
| **#include file delimiter** on page 7-13 | No impact | Valid value | No impact | No impact | `off` |
| **#include file delimiter** on page 7-15 | No impact | Valid value | No impact | No impact | Auto |
| **Signal display level** on page 7-17 | No impact | Valid integer | No impact | No impact | 10 |
| **Parameter tune level** on page 7-19 | No impact | Valid integer | No impact | No impact | 10 |
| **File packaging format** on page 7-21 | No impact | No impact | No impact | No impact | `Modular` |

**Mapping of Application Requirements to the Code Generation Pane: Data Type Replacement Tab**

| Configuration Parameter | Debugging | Traceability | Efficiency | Safety precaution | Factory Default |
|---|---|---|---|---|---|
| **Replace data type names in the generated code** on page 9-5 | No impact | On | No impact | No impact | Off |
| **Replacement Name** on page 9-7 | No impact | Valid character vector | No impact | No recommendation | '' |

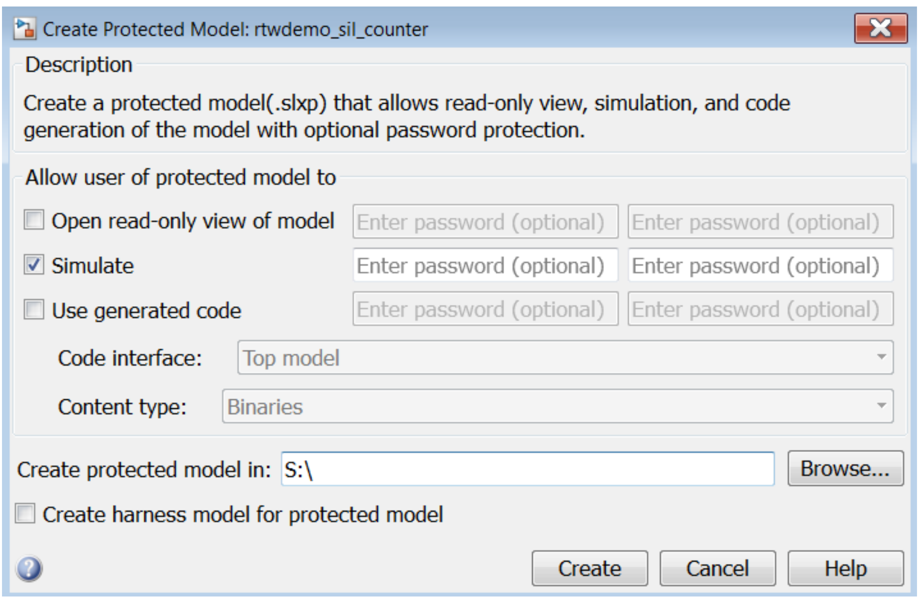
# See Also

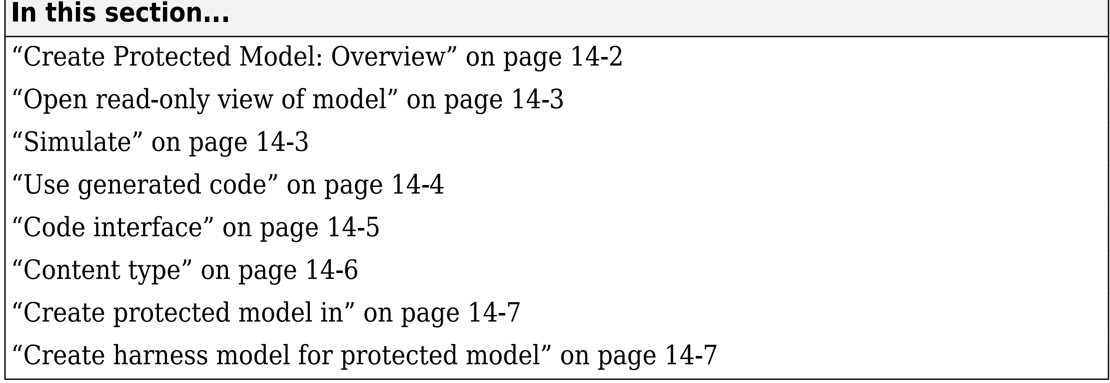
## Related Examples

- “Configure Model for Code Generation Objectives by Using Code Generation Advisor”
- “Code Generation Advisor Checks” (Simulink Coder)

# Parameters for Creating Protected Models

# Create Protected Model



**In this section...**

"Create Protected Model: Overview" on page 14-2
"Open read-only view of model" on page 14-3
"Simulate" on page 14-3
"Use generated code" on page 14-4
"Code interface" on page 14-5
"Content type" on page 14-6
"Create protected model in" on page 14-7
"Create harness model for protected model" on page 14-7

## 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

- "Simulate Protected Models from Third Parties" (Simulink)
- "Create a Protected Model" (Simulink Coder)

## 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

- "Create a Protected Model" (Simulink Coder)
- "Protect Models for Third-Party Use" (Simulink Coder)

## Simulate

Allow 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 Mode and Accelerator 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

- “Create a Protected Model” (Simulink Coder)
- “Protect Models for Third-Party Use” (Simulink 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 Support in Protected Models” (Simulink Coder)
- “Protect Models for Third-Party Use” (Simulink 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 Support in Protected Models” (Simulink Coder)
- “Protect Models for Third-Party Use” (Simulink Coder)
- “Code Interfaces for SIL and PIL”

## Content type

Select the appearance of the generated code.

#### Settings

**Default:** `Obfuscated source code`

`Binaries`

    Includes only binaries for the generated code.

`Obfuscated source code`

    Includes obfuscated headers and binaries for the generated code.

`Readable source code`

    Includes readable source code.

#### Dependencies

This parameter is enabled by selecting the **Use generated code** check box.

#### Alternatives

`Simulink.ModelReference.protect`

#### See Also

- “Code Generation Support in Protected Models” (Simulink Coder)
- “Protect Models for Third-Party Use” (Simulink Coder)

## Create protected model in

Specify the folder path for 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 for Third-Party Use” (Simulink Coder)
- “Create a Protected Model” (Simulink Coder)

## Create harness model for protected model

Create a harness model for the protected model. The harness model contains only a Model block that references the protected model.

#### Settings

**Default:** Off

On

Create a harness model for the protected model.

Off

Do not create a harness model for the protected model.

### Alternatives

`Simulink.ModelReference.protect`

### See Also

- "Harness Model" (Simulink Coder)
- "Test the Protected Model" (Simulink Coder)

# Model Advisor Checks

# Embedded Coder Checks

**In this section...**


"Embedded Coder Checks Overview" on page 15-3

"Check for blocks not recommended for C/C++ production code deployment" on page 15-3

"Identify lookup table blocks that generate expensive out-of-range checking code" on page 15-4

"Check output types of logic blocks" on page 15-6

"Check the hardware implementation" on page 15-7

"Identify questionable software environment specifications" on page 15-8

"Identify questionable code instrumentation (data I/O)" on page 15-10

"Check configuration parameters for MISRA C:2012" on page 15-11

"Check for blocks not recommended for MISRA C:2012" on page 15-14

"Check for unsupported block names" on page 15-16

"Check usage of Assignment blocks" on page 15-16

"Check for switch case expressions without a default case" on page 15-18

"Check for missing error ports for AUTOSAR receiver interfaces" on page 15-19

"Check bus object names that are used as element names" on page 15-20

"Check configuration parameters for secure coding standards" on page 15-20

"Check for blocks not recommended for secure coding standards" on page 15-23

"Identify questionable subsystem settings" on page 15-24

"Identify blocks that generate expensive fixed-point and saturation code" on page 15-25

"Check for missing const qualifiers in model functions" on page 15-27

"Identify questionable fixed-point operations" on page 15-28

"Identify blocks that generate expensive rounding code" on page 15-31

"Check for bitwise operations on signed integers" on page 15-32

"Check for recursive function calls" on page 15-33

"Check for equality and inequality operations on floating-point values" on page 15-33

"Check integer word length" on page 15-34

| In this section... |
| --- |
| “Check block names” on page 15-35 |

## Embedded Coder Checks Overview

Use Embedded Coder Model Advisor checks to configure your model for code generation.

### See Also

- “Run Model Checks” (Simulink)
- “Simulink Checks” (Simulink)
- “Simulink Coder Checks” (Simulink Coder)

## Check for blocks not recommended for C/C++ production code deployment

**Check ID**: `mathworks.codegen.PCGSupport`

Identify blocks not supported by code generation or not recommended for C/C++ production code deployment.

### Description

This check partially identifies model constructs that are not recommended for C/C++ production code generation. For Simulink Coder and Embedded Coder, these model construct identities appear in tables of Simulink Block Support (Simulink Coder). If you are using blocks with support notes for code generation, review the information and follow the given advice.

Available with Simulink Check™ and Embedded Coder.

### Results and Recommended Actions

| Condition | Recommended Action |
| --- | --- |
| The model or subsystem contains blocks that should not be used for production code deployment. | Consider replacing the blocks listed in the results. Click an element from the list of questionable items to locate condition. |

#### Capabilities and Limitations

You can:

- Run this check on your library models.
- Analyzes content of library linked blocks.
- Analyzes content in all masked subsystems.
- Exclude blocks and charts from this check if you have a Simulink Check license.

#### See Also

- “Blocks and Products Supported for C Code Generation” (Simulink Coder)
- “What Is a Model Advisor Exclusion?” (Simulink Check)

## Identify lookup table blocks that generate expensive out-of-range checking code

**Check ID**: `mathworks.codegen.LUTRangeCheckCode`

Identify lookup table blocks that generate code to protect against out-of-range inputs for breakpoint or index values.

#### Description

This check verifies that the following blocks do not generate code to protect against inputs that fall outside the range of valid breakpoint values:

- 1-D Lookup Table
- 2-D Lookup Table
- n-D Lookup Table
- Prelookup

This check also verifies that Interpolation Using Prelookup blocks do not generate code to protect against inputs that fall outside the range of valid index values.

Following the recommended actions increases both execution and ROM efficiency of the generated code.

Available with Embedded Coder.

#### Results and Recommended Actions

| Condition | Recommended Action |
|---|---|
| The lookup table block generates out-of-range checking code. | Change the setting on the block dialog box so that out-of-range checking code is not generated.<br>• For the 1-D Lookup Table, 2-D Lookup Table, n-D Lookup Table, and Prelookup blocks, select the check box for **Remove protection against out-of-range input in generated code**.<br>• For the Interpolation Using Prelookup block, select the check box for **Remove protection against out-of-range index in generated code**. |

#### Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Check license.

#### Action Results

Clicking **Modify** prevents lookup table blocks from generating out-of-range checking code, which makes the generated code more efficient.

#### See Also

- n-D Lookup Table
- Prelookup
- Interpolation Using Prelookup
- “Optimize Generated Code for Lookup Table Blocks” (Simulink)
- “What Is a Model Advisor Exclusion?” (Simulink Check)

## Check output types of logic blocks

**Check ID**: `mathworks.codegen.LogicBlockUseNonBooleanOutput`

Identify logic blocks that do not use `boolean` for the output data type.

### Description

This check verifies that the output data type of the following blocks is `boolean`:

- Compare To Constant
- Compare To Zero
- Detect Change
- Detect Decrease
- Detect Fall Negative
- Detect Fall Nonpositive
- Detect Increase
- Detect Rise Nonnegative
- Detect Rise Positive
- Interval Test
- Interval Test Dynamic
- Logical Operator
- Relational Operator

Using output data type `boolean` increases execution efficiency of the generated code.

Available with Embedded Coder.

### Results and Recommended Actions

| Condition | Recommended Action |
| --- | --- |
| The output data type of a logic block is not `boolean`. | In the block dialog box, set **Output data type** to `boolean`. |

### Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Check license.

### See Also

- “What Is a Model Advisor Exclusion?” (Simulink Check)

### Action Results

Clicking **Modify** forces logic blocks to use `boolean` as the output data type. If a logic block uses `uint8` for the output type, clicking **Modify** changes the output type to `boolean`.

## Check the hardware implementation

**Check ID**: `mathworks.codegen.HWImplementation`

Identify inconsistent or underspecified hardware implementation settings

### Description

The Simulink and Simulink Coder software require two sets of target specifications. The first set describes the final intended production target. The second set describes the currently selected target. If the configurations do not match, the code generator creates extra code to emulate the behavior of the production target. Inconsistencies or underspecification of hardware attributes can lead to inefficient or incorrect code generation for the target hardware.

Available with Embedded Coder.

### Results and Recommended Actions

| Condition | Recommended Action |
| --- | --- |
| Hardware implementation parameters are not set to recommended values. | In the Configuration Parameters dialog box, on the **Hardware Implementation** (Simulink) pane, specify the following parameters:<br>• **Byte ordering** (`ProdEndianess`)<br>• **Signed integer division rounds to** (`ProdIntDivRoundTo`)<br>In the Configuration Parameters dialog box, specify the following parameters:<br>• **Test hardware byte ordering** (`TargetEndianess`)<br>• **Test hardware signed integer division rounds to** (`TargetIntDivRoundTo`) |
| Hardware implementation **Production hardware** settings do not match **Test hardware** settings. | In the Configuration Parameters dialog box, consider selecting the **Test hardware is the same as production hardware** (`ProdEqTarget`) check box, or modify the settings to match. |

### See Also

"Run-Time Environment Configuration"

## Identify questionable software environment specifications

**Check ID**: `mathworks.codegen.SWEnvironmentSpec`

Identify questionable software environment settings.

### Description

- Support for some software environment settings can lead to inefficient code generation and nonoptimal results.

- Industry standards for C, such as ISO and MISRA, require identifiers to be unique within the first 31 characters.
- Stateflow charts with weak Simulink I/O data types lead to inefficient code.

Available with Embedded Coder.

### Results and Recommended Actions

| Condition | Recommended Action |
| --- | --- |
| The maximum identifier length does not conform with industry standards for C. | In the Configuration Parameters dialog box, on the **Code Generation > Symbols** pane, set the **Maximum identifier length** (Simulink Coder) parameter to `31` characters. |
| In the Configuration Parameters dialog box, the parameters on the **Code Generation** > **Interface** pane are not set to recommended values. | In the Configuration Parameters dialog box, on the **Code Generation** > **Interface** (Simulink Coder) pane, clear the following parameters:<br>• **Support: continuous time**<br>• **Support: non-finite numbers**<br>In the Configuration Parameters dialog box, clear **Support non-inlined S-functions**. |
| In the Configuration Parameters dialog box, the parameters on the **Code Generation** > **Symbols** pane are not set to recommended values. | In the Configuration Parameters dialog box, on the **Code Generation** > **Symbols** pane, set the **Generate scalar inlined parameters as** (Simulink Coder) parameter to `Literals`. |
| In the Configuration Parameters dialog box, on the **Code Generation** > **Interface** pane, **Support: variable-size signals** is selected. This might lead to inefficient code. | If you do not intend to support variable-sized signals, in the Configuration Parameters dialog box, on the **Code Generation** > **Interface** pane, clear **Support: variable-size signals** (Simulink Coder). |

| Condition | Recommended Action |
|---|---|
| The model contains Stateflow charts with weak Simulink I/O data type specifications. | Select the Stateflow chart property **Use Strong Data Typing with Simulink I/O** (Stateflow). You might need to adjust the data types in your model after selecting the property. |

### Limitations

A Stateflow license is required when using Stateflow charts.

### See Also

"Strong Data Typing with Simulink I/O" (Stateflow)

## Identify questionable code instrumentation (data I/O)

**Check ID**: `mathworks.codegen.CodeInstrumentation`

Identify questionable code instrumentation.

### Description

- Instrumentation of the generated code can cause nonoptimal results.
- Test points require global memory and are not optimal for production code generation.

Available with Embedded Coder.

### Results and Recommended Actions

| Condition | Recommended Action |
|---|---|
| Interface parameters are not set to recommended values. | In the Configuration Parameters dialog box, on the **Code Generation** > **Interface** (Simulink Coder) pane, set the parameters to the recommended values. |
| Blocks generate assertion code. | In the Configuration Parameters dialog box, set **Model Verification block enabling** (Simulink) to `Disable All` on a block-by-block basis or globally. |

| Condition | Recommended Action |
|---|---|
| Block output signals have one or more test points and, if you have an Embedded Coder license, the **Ignore test point signals** check box is cleared in the Configuration Parameters dialog box. | Remove test points from the specified block output signals. For each signal, in the Signal Properties dialog box (Simulink), clear the **Test point** check box.<br><br>Alternatively, if the model is using an ERT-based system target file, select the **Ignore test point signals** check box in the Configuration Parameters dialog box to ignore test points during code generation. |

### Capabilities and Limitations

If you have a Simulink Check license, you can exclude blocks and charts from this check.

### See Also

- “What Is a Model Advisor Exclusion?” (Simulink Check)

## Check configuration parameters for MISRA C:2012

**Check ID**: `mathworks.misra.CodeGenSettings`

Identify configuration parameters that might impact MISRA C:2012 compliant code generation.

### Description

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications.

Available with Embedded Coder and Simulink Check.

**Results and Recommended Actions**

| Condition | Recommended Action |
|---|---|
| **Use division for fixed-point net slope computation** is not set to `On` or `Use division for reciprocals of integers only`. | Set configuration parameter **Use division for fixed-point net slope computation** to `On` or `Use division for reciprocals of integers only`. |
| **Bitfield declarator type specifier** is set to `uchar_T when any these parameters are selected:`<br>• **Pack Boolean data into bitfields**<br>• **Use bitsets for storing state configuration**<br>• **Use bitsets for storing Boolean data** | Set configuration parameter **Bitfield declarator type specifier** to `uint_T`. |
| **Model Verification block enabling** is set to `Use local settings` or `Enable All`. | Set configuration parameter **Model Verification block enabling** to `Disable All`. |
| **Wrap on overflow** is set to `None` | Set configuration parameter **Wrap on overflow** to `warning` or `error`. |
| **Inf or NaN block output** is set to `None` | Set configuration parameter **Inf or NaN block output** to `warning` or `error`. |
| **Signed integer division rounds to** is not set to `Zero` or `Floor`. | Set configuration parameter **Signed integer division rounds to** to `Zero` or `Floor`. |
| **Dynamic memory allocation in MATLAB Function blocks** is selected. | Clear configuration parameter **Dynamic memory allocation in MATLAB Function blocks**. |
| **System target file** is set to a GRT-based target. | Set configuration parameter **System target file** to an ERT-based target. |
| **Maximum identifier length** | Set the configuration parameter value to the implementation dependent limit. The default is `31`. |
| **System-generated identifiers** is set to `Classic`. | Set configuration parameter **System-generated identifiers** to `Shortened`. |

| Condition | Recommended Action |
|---|---|
| **Code replacement library**is not set to None or AUTOSAR 4.0. | Set configuration parameter **Code replacement library** to None or AUTOSAR 4.0 |
| **Shared code placement** is set to Auto. | Set configuration parameter **Shared code placement** to Shared location |
| **Support: non-finite numbers** is selected. | Clear configuration parameter **Support: non-finite numbers** |
| **Support: complex numbers** is selected. | Clear configuration parameter **Support: complex numbers** (ERT-based target only) |
| **Support: continuous time** is selected. | Clear configuration parameter **Support: continuous time** (ERT-based target only) |
| **MAT-file logging** is selected. | Clear configuration parameter **MAT-file logging** |
| **Generate shared constants** is selected. | Clear configuration parameter **Generate shared constants**. |
| **Standard math library** does not indicate the correct C programming language. | Set configuration parameter **Standard math library** to C89/C90 (ANSI) or C99 (ISO) depending on toolchain |
| **Support non-inlined S-functions** is selected. | Clear configuration parameter **Support non-inlined S-functions** (ERT-based target only) |
| **Use dynamic memory allocation for model initialization** is selected when **Code Interface Packaging** is set to Reusable Function. | Clear configuration parameter **Use dynamic memory allocation for model initialization**.<br><br>Select only when configuration parameter **Code Interface Packaging** is set to Reusable Function. |
| **Parenthesis level** is not set to Maximum (Specify precedence with parentheses). | Set configuration parameter **Parentheses level** to Maximum (Specify precedence with parentheses). |
| **Replace multiplications by powers of two with signed bitwise shifts** is selected. | Clear configuration parameter **Replace multiplications by powers of two with signed bitwise shifts**. |

| Condition | Recommended Action |
|---|---|
| **Casting Modes** is not set to `Standards Compliant`. | Set configuration parameter **Casting Modes** to `Standards Compliant`. |
| **Allow right shifts on signed integers** is selected. | Clear configuration parameter **Allow right shifts on signed integers**. |
| **Preserve static keyword in function declarations** is cleared when **File packaging format** is set to `Compact` or `CompactWithDataFile`. | Select configuration parameter **Preserve static keyword in function declarations** (ERT-based target only). |

### Action Results

Clicking **Modify All** changes the parameter values to the recommended values.

Subchecks depend on the results of the subchecks noted with **D** in the results table in the Model Advisor window.

### Capabilities and Limitations

This check does not review referenced models.

### See Also

- "hisl_0060: Configuration parameters that improve MISRA C:2012 compliance" (Simulink)
- "MISRA C Guidelines"
- "MISRA C:2012 Compliance Considerations" (Simulink)

## Check for blocks not recommended for MISRA C:2012

**Check ID**: `mathworks.misra.BlkSupport`

Identify blocks that are not supported or recommended for MISRA C:2012 compliant code generation.

### Description

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications.

Available with Embedded Coder and Simulink Check.

#### Results and Recommended Actions

| Condition | Recommended Action |
| --- | --- |
| Lookup Table blocks using cubic spline interpolation or extrapolation methods were found in the model or subsystem. | Consider other interpolation and extrapolation methods for the Lookup Table blocks. |
| Deprecated Lookup Table blocks were found in the model or subsystem.<br><br>The deprecated Lookup Table blocks are Lookup and Lookup2D. | Consider replacing the deprecated Lookup Table blocks. |
| S-Function Builder blocks were found in the model or subsystem. | Consider replacing the S-Function Builder blocks with blocks recommended for production. |
| From Workspace blocks were found in the model or subsystem. | Consider replacing the From Workspace blocks with blocks recommended for production. |

#### Capabilities and Limitations

You can:

- Run this check on your library models.
- Analyzes content of library linked blocks.
- Analyzes content in all masked subsystems.
- Exclude blocks and charts from this check if you have a Simulink Check license.

#### See Also

- “hisl_0020: Blocks not recommended for MISRA C:2012 compliance” (Simulink)
- “MISRA C Guidelines”
- “MISRA C:2012 Compliance Considerations” (Simulink)
- “What Is a Model Advisor Exclusion?” (Simulink Check)

## Check for unsupported block names

**Check ID**: `mathworks.misra.BlockNames`

Identify block names containing `/`.

### Description

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications.

Available with Embedded Coder and Simulink Check.

### Results and Recommended Actions

| Condition | Recommended Action |
| --- | --- |
| Block names containing `/` were found in the model or subsystem. | Remove `/` from the block name. |

### Capabilities and Limitations

- Runs on library models.
- Analyzes content of library linked blocks.
- Analyzes content in all masked subsystems.
- If you have a Simulink Check license, allows exclusions of blocks and charts.

### See Also

- MISRA C:2012, Rule 3.1
- “MISRA C Guidelines”.
- “MISRA C:2012 Compliance Considerations” (Simulink)

## Check usage of Assignment blocks

**Check ID**: `mathworks.misra.AssignmentBlocks`

Identify Assignment blocks that do not have block parameter **Action if any output element is not assigned** set to **Error** or **Warning**.

#### Description

This check applies to the Assignment block that is available in the Simulink block library under **Simulink > Math Operations**.

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications, as well as code that complies with the CERT C, CWE, ISO/IEC TS 17961 standards.

Available with Embedded Coder and Simulink Check.

#### Results and Recommended Actions

| Condition | Recommended Action |
| --- | --- |
| The model or subsystem might contain Assignment blocks with incomplete array initialization that do not have block parameter **Action if any output element is not assigned** set to **Error** or **Warning**. | Set block parameter **Action if any output element is not assigned** to one of the recommended values:<br>• **Error**, if Assignment block is not in an Iterator subsystem.<br>• **Warning**, if Assignment block is in an Iterator subsystem. |

#### Capabilities and Limitations

- Runs on library models.
- Analyzes content of library linked blocks.
- Analyzes content in all masked subsystems.
- If you have a Simulink Check license, allows exclusions of blocks and charts.

#### See Also

- MISRA C:2012, Rule 9.1
- ISO/IEC TS 17961: 2013, uninitref
- CERT C, EXP33-C
- CWE, CWE-908
- “hisl_0029: Usage of Assignment blocks” (Simulink)
- “MISRA C Guidelines”

- “MISRA C:2012 Compliance Considerations” (Simulink)
- “Secure Coding Standards”

## Check for switch case expressions without a default case

**Check ID**: `mathworks.misra.SwitchDefault`

Identify switch case expressions that do not have a default case.

### Description

The check flags model objects that have switch case expressions without a default case.

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications, as well as code that complies with the CERT C, CWE, ISO/IEC TS 17961 standards.

The check does not flag blocks without default cases if they are justified with a Polyspace® annotation. When you run the check, the **Blocks with justification** table lists blocks without default cases that have a justification.

Available with Embedded Coder and Simulink Check.

### Results and Recommended Actions

| Condition | Recommended Action |
| --- | --- |
| Model object has a switch case expression without a default case. | For Switch Case blocks, consider selecting block parameter **Show default case** to explicitly specify a default case. |

### Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Check license.

### See Also

- MISRA C:2012, Rule 16.4

- ISO/IEC TS 17961: 2013, swtchdflt
- CERT C, MSC01-C
- CWE, CWE-478
- “Annotate Code and Hide Known or Acceptable Results” (Polyspace Bug Finder)
- “Secure Coding Standards”

## Check for missing error ports for AUTOSAR receiver interfaces

**Check ID**: `mathworks.misra.AutosarReceiverInterface`

Identify AUTOSAR receiver interface inports that do not have matching error ports.

### Description

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications. The check flags AUTOSAR receiver interfaces inports that are missing error ports.

The check does not flag missing error ports if they are justified with a Polyspace annotation. When you run the check, the **Blocks with justification** table lists the missing error ports that have a justification.

Available with Embedded Coder and Simulink Check.

### Results and Recommended Actions

| Condition | Recommended Action |
| --- | --- |
| AUTOSAR receiver interface inport does not have a matching error port. | Add missing error port and map to the corresponding AUTOSAR receiver interface inport. |

### Capabilities and Limitations

You can:

- Analyzes top layer/root level models.
- Exclude blocks and charts from this check if you have a Simulink Check license.

#### See Also

- MISRA C: 2012, Directive 4.7
- “MISRA C Guidelines”
- “What Is a Model Advisor Exclusion?” (Simulink Check)
- “Annotate Code and Hide Known or Acceptable Results” (Polyspace Bug Finder)

## Check bus object names that are used as element names

**Check ID**: `mathworks.misra.BusElementNames`

Identify bus object names that are used as bus element names.

#### Description

Using this check increases the likelihood of generating code for embedded applications that is compliant with MISRA C:2012. The check flags instances where a Simulink.Bus object name is used as the Simulink.Bus element name.

Available with Embedded Coder and Simulink Check.

#### Results and Recommended Actions

| Condition | Recommended Action |
| --- | --- |
| A bus object name is being used as a bus element name. | Change either the flagged bus object name or the bus element name so that they are not identical. |

#### See Also

- MISRA C:2012, Rule 5.6
- MISRA AC AGC, Rule 5.3
- “MISRA C Guidelines”

## Check configuration parameters for secure coding standards

**Check ID**: `mathworks.security.CodeGenSettings`

Identify configuration parameters that might impact compliance with secure coding standards.

### Description

Following the recommendations of this check increases the likelihood of generating code that complies with CERT C, CWE, ISO/IEC TS 17961 secure coding standards.

Available with Embedded Coder and Simulink Check.

### Results and Recommended Actions

| Condition | Recommended Action |
| --- | --- |
| **Model Verification block enabling** is set to `Use local settings` or `Enable All`. | In the Configuration Parameters dialog box, set **Model Verification block enabling** to `Disable All`. |
| **System target file** is set to a GRT-based target. | In the Configuration Parameters dialog box, on the **Code Generation** > **General** pane, set **System target file** to an ERT-based target. |
| **Code Generation** > **Interface** parameters are not set to the recommended values. | In the Configuration Parameters dialog box, on the **Code Generation** > **Interface** pane:<br>• Set **Code replacement library** to `None` or `AUTOSAR 4.0`<br>• Clear **Support: non-finite numbers**<br>• Clear **Support: continuous time** (ERT-based target only)<br>In the Configuration Parameters dialog box:<br>• Clear **Support non-inlined S-functions** (ERT-based target only)<br>• Clear **MAT-file logging** |

| Condition | Recommended Action |
|---|---|
| **Signed integer division rounds to** is not set to `Zero` or `Floor`. | In the Configuration Parameters dialog box, on the **Hardware Implementation** pane, set **Signed integer division rounds to** to `Zero` or `Floor`. |
| **Replace multiplications by powers of two with signed bitwise shifts** is selected. | In the Configuration Parameters dialog box, on the **Code Generation** > **Code Style** pane, clear **Replace multiplications by powers of two with signed bitwise shifts**. |
| **Allow right shifts on signed integers** is selected. | In the Configuration Parameters dialog box, on the **Code Generation** > **Code Style** pane, clear **Allow right shifts on signed integers**. |
| **Use dynamic memory allocation for model initialization** is selected. | In the Configuration Parameters dialog box, clear **Use dynamic memory allocation for model initialization**. |
| **Wrap on overflow** is set to `None` | In the Configuration Parameters dialog box, on the **Diagnostics** > **Data Validity** pane, set **Wrap on overflow** to `warning` or `error`. |
| **Inf or NaN block output** is set to `None` | In the Configuration Parameters dialog box, on the **Diagnostics** > **Data Validity** pane, set **Inf or NaN block output** to `warning` or `error`. |
| **Dynamic memory allocation in MATLAB Function blocks** is selected. | In the Configuration Parameters dialog box, clear **Dynamic memory allocation in MATLAB Function blocks**. |

### Action Results

Clicking **Modify All** changes the parameter values to the recommended values.

Subchecks depend on the results of the subchecks noted with **D** in the results table in the Model Advisor window.

### See Also

"Secure Coding Standards"

## Check for blocks not recommended for secure coding standards

**Check ID**: `mathworks.security.BlockSupport`

Identify blocks not recommended for compliance with secure coding standards.

### Description

Following the recommendations of this check increases the likelihood of generating code that complies with CERT C, CWE, ISO/IEC TS 17961 secure coding standards.

Available with Embedded Coder and Simulink Check.

### Results and Recommended Actions

| Condition | Recommended Action |
| --- | --- |
| Lookup Table blocks using cubic spline interpolation or extrapolation methods were found in the model or subsystem. | Consider other interpolation and extrapolation methods for the Lookup Table blocks. |
| Deprecated Lookup Table blocks were found in the model or subsystem. | Consider replacing the deprecated Lookup Table blocks. |
| S-Function Builder blocks were found in the model or subsystem. | Consider replacing the S-Function Builder blocks with blocks recommended for production. |
| From Workspace blocks were found in the model or subsystem | Consider replacing the From Workspace blocks with blocks recommended for production. |

### 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

- “What Is a Model Advisor Exclusion?” (Simulink Check)
- “Secure Coding Standards”

## Identify questionable subsystem settings

**Check ID**: `mathworks.codegen.QuestionableSubsysSetting`

Identify questionable subsystem block settings.

### Description

Subsystem blocks implemented as void-void functions in the generated code use global memory to store the subsystem I/O.

Available with Embedded Coder.

### Results and Recommended Actions

| Condition | Recommended Action |
| --- | --- |
| Subsystem blocks have the **Subsystem Parameters** > **Function packaging** option set to `Nonreusable function`. | Set the **Subsystem Parameters** > **Function packaging** parameter to `Auto`. |
| Subsystem blocks have the **Subsystem Parameters** > **Function packaging** option set to `Reusable function`. | Set the **Subsystem Parameters** > **Function packaging** parameter to `Auto`. |

### Capabilities and Limitations

If you have a Simulink Check license, you can exclude blocks and charts from this check.

### See Also

- Subsystem block
- “What Is a Model Advisor Exclusion?” (Simulink Check)

## Identify blocks that generate expensive fixed-point and saturation code

**Check ID**: `mathworks.codegen.BlockSpecificQuestionableFxptOperations`

Identify fixed-point operations that can lead to nonoptimal results.

### Description

Certain block settings can lead to expensive fixed-point and saturation code.

### Results and Recommended Actions

| Conditions | Recommended Action |
| --- | --- |
| Blocks generate expensive saturation code. | Check whether your application requires setting **Function Block Parameters** > **Signal Attributes** > **Saturate on integer overflow**. Otherwise, clear the **Saturate on integer overflow** parameter for the most efficient implementation of the block in the generated code. |
| Product blocks are multiplying signals with mismatched slope adjustment factors. The net slope computation uses multiplication followed by shifts, which is inefficient for some target hardware. | Set the **Optimization** > **Use division for fixed-point net slope computation** parameter to `On`, or `Use division for reciprocals of integers only` if the net slope can be approximated by a fraction and division is more efficient than multiplication and shifts on the target hardware.<br><br>**Note** This optimization takes place only if certain simplicity and accuracy conditions are met. For more information, see “Handle Net Slope Computation” (Fixed-Point Designer). |
| Product blocks are configured with a divide operation for the first input and a multiply operation for the second input. | Reverse the inputs so the multiply operation occurs first and the division operation occurs second. |

| Conditions | Recommended Action |
| --- | --- |
| Product blocks are configured to do multiple division operations. | Multiply all the denominator terms together, and then do a single division using cascading Product blocks. |
| Product blocks are configured to do many multiplication or division operations. | Split the operations across several blocks, with each block performing one multiplication or one division operation. |
| Protection code generated as part of the division operation is redundant. | Verify that your model cannot cause exceptions in division operations and then remove redundant protection code by setting the **Optimization > Remove code that protects against division arithmetic exceptions** (Simulink Coder) parameter in the Configuration Parameters dialog box. |
| The data type range of the inputs of Sum blocks exceeds the data type range of the output, which can cause overflow or saturation. | Change the output and accumulator data types so the range equals or exceeds all input ranges.<br><br>For example, if the model has two inputs<br>• `int8` (–128 to 127)<br>• `uint8` (0 to 255)<br><br>The data type range of the output and accumulator must equal or exceed –128 to 255. A `int16` (–32768 to 32767) data type meets this condition. |
| A Sum block has an input with a slope adjustment factor that does not equal the slope adjustment factor of the output. | Change the data types so the inputs, outputs, and accumulator have the same slope adjustment factor. |
| The net sum of the Sum block input biases does not equal the bias of the output. | Change the bias of the output scaling, making the net bias adjustment zero. |
| The input and output of the MinMax block have different data types. | Change the data type of the input or output. |

| Conditions | Recommended Action |
|---|---|
| The input of the MinMax block has a different slope adjustment factor than the output. | Change the scaling of the input or the output. |
| The initial condition of the Discrete-Time Integrator block is used to initialize both the state and the output. | Set the **Function Block Parameters > Initial condition setting** parameter to `State (most efficient)`. |
| Parameter overflow occurred for the Compare to Zero block. This block uses the input data type to represent zero. The input data type cannot represent zero exactly, so the input value was compared to the closest representable value of zero. | Select an input data type that can represent zero. |
| Parameter overflow occurred for the following Compare to Constant block. This block uses the input data type to represent its **Constant value** parameter. The **Constant value** parameter is outside the range that the input data type can represent. The input signal was compared to the closest representable value of the **Constant value** parameter. | Choose an input data type that can represent the **Constant value** parameter or change the **Constant value** parameter to match the input data type. |

### Capabilities and Limitations

- A Fixed-Point Designer license is required to generate fixed-point code.
- If you have a Simulink Check license, you can exclude blocks and charts from this check.

### See Also

- “Identify Blocks that Generate Expensive Fixed-Point and Saturation Code” (Fixed-Point Designer)
- “What Is a Model Advisor Exclusion?” (Simulink Check)

## Check for missing const qualifiers in model functions

**Check ID**: `mathworks.misra.ModelFunctionInterface`

Identify missing const qualifiers in input data pointers.

#### Description

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications. The check flags input data pointers that do not have a const qualifier.

Available with Embedded Coder and Simulink Check.

#### Results and Recommended Actions

| Condition | Recommended Action |
| --- | --- |
| A const qualifier is not defined for the input data pointer. | Consider adding a const qualifier to the input data pointer. |

#### See Also

- MISRA C:2012, Rule 8.13
- “MISRA C Guidelines”

## Identify questionable fixed-point operations

**Check ID**: `mathworks.codegen.QuestionableFxptOperations`

Identify fixed-point operations that can lead to nonoptimal results.

#### Description

Less efficient code can result from blocks that generate cumbersome multiplication and division operations, expensive conversion code, inefficiencies in lookup table blocks, and expensive comparison code.

#### Results and Recommended Actions

| Conditions | Recommended Action |
| --- | --- |
| Integer division generated code is large. | In the Configuration Parameters dialog box, on the **Hardware Implementation** pane, set the **Signed integer division rounds to** (Simulink) parameter to the recommended value. |
| Lookup Table vector of input values is not evenly spaced. | If breakpoint data is nontunable, adjust the data to have even, power of 2 spacing. See `fixpt_look1_func_approx`. |
| Lookup Table vector of input values is not evenly spaced when quantized, but it is very close to being evenly spaced. | If breakpoint data is nontunable, adjust the data to have even, power of 2 spacing. See `fixpt_evenspace_cleanup`. |
| Lookup Table vector of input values is evenly spaced, but the spacing is not a power of 2. | If breakpoint data is nontunable, adjust the data to have even, power of 2 spacing. See `fixpt_look1_func_approx`. |
| For a Prelookup or n-D Lookup Table block, **Index search method** is `Evenly spaced points`. Breakpoint data does not have power of 2 spacing. | If breakpoint data is nontunable, adjust the data to have even, power of 2 spacing. Otherwise, in the block parameter dialog box, specify a different **Index search method** to avoid the computation-intensive division operation. |
| n-D Lookup Table breakpoint data is not evenly spaced and **Index search method** is not `Evenly spaced points`. | If breakpoint data is nontunable, adjust the data to have even, power of 2 spacing and then set **Index search method** to `Evenly spaced points`. |
| n-D Lookup Table breakpoint data is evenly spaced and **Index search method** is `Evenly spaced points`. But the spacing is not a power of 2. | If breakpoint data is nontunable, adjust the data to have even, power of 2 spacing. See `fixpt_look1_func_approx`. |
| n-D Lookup Table breakpoint data is evenly spaced, but the spacing is not a power of 2. Also, **Index search method** is not `Evenly spaced points`. | Set **Index search method** to `Evenly spaced points`. Also, if the data is nontunable, consider an even, power of 2 spacing. |

| Conditions | Recommended Action |
| --- | --- |
| n-D Lookup Table breakpoint data is evenly spaced, and the spacing is a power of 2. But the **Index search method** is not `Evenly spaced points`. | Set **Index search method** to `Evenly spaced points`. |
| Blocks require multiword operations in generated code. | Adjust the word lengths of inputs to operations so that they do not exceed the largest word size of your processor. For more information, see "Fixed-Point Multiword Operations In Generated Code" (Fixed-Point Designer). |
| Blocks require cumbersome multiplication. | Restrict multiplication operations:<br>• So the product integer size is not larger than the target integer size.<br>• To the recommended size. |
| Product blocks are multiplying signals with mismatched slope adjustment factors. | Change the scaling of the output so that its slope adjustment factor is the product of the input slope adjustment factors. |
| Blocks multiply signals with nonzero bias. | Insert a Data Type Conversion block before and after the block containing the multiplication operation. |
| The inputs of the Relational Operator block have different data types. | • Change the data type and scaling of the invariant input to match other inputs.<br>• Insert Data Type Conversion blocks before the Relational Operator block to convert both inputs to a common data type. |
| The inputs of the Relational Operator block have different slope adjustment factors. | Change the scaling of either input. |
| The output of the Relational Operator block is constant. This might result in dead code which will be eliminated by Simulink Coder. | Review your model design and either remove the Relational Operator block or replace it with the constant. |

#### Capabilities and Limitations

- A Fixed-Point Designer license is required to generate fixed-point code.

- If you have a Simulink Check license, you can exclude blocks and charts from this check.

### See Also

- 1-D Lookup Table
- n-D Lookup Table
- Prelookup
- “Identify Questionable Fixed-Point Operations” (Fixed-Point Designer)
- “What Is a Model Advisor Exclusion?” (Simulink Check)

## Identify blocks that generate expensive rounding code

**Check ID**: `mathworks.codegen.ExpensiveSaturationRoundingCode`

Check for blocks that generate expensive rounding code.

### Description

Generated rounding code is inefficient because of **Integer rounding mode** parameter setting.

Available with Embedded Coder.

### Results and Recommended Actions

| Condition | Recommended Action |
|---|---|
| Generated code is inefficient. | Set the **Function Block Parameters > Integer rounding mode** parameter to the recommended value. |

### Capabilities and Limitations

If you have a Simulink Check license, you can exclude blocks and charts from this check.

### See Also

- “Identify Blocks that Generate Expensive Rounding Code” (Fixed-Point Designer)
- “What Is a Model Advisor Exclusion?” (Simulink Check)

## Check for bitwise operations on signed integers

**Check ID**: `mathworks.misra.CompliantCGIRConstructions`

Identify Simulink blocks that contain bitwise operations on signed integers. The check does not flag MATLAB Function or Stateflow blocks that use signed operands for bitwise operators.

### Description

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications, as well as code that complies with the CERT C and CWE standards.

Available with Embedded Coder and Simulink Check.

This check requires a Stateflow license.

### Results and Recommended Actions

| Condition | Recommended Action |
| --- | --- |
| The model has blocks that contain bitwise operations on signed integers. | Consider using unsigned integers for bitwise operations. |

### Capabilities and Limitations

You can:

- The check assumes that code is generated for the whole model. When code is generated by a subsystem build or export functions, the check can product incorrect results.
- Exclude blocks and charts from this check if you have a Simulink Check license.

### See Also

- MISRA C:2012, Rule 10.1
- CERT C, INT13-C
- CWE, CWE-682
- “hisl_0060: Configuration parameters that improve MISRA C:2012 compliance” (Simulink)

- “MISRA C:2012 Compliance Considerations” (Simulink)
- “Secure Coding Standards”

## Check for recursive function calls

**Check ID**: `mathworks.misra.RecursionCompliance`

Identify recursive function calls in Stateflow charts.

### Description

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications. The check flags charts that have recursive function calls.

Available with Embedded Coder and Simulink Check.

### Results and Recommended Actions

| Condition | Recommended Action |
| --- | --- |
| Chart has a recursive function call. | Remove recursive function call. |

### See Also

- MISRA C:2012, Dir 17.2
- “Guidelines for Avoiding Unwanted Recursion in a Chart” (Stateflow)

## Check for equality and inequality operations on floating-point values

**Check ID**: `mathworks.misra.CompareFloatEquality`

Identify equality and inequality operations on floating-point values.

### Description

The check flags sources causing equality or inequality operations on floating-point values.

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications, as well as code that complies with the CERT C and CWE standards.

The check does not flag blocks with equality or inequality operations on floating-point values if they are justified with a Polyspace annotation. When you run the check, the **Blocks with justification** table lists blocks with equality or inequality operations that have a justification.

Available with Embedded Coder and Simulink Check.

#### Results and Recommended Actions

| Condition | Recommended Action |
| --- | --- |
| Model object has an equality or inequality operation on a floating-point value. | Consider using non-floating-point values for equality or inequality operations. |

#### Capabilities and Limitations

You can:

- Exclude blocks and charts from this check if you have a Simulink Check license.

#### See Also

- MISRA C:2012, Dir 1.1
- CERT C, FLP00-C
- CWE, CWE-697
- “Annotate Code and Hide Known or Acceptable Results” (Polyspace Bug Finder)
- “Secure Coding Standards”

## Check integer word length

**Check ID**: `mathworks.misra.IntegerWordLengths`

Identify integer word lengths that do not comply with hardware implementation settings

#### Description

The check flags integers whose word lengths exceed the number of bits permitted via the hardware implementation settings.

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications, as well as code that complies with the CERT C and CWE standards.

Available with Embedded Coder and Simulink Check.

#### Results and Recommended Actions

| Condition | Recommended Action |
| --- | --- |
| Model object contains integer word lengths that are not compliant with hardware implementation settings. | Update the integer so its length does not exceed the permitted number of bits. You can view the permitted number of bits in the Configuration Parameters dialog box, on the **Hardware Implementation > Device details** pane. |

#### Capabilities and Limitations

You can:

- Exclude blocks and charts from this check if you have a Simulink Check license.

#### See Also

- MISRA C:2012, Rule 10.1
- CERT C, INT13-C
- CWE, CWE-682
- “MISRA C Guidelines”
- “What Is a Model Advisor Exclusion?” (Simulink Check)
- “Secure Coding Standards”

## Check block names

**Check ID**: `mathworks.codegen.BlockNames`

Checks whether block names in the **Code Perspective** pane include invalid characters.

### Description

This edit-time check evaluates the block names in the **Code Perspective** pane. The check reports invalid characters in block names, except for:

- Blocks that are ignored or not recommended for code generation
- Virtual Subsystem blocks

The check verifies that the names of all blocks comply with these guidelines:

**Form**:

*name*:

- Does not start with a number
- Does not include spaces at the beginning of a block name
- Does not use double byte characters
- Carriage returns are allowed

**Allowed Characters**:

*name*:

a b c d e f g h i j k l m n o p q r s t u v w x y z

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

0 1 2 3 4 5 6 7 8 9 _

### Results and Recommended Actions

| Condition | Recommended Action |
| --- | --- |
| The block name in the **Code Perspective** pane does not conform to the guidelines. | Update the block name to comply with the guidelines. |

### Capabilities and Limitations

- Runs on library models.

- Analyzes content of library-linked blocks.
- Analyzes content in all masked subsystems.
- Allows exclusions of blocks and charts.

### See Also

- “Simulink Built-In Blocks That Support Code Generation”

# 16

# Tools in Embedded Coder—Alphabetical List

# Embedded Coder Dictionary

Create code definitions, which control code generation for model data and functions

## Description

The Embedded Coder Dictionary is a graphical interface for creating custom code definitions. By applying these definitions in models, you and your users can generate code that conforms to a specific software architecture by default. For example, you can create your own storage class, which you and your users can apply by default to a category of model data such as root-level inputs.

You can create these types of code definitions:

- Storage classes, which control the code generated for model data.
- Function customization templates, which control naming of model entry-point functions, such as `model_step`. The templates also apply memory sections to the entry-point functions.
- Memory sections, which control the placement of data and functions in memory. The generated code includes custom decorations, such as pragmas, whose syntax you specify.

The Embedded Coder Dictionary has a tab for each type of code definition. In each tab, you configure the properties of code definitions. Use the table to configure properties and compare definitions side by side. To access properties that do not appear in the table, use the Property Inspector.

The definitions that you create in the dictionary can be applied to model elements by configuring model-wide default settings in the Code Mapping Editor (see “Configure Default Code Generation for Categories of Model Data and Functions”). To create storage classes and memory sections that you can use outside of the Code Mapping Editor, use the Custom Storage Class Designer (see “Create Code Definitions to Override Default Settings”).

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

# Open the Embedded Coder Dictionary

- To open the Embedded Coder Dictionary that a model uses, use one of these techniques:

  - In the Code Mapping Editor (see Code Mapping Editor), click the Embedded Coder Dictionary icon .
  - In a model window, select **Code** > **C/C++ Code** > **Embedded Coder Dictionary**.

  If the model is not linked to a Simulink data dictionary (see "What Is a Data Dictionary?" (Simulink)), the Embedded Coder Dictionary window displays code generation definitions that are stored in the model file. If the model is linked to a data dictionary, the window displays definitions that are stored in that data dictionary or, if applicable, in a referenced dictionary.
- To open the Embedded Coder Dictionary in a Simulink data dictionary, in the Model Explorer **Model Hierarchy** pane:

  1. Under the dictionary node, select the **Embedded Coder** node.

     If you do not see the node, right-click the dictionary node and select **Show Empty Sections**.
  2. In the Dialog pane (the right pane), click **Open Embedded Coder Dictionary**.

# Examples

## Create and Verify Custom Storage Class

In a model, create a storage class that aggregates internal model data, including block states, into a structure whose characteristics you can control. Then, verify the storage class by generating code from the model.

1. Open the example model `rtwdemo_roll`.

   ```
   rtwdemo_roll
   ```
2. In the model, select **Code** > **C/C++ Code** > **Configure Model in Code Perspective**.
3. Underneath the block diagram, under **Code Mappings** > **Data Defaults**, click the Embedded Coder Dictionary icon.

Because `rtwdemo_roll` is not linked to a Simulink data dictionary, the Embedded Coder Dictionary window displays code generation definitions that are stored in the model file.

4. In the Embedded Coder Dictionary window, click the **Add** button. A new storage class named `StorageClass1` appears at the bottom of the list.
5. Select the new storage class. On the right side of the Embedded Coder Dictionary window, in the Property Inspector, set the property values listed in this table.

| Property | Value |
|---|---|
| **Name** | `InternalStruct` |
| **Storage Type** | `Structured` |
| **Header File** | `internalData_$R.h` |
| **Definition File** | `internalData_$R.c` |
| **Structure Properties** > **Type Name** | `internalData_T_$M` |
| **Structure Properties** > **Instance Name** | `internalData_$M` |

6. In the model, under **Code Mappings** > **Data Defaults**, for the **Internal data** row, set **Storage Class** to `InternalStruct`.
7. Generate code from the model.
8. In the code generation report, under **Generated Code** > **Data files**, inspect `internalData_rtwdemo_roll.h`. The file defines the structure type `internalData_T_`, whose fields represent block states in the model.

```
/* Storage class 'InternalStruct', for system '<Root>' */
typedef struct {
  real32_T FixPtUnitDelay1_DSTATE;     /* '<S7>/FixPt Unit Delay1' */
  real32_T Integrator_DSTATE;          /* '<S1>/Integrator' */
  int8_T Integrator_PrevResetState;    /* '<S1>/Integrator' */
} internalData_T_;
```

The file also declares a global structure variable named `internalData_`.

```
/* Storage class 'InternalStruct' */
extern internalData_T_ internalData_;
```

9. Inspect the file `internalData_rtwdemo_roll.c`. The file allocates memory for `internalData_`.

```
/* Storage class 'InternalStruct' */
internalData_T_ internalData_;
```

## Create Function Customization Template

With a function template, you can specify a rule that governs the names of generated entry-point functions. This technique helps save time and maintenance effort in a model with many entry-point functions, such as an export-function model or a multirate, multitasking model.

This example shows how to create a function template that specifies the naming rule `func_$N_$R`. `$N` is the base name of each generated function and `$R` is the name of the Simulink model.

1. Open the example model `rtwdemo_mrmtbb`.
2. Update the block diagram. This multitasking model has two execution rates, so the generated code includes two corresponding entry-point functions.
3. In the model, set **Configuration Parameters > Code Generation > System target file** to `ert.tlc`. To use a function customization template, you must use an ERT-based system target file.
4. In the model, enter the Code Perspective and open the Embedded Coder Dictionary.
5. In the Embedded Coder Dictionary, on the **Function Customization Templates** tab, click the **Add** button.
6. For the new function template, set these properties:

   - **Name** to `myFunctions`.
   - **Function Name** to `func_$N_$R`.
7. In the Code Mapping Editor for the model, on the **Function Defaults** tab, for the **Initialize/Terminate** and **Execution** rows, set **Function Customization Template** to `myFunctions`.
8. Generate code from the model.
9. In the code generation report, inspect `rtwdemo_mrmtbb.c`. The file defines the two execution functions, `func_step_rtwdemo_mrmtbb0` and `func_step_rtwdemo_mrmtbb1`, whose names conform to the rule that you specified in the function template.

## Create Memory Section

For an example that shows how to create a memory section, see “Control Data and Function Placement in Memory by Inserting Pragmas”.

## Create Storage Class to Use with Parameter and Nonparameter Data

This example shows how to create a storage class that places global variable definitions and declarations in files whose names depend on the model name. You create two copies of the storage class so that you can use one copy with parameter data (the data category **Local parameters**) and the other copy with nonparameter data.

Typically, the generated code initializes parameter data statically, outside any function, and initializes nonparameter data dynamically, in the model initialization function. When you create a storage class by using the Custom Storage Class Designer or an Embedded Coder Dictionary, you set the **Data Initialization** property to specify the initialization mechanism.

In an Embedded Coder Dictionary, for each storage class, you must select either `Static` or `Dynamic`. Consider creating one copy of the storage class for parameter data (`Static`) and one copy for nonparameter data (`Dynamic`).

1. Open the example model `rtwdemo_roll`.

   ```
   rtwdemo_roll
   ```
2. In the model, select **Code** > **C/C++ Code** > **Configure Model in Code Perspective**.
3. Underneath the block diagram, under **Code Mappings**, click the Embedded Coder Dictionary icon.
4. In the Embedded Coder Dictionary, click the **Add** button.
5. For the new storage class, set these properties:

   - **Name** to `SigsStates`
   - **Header File** to `$R_my_data.h`
   - **Definition File** to `$R_my_data.c`

   By default, the **Data Initialization** property is set to `Dynamic`, which means the storage class is suitable for use with nonparameter data.
6. Click the **Duplicate** button. A new storage class, `SigsStates_copy`, appears.
7. For the new storage class, set these properties:

   - **Name** to `Params`
   - **Data Initialization** to `Static`

1. In the model, under **Code Mappings > Data Defaults**, for the **Local parameters** row, in the **Storage Class** column, select `Params`.
2. For the **Internal data** row, set **Storage Class** to `SigsStates`.
3. Configure some parameter data elements in the model so that optimizations do not eliminate them from the generated code. Underneath the block diagram, open the Model Data Editor by selecting the **Model Data Editor** tab.
4. Select the **Parameters** tab.
5. In the model, navigate into the `BasicRollMode` subsystem.
6. Update the block diagram. Now, the data table contains rows that correspond to workspace variables used by the model.
7. Next to the **Filter contents** box, activate the **Filter using selection** button.
8. In the model, select the three Gain blocks.
9. In the Model Data Editor, in the data table, select the three rows that correspond to the `dispGain`, `intGain`, and `rateGain` variables in the model workspace.
10. For each variable, in the **Storage Class** column, select `Convert to parameter object`.

    The Model Data Editor converts the workspace variables to `Simulink.Parameter` objects. The new objects use the storage class `Model default`, which means they acquire the default storage class that you specified for **Local parameters** in the Code Mapping Editor.
11. Generate code from the model.
12. In the code generation report, inspect the files `rtwdemo_roll_my_data.c` and `rtwdemo_roll_my_data.h`. The files define and declare global variables that correspond to the parameter objects and some block states, such as the state of the Integrator block in the `BasicRollMode` subsystem.

    ```
    /* Storage class 'SigsStates' */
    real32_T rtwdemo__FixPtUnitDelay1_DSTATE;
    real32_T rtwdemo_roll_Integrator_DSTATE;
    int8_T rtwde_Integrator_PrevResetState;

    /* Storage class 'Params' */
    real32_T dispGain = 0.75F;
    real32_T intGain = 0.5F;
    real32_T rateGain = 2.0F;
    ```

### Share Code Generation Definitions Between Models by Using Simulink Data Dictionary

For an example that shows how to share code generation definitions between models by using data dictionaries, see “Share Embedded Coder Dictionary Definition Between Models”.

### Refer to Code Generation Definitions in a Package

You can configure an Embedded Coder Dictionary to refer to code generation definitions that you store in a package (see “Create Code Definitions to Override Default Settings”). Those definitions then appear available for selection in the Code Mapping Editor. In this example, you configure the Embedded Coder Dictionary in `rtwdemo_roll` to refer to definitions stored in the built-in example package `ECoderDemos`.

1. Open the Embedded Coder Dictionary for `rtwdemo_roll`. For instructions, see “Create and Verify Custom Storage Class” on page 16-3.
2. In the Embedded Coder Dictionary window, click the **Manage Package** button.
3. In the Manage Package dialog box, click **Refresh**. Wait until more options appear in the **Change package** drop-down list.
4. Set **Change package** to `ECoderDemos` and click **Load Package**.

   In the Embedded Coder Dictionary window, on the **Storage Classes** tab, the table shows the storage classes defined in the `ECoderDemos` package. Now, in `rtwdemo_roll`, you can select these storage classes under **Code Mappings** > **Data Defaults**.

## Parameters

These properties appear in the Property Inspector on the right side of the Embedded Coder Dictionary window. In the table in the middle of the window, some properties appear as columns to facilitate batch editing.

#### Storage Classes

#### Name — Name of storage class

`StorageClass1` (default) | text

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

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

#### `Source` — Location of storage class definition

text

This property is read-only.

The location of the storage class definition.

- `Built-in` — Provided by Simulink.
- Model name — Defined in a Simulink model.
- Dictionary name — Defined in a Simulink data dictionary (see “What Is a Data Dictionary?” (Simulink)).
- Package name — Defined in the Simulinkpackage or in a custom package (see “Create Custom Storage Classes by Using the Custom Storage Class Designer”).

#### `Description` — Purpose and functionality of storage class

text

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

#### `Memory Section` — Location in memory to allocate data

`None` (default) | existing memory section

Location in memory to allocate data, specified as a memory section that exists in the Embedded Coder Dictionary on the **Memory Sections** tab. For information about memory sections, see “Control Data and Function Placement in Memory by Inserting Pragmas”.

#### `Data Scope` — Specification to generate data definition

`Exported` (default) | `Imported`

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

##### Dependencies

- Setting this property to `Imported`:

  - Disables **Definition File**. To include your external source code file in the build process, use model configuration parameters. For an example, see “Customize Interfaces of Generated Entry-Point Functions”.
  - Means that you cannot set **Header File** to `$N.h`, though you can use the `$N` token.
- To set this property to `Exported`, you must use one of the tokens `$N` or `$R` in the value of **Header File**.

### `Data Initialization` — How to initialize data

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

Specification that the generated code initialize the data.

- `Dynamic` — The generated code initializes the data as part of the model initialization entry-point function.
- `Static` — The generated code initializes the data in the same statement that defines and allocates memory for the data. The assignment statement appears at the top of a `.c` or `.cpp` source file, outside of a function.
- `None` — The generated code does not initialize the data.

#### Dependencies

- If you select **Const**, you cannot set this property to `Dynamic`.
- Setting this property to `Dynamic` disables **Const**.

### `Header File` — Name of header file that declares data

`$N.h` (default) | text

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

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

#### Dependencies

- If you set **Data Scope** to `Exported`, you must use one of the tokens `$R` or `$N` in the value of this property.
- If you set **Data Scope** to `Imported`, you cannot set the value of this property to `$N.h`, but you can use the `$N` token.

### `Definition File` — Name of source file that defines data

`$N.c` (default) | text

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

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

#### Dependencies

Setting **Data Scope** to `Imported` disables **Definition File**. To include your external source code file in the build process, use model configuration parameters. For an example, see “Customize Interfaces of Generated Entry-Point Functions”.

### `Const` — Specification to apply `const` qualifier

`off` (default) | `on`

Specification to apply the `const` qualifier to the data.

#### Dependencies

- If you select this property, you cannot set **Data Initialization** to `Dynamic`.
- Setting **Data Initialization** to `Dynamic` disables this property.

### `Volatile` — Specification to apply `volatile` qualifier

`off` (default) | `on`

Specification to apply the `volatile` qualifier to the data.

### `Other Qualifier` — Specification to apply a custom qualifier

text

Specification to apply a custom qualifier to the data. For example, some memory architectures support qualifiers `far` and `huge`.

Do not use this property to apply the keyword `static`. Instead, use the built-in storage class `FileScope`, which you cannot apply with the Code Mapping Editor. See “Choose a Storage Class for Controlling Data Representation in the Generated Code”.

### `Storage Type` — Specification to aggregate data into a structure

`Unstructured` (default) | `Structured`

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

#### Dependencies

Setting this property to `Structured` enables **Type Name** and **Instance Name**.

### `Type Name` — Name of structure type

`$R$N$G$M` (default) | text

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

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

#### Dependencies

Setting **Storage Type** to `Structured` enables this property.

### `Instance Name` — Name of structure variable
`$N$G$M` (default) | text

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

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

#### Dependencies

Setting **Storage Type** to `Structured` enables this property.

#### Function Customization Templates

### `Name` — Name of function template
`FunctionTemplate1` (default) | text

Name of the template. The name must be unique among the function templates in the dictionary. Embedded Coder provides the built-in templates listed in this table.

| Template | Description |
| --- | --- |
| `ModelFunction` | In the Code Mapping Editor, use for entry-point functions for initialization, execution, termination, and reset (see “Configure Default Code Generation for Functions”) |
| `UtilityFunction` | In the Code Mapping Editor, use for shared utility functions (see “Configure Default Code Generation for Functions”) |

### `Function Name` — Names of generated functions
`$R$N` (default) | text

Names of the functions in the generated code, specified as a naming rule. A naming rule includes a combination of text and tokens. Valid tokens are listed in this table.

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

### `Source` — Location of function template definition
text

This property is read-only.

The location of the function template definition.

- Model name — Defined in a Simulink model.
- Dictionary name — Defined in a Simulink data dictionary (see “What Is a Data Dictionary?” (Simulink)).

### `Description` — Purpose and functionality of function template
text

Custom text that you can use to describe the purpose and functionality of the function template.

#### Memory Sections

### `Name` — Name of memory section
text

Name of the memory section. The name must be unique among the memory sections in the dictionary. Embedded Coder provides the built-in memory sections listed in this table.

| Memory Section | Description |
|---|---|
| `MemConst` | Apply the storage type qualifier `const` to the data. |
| `MemVolatile` | Apply the storage type qualifier `volatile` to the data. |

| Memory Section | Description |
|---|---|
| `MemConstVolatile` | Apply the storage type qualifiers `const` and `volatile` to the data. |

**`Source` — Location of memory section definition**
text

This property is read-only.

The location of the memory section definition.

- Model name — Defined in a Simulink model.
- Dictionary name — Defined in a Simulink data dictionary (see “What Is a Data Dictionary?” (Simulink)).
- Package name — Defined in the Simulinkpackage or in a custom package (see “Create Code Definitions to Override Default Settings”).

**`Description` — Purpose and functionality of memory section**
text

Custom text that you can use to describe the purpose and functionality of the memory section.

**`Comment` — Comment to insert in the generated code**
text

Code comment that the code generator includes with the pragmas or other decorations that you specify with **Pre Statement** and **Post Statement**.

**`Pre Statement` — Code to insert before data or function code**
text

Code, such as pragmas, to insert before the definitions and declarations of the data or functions that are in the memory section.

When you set **Statements Surround** to `Each variable`, you can use the token `$N` to represent the name of each variable or function that uses the memory section.

**`Post Statement` — Code to insert after data or function code**
text

Code, such as pragmas, to insert after the definitions and declarations of the data or functions that are in the memory section.

When you set **Statements Surround** to `Each variable`, you can use the token `$N` to represent the name of each variable or function that uses the memory section.

#### `Statements Surround` — Specification to wrap data and functions separately or in a group

`Each variable` (default) | `Group of variables`

Specification to insert code statements (**Pre Statement** and **Post Statement**):

- Around each variable and function that uses the memory section. Select `Each variable`.
- Once, around the entire memory section. The generated code aggregates the variable and function definitions into a contiguous code block and surrounds the block with the statements. Select `Group of variables`.

## Limitations

- You cannot create or modify code generation definitions programmatically. However, you can delete, copy, and move code definitions between models and data dictionaries by using these functions:
  - `coder.dictionary.copy`
  - `coder.dictionary.move`
  - `coder.dictionary.remove`
- A storage class or function template that you create in an Embedded Coder Dictonary cannot use a memory section that you load from a package (as described in “Refer to Code Generation Definitions in a Package” on page 16-8). You must use an Embedded Coder Dictionary memory section.
- You cannot create code generation definitions in a `.mdl` model file.
- For additional limitations for code generation definitions in the Embedded Coder Dictionary of a data dictionary (`.sldd` file), see “Limitations for Definitions in a Simulink Data Dictionary”.

## See Also

**Introduced in R2018a**

# Code Mapping Editor

Associate model data elements and entry-point functions with code definitions

## Description

The Code Mapping Editor is a graphical interface for configuring model data elements and entry-point functions for code generation. Associate each category of model data element with a specific storage class throughout a model. A storage class defines properties (for example, appearance and location) that the code generator uses when producing code for associated data. Similarly, associate each category of model entry-point function with a specific function customization template. The templates define how the code generator produces code for associated functions. If necessary, override default mappings for specific data elements or functions by using the **Code** view of the Model Data Editor or function prototype control.

The Code Mapping Editor display consists of two tabbed tables: **Data Defaults** and **Function Defaults**. Use the tables to set default code definitions for categories of model data elements and functions. The **Code** section of the Property Inspector shows your selection and whether a memory section is defined for the storage class or function customization template.

## Open the Code Mapping Editor

- To prepare a model for code generation, use Embedded Coder Quick Start. Quick Start places your model in Code Perspective mode, which includes the Code Mapping Editor.
- In the model window, click the perspective control in the lower-right corner and select **Code**.
- In the model window, select **Code > C/C++ Code > Configure Model in Code Perspective**.
- If you close the Code Mapping Editor, The Model Editor window remains in Code Perspective mode. To reopen the Code Mapping Editor, select **View > Code Mappings**.

# Examples

## Configure Code Generation for Root Inports and Outports

This example shows how to configure code generation for the root Inport and Outport blocks throughout a model. Applying default configurations can save time, especially for large-scale models that use a significant amount of data. After applying default mappings, as necessary, you can adjust mappings for individual data elements by using the Model Data Editor.

1 Copy external code files into your current MATLAB folder.

```
copyfile(fullfile(matlabroot,'toolbox','rtw','rtwdemos','input_data.c'));
copyfile(fullfile(matlabroot,'toolbox','rtw','rtwdemos','input_data.h'));
copyfile(fullfile(matlabroot,'toolbox','rtw','rtwdemos','heading_mode.c'));
copyfile(fullfile(matlabroot,'toolbox','rtw','rtwdemos','heading_mode.h'));
```

2 Open model `rtwdemo_roll`.

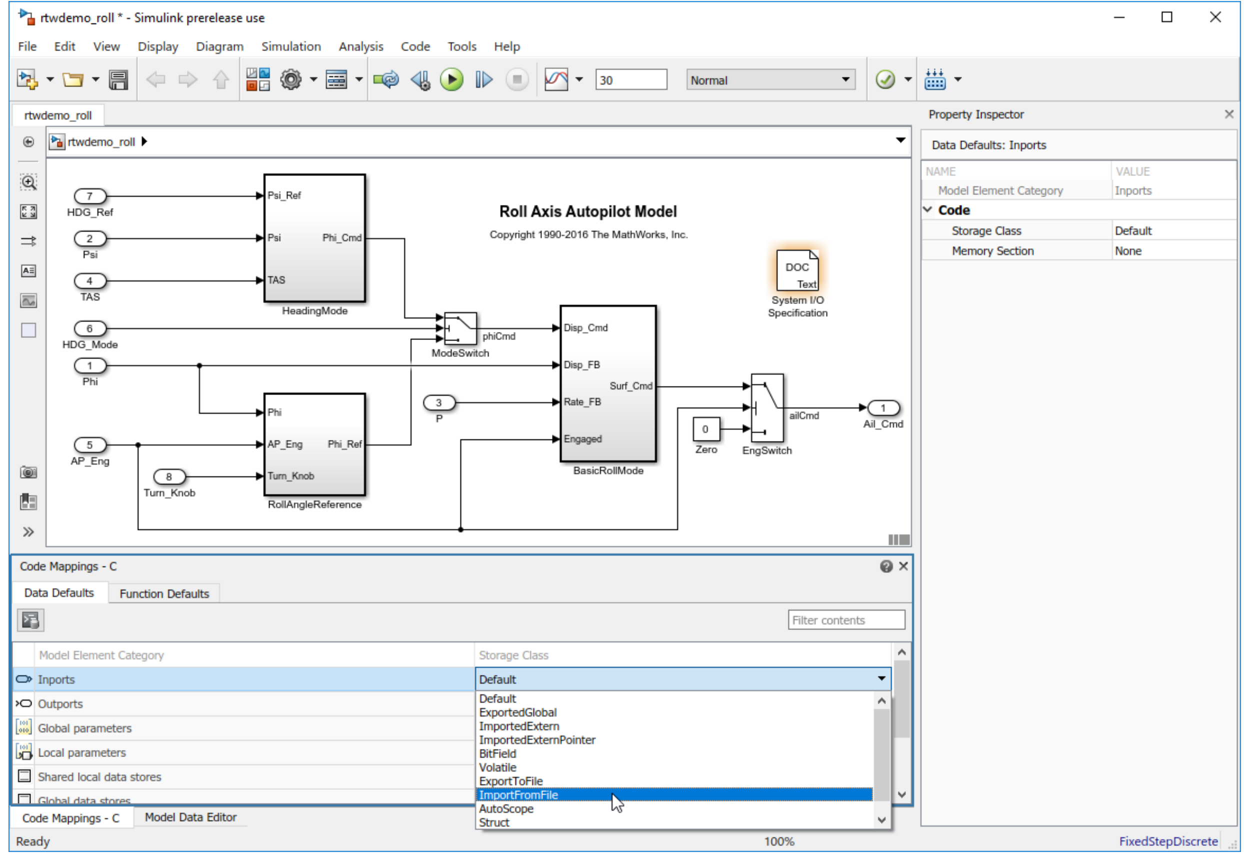
3 Open Code Perspective by selecting **Code> C/C++ Code > Configure Model in Code Perspective**.

Configure the code generator to:

- Use header file `input_data.h` to declare the variables representing model Inport blocks.
- Represent variables for model Outport blocks as separate global variables.
- Define output variables in `output_data.c` and declare them in `output_data.h`.

1 Open the Code Mapping Editor. Under **Code Mappings - C**, click **Data Defaults** tab if not already selected.

2 Set the storage class for model element category **Inports** to `ImportFromFile`.



3 In the Property Inspector, set **Header File** to `input_data.h`.

4 Set the storage class for model element category **Outports** to `ExportToFile`.

5 In the Property Inspector, set **Header File** to `output_data.h` and **Definition File** to `output_data.c`.

Override the default source location for inport variable `HDG_Mode`. That variable is declared in the external file `heading_mode.h`.

1 Open the Model Data Editor by clicking the **Model Data Editor** tab.

2 Select the `HDG_Mode` row.

3 Set **Storage Class** to `ImportFromFile`.

**4** Set **Header File** to `heading_mode.h`.

**1** Configure the code generator to produce variable names in the code for Inport and Outport blocks that match the variable names in external files `input_data.h` and `heading_mode.h`. Set the model configuration parameter **Global variables** to `$N $M`, removing the `rt` prefix that the code generator applies by default.

**2** Include external source files `input_data.c` and `heading_mode.c` in the code generation and build process. Set the model configuration parameter **Source files** to `input_data.c heading_mode.c`.

Generate code and verify that the code generated for Inport and Output blocks is correct.

- `rtwdemo_roll.h` includes three header files associated with storage classes:

```
#include "output_data.h"
#include "heading_mode.h"
#include "input_data.h"
```

- `heading_mode.c` includes `heading_mode.h` and defines variable `HDG_Mode`.

```
#include "heading_mode.h"
boolean_T HDG_Mode;
```

- `input_data.c` defines the variables declared in `input_data.h`.

```
#include "input_data.h"

boolean_T AP_Eng;
real32_T HDG_Ref;
real32_T Rate_FB;
real32_T Phi;
real32_T Psi;
real32_T TAS;
real32_T Turn_Knob;
```

- `output_data.c` includes this exported data definition:

```
real32_T Ail_Cmd;
```

- `output_data.h` includes this exported data declaration:

```
extern real32_T Ail_Cmd;
```

## Configure Default Function Names for Entry-Point Functions

By default, the code generator uses the identifier naming rule `$R$N` to name entry-point functions. `$R` is the name of the root model. `$N` is the name of the function, for example, `initialize`, `step`, and `terminate`. To intergrate generate code with existing external code or to comply with naming standards or guidelines, you can adjust the default naming rule. This example shows how to add the text string `myproj_` as a prefix to `$R$N`. Adjusting the default naming rule can save time, especially for multirate models for which the code generator produces a unique `step` function for each rate.

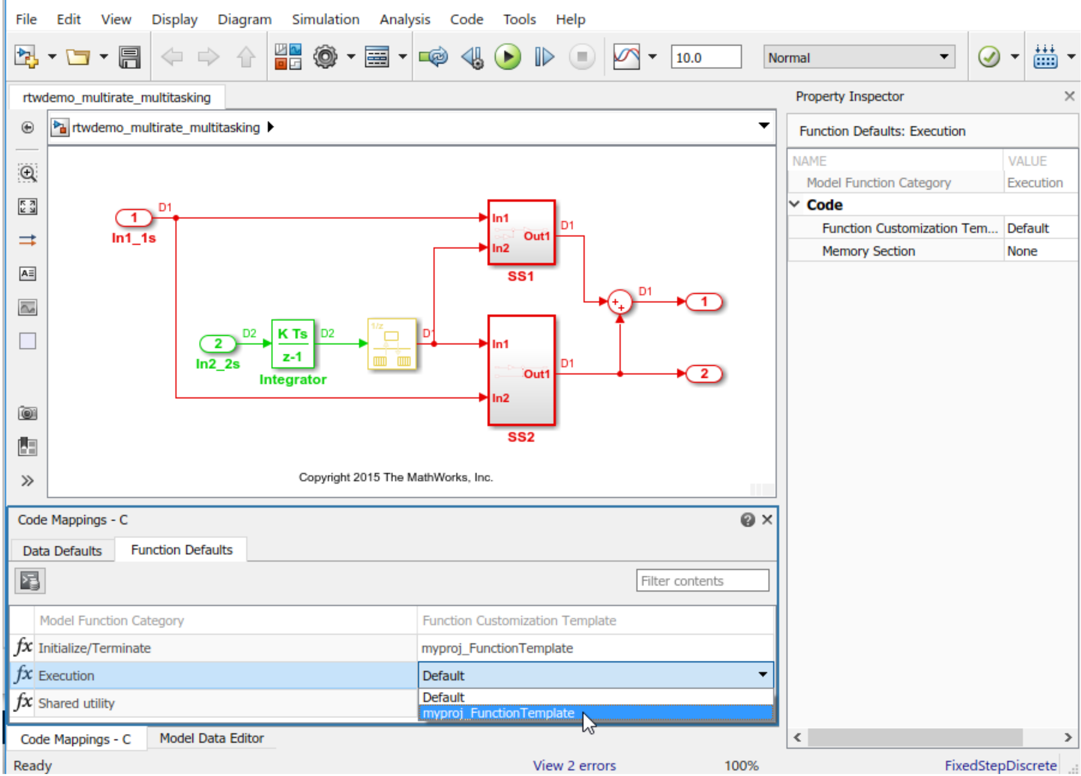
Open model `rtwdemo_multirate_multitasking` and save a copy to a writable location.

Create a function customization template that defines the naming rule `myproj_$R$N`.

1. In the model window, open the Embedded Coder Dictionary by clicking `Code>C/C++ Code>Embedded Coder Dictionary`.
2. Click the **Function Customization Templates** tab.
3. Click **Add**.
4. In the **Name** column of the new table row, name the new template`myproj_FunctionTemplate`.
5. In the **Function Name** column, enter the naming rule `myproj_$R$N`.
6. Close the coder dictionary.

1. Open Code Perspective by selecting **Code> C/C++ Code > Configure Model in Code Perspective**.

2 For the **Initialize/Terminate** and **Execution** function categories, change the default function customization template from `Default` to `myproj_FunctionTemplate`.




Generate code and verify the entry-point function names.

```
void myproj_rtwdemo_multirate_multitasking_step0(void) /* Sample time: [1.0s, 0.0s] */
{
  (rtM->Timing.RateInteraction.TID0_1)++;
  if ((rtM->Timing.RateInteraction.TID0_1) > 1) {
       rtM->Timing.RateInteraction.TID0_1 = 0;
  }

  if (rtM->Timing.RateInteraction.TID0_1 == 1) {
      rtDW.RateTransition = rtDW.RateTransition_Buffer0;
  }
      rtY.Out2 = 2.0 * rtDW.RateTransition + rtU.In1_1s;
      rtY.Out1 = (3.0 * rtDW.RateTransition + rtU.In1_1s) * 5.0 + rtY.Out2;
}

/* Model step function for TID1 */
void myproj_rtwdemo_multirate_multitasking_step1(void) /* Sample time: [2.0s, 0.0s] */
```

```
{
  rtDW.RateTransition_Buffer0 = rtDW.Integrator_DSTATE;
  rtDW.Integrator_DSTATE += 2.0 * rtU.In2_2s;
}

void myproj_rtwdemo_multirate_multitasking_initialize(void)
{
  /* (no initialization code required) */
}

void myproj_rtwdemo_multirate_multitasking_terminate(void)
{
  /* (no terminate code required) */
}
```

# Parameters

**Data Defaults**

### `Model Element Category` — Category of model data element

character vector

Names a category of Simulink model 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. |
| Outports | Root-level output ports of a model. |
| Global parameters | Parameters that are defined in the base workspace or in a data dictionary. Multiple models in an application can use these parameters. |
| Local parameters | Parameters that are defined within a model, such as parameters in the model workspace. Excludes model arguments. |
| 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. |
| Shared local data stores | Data Store Memory blocks with 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. |

| Model Element Category | Description |
| --- | --- |
| Internal data | Local data, such as data stores, discrete block states, block output signals, and zero-crossing signals. |
| Constants | Constant-value block output and constant parameters in a model. |

**`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.

**Function Defaults**

**`Model Function Category` — Category of model functions**
character vector

Names a category of Simulink model functions. The function customization template that you set for a category applies to functions in that category throughout the model.

| Model Function Category | Description |
| --- | --- |
| Initialize/Terminate | Entry-point functions for initialization and termination |
| Execution | Entry-point functions for initiatiating exeuction and resets |
| Shared utility | Shared utility functions |

**`Function Customization Template` — Code definition for functions**
character vector

Definition (specification) that the code generator uses to determine properties, such as appearance and location, for code that it produces for model functions.

## See Also

Embedded Coder Dictionary

### Topics

"Configure Default Code Generation for Categories of Model Data and Functions"
"Control Data and Function Interface in Generated Code"

"Configure Code Generation for Model Entry-Point Functions"

**Introduced in R2018a**

# 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

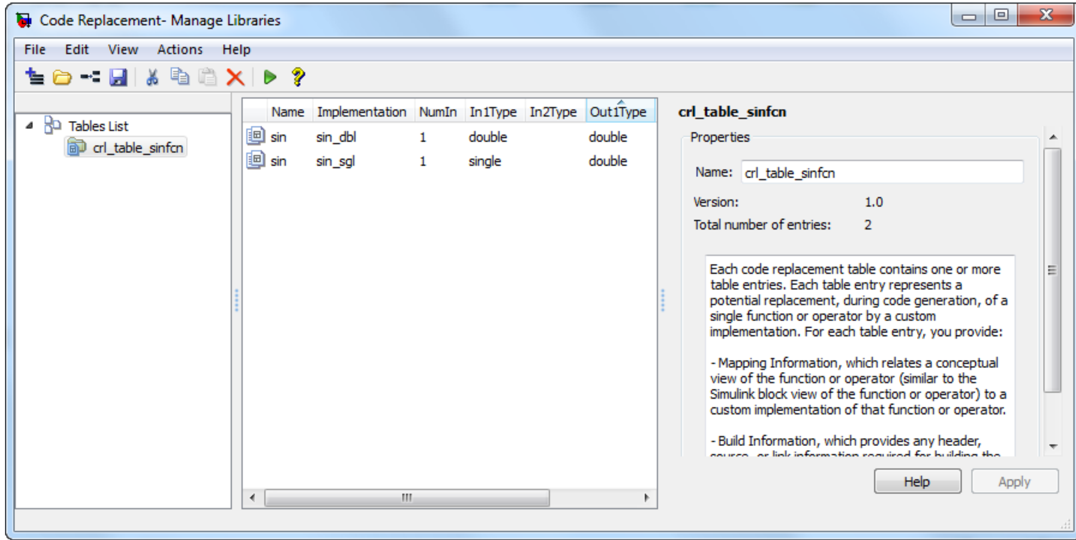
At the command prompt, type `crtool`.

## Examples

### Open an Existing Table in the Tool

This example shows how to open a code replacement table, `crl_table_sinfcn`, in the Code Replacement Tool.

```
crtool('crl_table_sinfcn')
```




- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Quick Start Code Replacement Library Development - Simulink®”

# Parameters

**Entry Summary Information (Center Pane)**

### Name — Name of table entry (read-only)

character vector

Conceptual name of the function or operation being replaced. Can name a math operation, function, BLAS operation, CBLAS operation, net slope fixed-point operation, semaphore or mutex entry, or customization entry.

### Implementation — Name of replacement function

character vector

Name of the implementation (replacement) function.

### NumIn — Number of input arguments (read-only)

scalar integer

Number of input arguments.

### `In`*n*`Type` — Data type of conceptual input argument

character vector

Data type of a conceptual input argument.

### `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**
Column-major (default) | Row-major| | Column-and-Row

Vector that specifies the lower range of the matrix dimensions.

**`Upper range` — Upper range of matrix dimensions**
[2 2] (default)

Vector that specifies the upper range of the matrix dimensions.

**`Array layout supported by entry` — Layout for array storage**
Column-major (default) | Row-major | Column-and-Row

Order in which array elements are stored in memory. Row-major layout can improve performance for certain algorithms and ease integration with external code or data that uses the row-major layout.

**`Make conceptual and implementation argument types the same` — Data type consistency**
selected (default) | cleared

Whether you want the data types for your implementation arguments to be the same as the conceptual argument types. For example, most ANSI-C functions operate on and return `double` data. Clear the check box if want to map the conceptual representation of a function or operation to an implementation representation that specifies an argument and return value. For example, clear the check box to map the conceptual representation of the function `sin` to an implementation representation that specifies an argument and return value of type `single` (`single` `sin`(`single`)).

**`Name` — Name of replacement function**
character vector

Name of the replacement function.

**`C++ namespace` — Namespace of replacement function**
character vector

Namespace of the replacement function.

**`Function returns void` — Function returns void**
selected (default) | cleared

Whether your implementation function returns `void`.

**`Function arguments` — Replacement argument names**
yn | un

Names of input and output arguments of your replacement function.

**`Data type (replacement)` — Replacement argument data type**
double (default) | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | boolean | void | integer | size_t | long | ulong | long long | ulong long | char

Data type of a selected input or output argument of your replacement function.

**`I/O type` — Replacement argument I/O type**
OUTPUT | INPUT

Whether a selected argument of your replacement function is an input or output argument.

**`Const` — Const replacement argument**
cleared (default) | selected

Whether to apply the `const` type qualifier to a selected argument of your replacement function.

**`Pointer` — Pointer replacement argument**
cleared (default) | selected

Whether a selected argument of your replacement function is a pointer.

**`Complex (replacement)` — Replacement argument complexity**
cleared (default) | selected

Whether the selected input or output argument of the replacement function is real or complex.

**`Integer saturation mode` — Saturation mode**
unspecified Saturation (default) | wrap on overflow | saturate on overflow

Saturation mode supported by the replacement function.

**`Rounding modes` — Rounding modes**
unspecified rounding (default) | floor | ceil | zero | nearest | MATLAB nearest | simplest | conv

Rounding modes supported by the replacement function.

**`Allow expressions as inputs` — Expressions as inputs**
selected (default) | cleared

Whether your replacement function accepts expression inputs. If you select the parameter, the code generator integrates an expression input into the generated code rather than inserting a temporary variable in place of the expression input.

**`Function modifies internal or global state` — State modification**
cleared (default) | selected

Whether your replacement function modifies variables representing internal or global state.

#### Entry Build Information (Right Pane)

**`Implementation header file` — Header file for replacement function**
character vector

Header file for the replacement function (for example, `my_rep_func.h`).

**`Implementation source file` — Source file for replacement function**
character vector

Source file for the replacement function (for example, `my_rep_func.c`).

**`Additional header files/include paths` — Names and paths of additional header files**
character vector

Names and paths of additional header files to include for the replacement function (for example, `support_files.h` and `matlab\customization\mylib\include`).

**`Additional source files/ paths` — Names and paths of additional source files**
character vector

Names and paths of additional source files to include for the replacement function (for example, `support_files.c` and `matlab\customization\mylib\src`).

**`Additional object files/ paths` — Names and paths of link object files**
character vector

Names and paths of link object files to use for the replacement function (for example, `support_files.o` and `matlab\customization\mylib\bin`).

**`Additional link flags` — Link flags to use**
character vector

Link flags to use for the replacement function (for example, `-MD -Gy`).

**`Additional compile flags` — Compile flags to use**
character vector

Compile flags to use for the replacement function (for example, `-Zi -Wall`).

**`Copy files to build directory` — Copy files to build folder**
cleared (default) | selected

Whether the code generator copies files from external folders to the build folder before starting the build process.

### Programmatic Use

`crtool(table)` opens the Code Replacement Tool and displays the contents of `table`, where `table` is a character vector that names a MATLAB file that defines code replacement tables. The file must be in the current folder or on the MATLAB path.

## See Also

### Topics

"What Is Code Replacement?"
"What Is Code Replacement Customization?"
"Quick Start Code Replacement Library Development - Simulink®"

**Introduced in R2014b**

# Code Replacement Viewer

Explore content of code replacement libraries

## Description

The Code Replacement Viewer displays the content of code replacement libraries. Use the tool to explore and choose a library. If you develop a custom code replacement library, use the tool to verify table entries.

- 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 that the library contains. When you select a code replacement table, the viewer displays function and operator code replacement entries that are in that table.

### 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. |

| Field | Description |
|---|---|
| **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. |
| **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:<br>`RTW_SATURATE_ON_OVERFLOW`<br>`RTW_WRAP_ON_OVERFLOW`<br>`RTW_SATURATE_UNSPECIFIED` |

| Field | Description |
| --- | --- |
| **Rounding modes** | Rounding modes that the implementation function supports. One or more of:<br>RTW_ROUND_FLOOR<br>RTW_ROUND_CEILING<br>RTW_ROUND_ZERO<br>RTW_ROUND_NEAREST<br>RTW_ROUND_NEAREST_ML<br>RTW_ROUND_SIMPLEST<br>RTW_ROUND_CONV<br>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. |

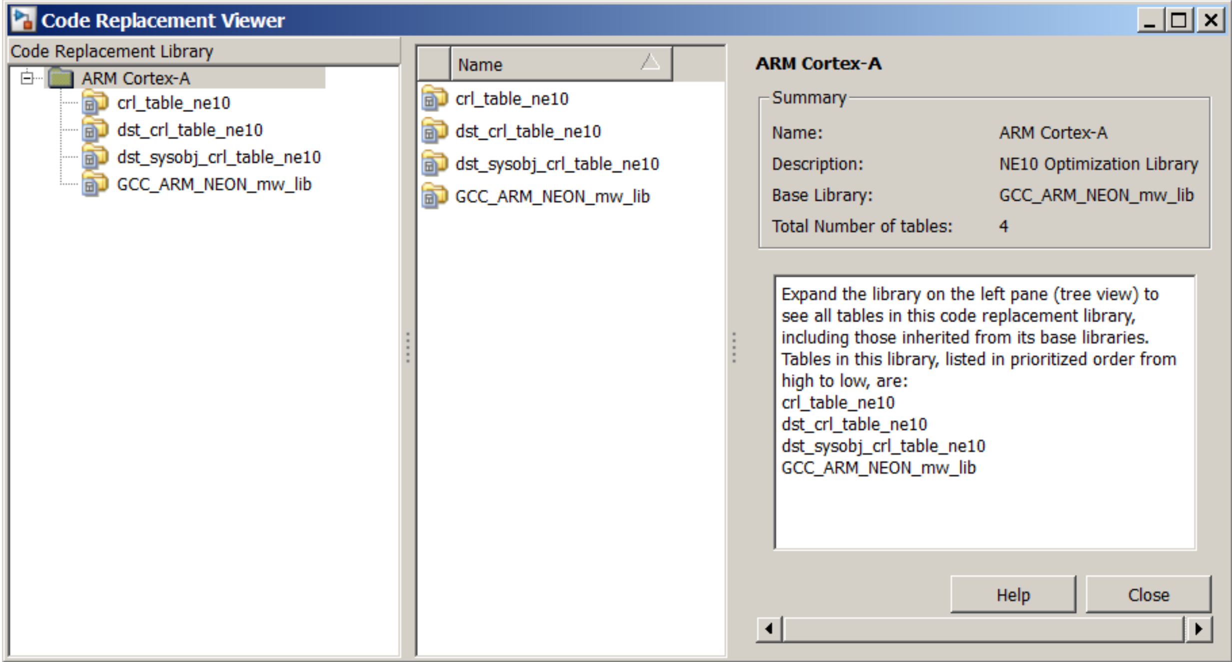
## Open the Code Replacement Viewer

Open from the MATLAB command prompt using `crviewer`.

## Examples

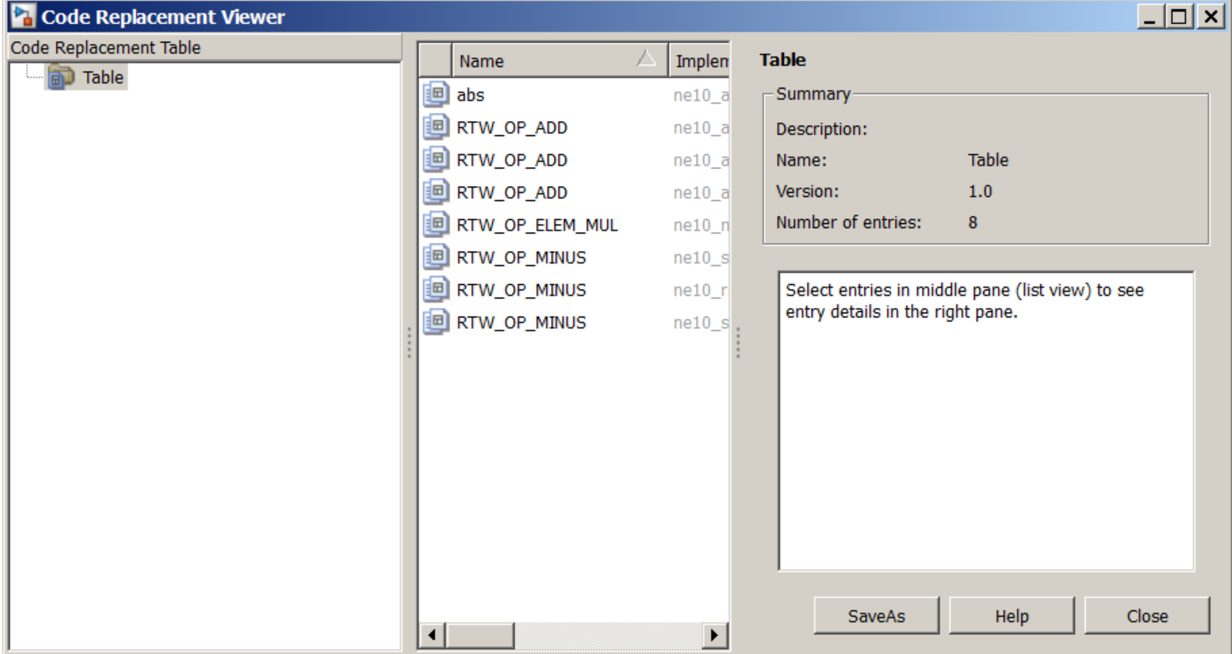
### Display Contents of Code Replacement Library

```
crviewer('ARM Cortex-A')
```




### Display Contents of Code Replacement Table

```
crviewer(clr_table_ne10)
```




- “Choose a Code Replacement Library”
- “Verify Code Replacements”

## 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. For example, `'GNU 99 extensions'`.

`crviewer(table)` opens the Code Replacement Viewer and displays the contents of `table`, where `table` is a MATLAB file that defines code replacement tables. The file must be in the current folder or on the MATLAB path.

## See Also

### Topics

“Choose a Code Replacement Library”
“Verify Code Replacements”
“What Is Code Replacement?”
“What Is Code Replacement Customization?”
“Code Replacement Libraries”
“Code Replacement Terminology”

**Introduced in R2014b**

# 17

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

# ssSetSymbolicDimsSupport

Specify whether an S-function supports symbolic dimensions

## Languages

C, C++

## Syntax

```
void ssSetSymbolicDimsSupport(SimStruct *S, const boolean_T val)
```

## Arguments

`S`

SimStruct representing an S-Function block.

`val`

Boolean value corresponding to whether the S-Function block supports symbolic dimensions.

## Returns

This function does not return a value.

## Example

Call this function from inside the `mdlInitializeSizes` function. For an example of how to use this function to configure an S-function that supports forward and backward propagation of symbolic dimensions, see “Configure Dimension Variants for S-Function Blocks”

**Introduced in R2016a**

# mdlSetInputPortSymbolicDimensions

Specify symbolic dimensions of an input port and how those dimension propagate forward

## Languages

C, C++

## Syntax

```
#if defined(MATLAB_MEX_FILE)
#define MDL_SET_INPUT_PORT_SYMBOLIC_DIMENSIONS

static void mdlSetInputPortSymbolicDimensions(SimStruct *S, int_T portIndex,
 SymbDimsId symbDimsId)

{
}
#endif
```

## Arguments

`S`

SimStruct representing an S-Function block.

`portIndex`

Index of an input port.

`symbDimsId`

Unique integer value corresponding to a symbolic dimension specification.

## Returns

This function does not return a value.

## Example

For an example of how to use this function to configure an S-function that supports forward and backward propagation of symbolic dimensions, see “Configure Dimension Variants for S-Function Blocks”.

**Introduced in R2016a**

# mdlSetOutputPortSymbolicDimensions

Specify symbolic dimensions of an output port and how those dimension propagate backward

## Languages

C, C++

## Syntax

```
#if defined(MATLAB_MEX_FILE)
#define MDL_SET_OUTPUT_PORT_SYMBOLIC_DIMENSIONS

static void mdlSetOutputPortSymbolicDimensions(SimStruct *S, int_T portIndex,
 SymbDimsId symbDimsId)

{
}
#endif
```

## Arguments

`S`

SimStruct representing an S-Function block.

`portIndex`

Index of an output port.

`symbDimsId`

Unique integer value corresponding to a symbolic dimension specification.

## Returns

This function does not return a value.

## Example

Call this function from inside the `mdlInitializeSizes` function. For an example of how to use this function to configure an S-function that supports forward and backward propagation of symbolic dimensions, see “Configure Dimension Variants for S-Function Blocks”.

**Introduced in R2016a**

# ssRegisterSymbolicDimsExpr

Create `SymbDimsId` from expression string (`aExpr`)

## Languages

C, C++

## Syntax

```
SymbDimsId ssRegisterSymbolicDimsExpr(SimStruct *S, const char_T* aExpr)
```

## Arguments

`S`

SimStruct representing an S-Function block.

`aExpr`

Expression string that forms a valid syntax in C.

## Returns

A unique `SymbDimsId` (integer value) that represents a symbolic dimension specification.

## Example

This example creates a `SymbDimsId` for the expression string `[ F / C , D * (B-3)]`.

```
const SymbDimsId outputDimsId =
       ssRegisterSymbolicDimsExpr(S, "[ F / C , D * (B-3)]");
```

**Introduced in R2016a**

# ssRegisterSymbolicDims

Create `SymbDimsId` from array of `SymDimsIds`

## Languages

C, C++

## Syntax

```
SymbDimsId ssRegisterSymbolicDims(SimStruct *S, const SymbDimsId* aDimsVec,
 const size_t aNumDims)
```

## Arguments

`S`

SimStruct representing an S-Function block.

`aDimsVec`

Array of `SymDimsIds`

`aNumDims`

Size of `SymDimsId` array

## Returns

A unique `SymbDimsId` (integer value) that represents a symbolic dimension specification.

## Example

For an example of how to use this function to configure an S-function that supports forward and backward propagation of symbolic dimensions, see “Configure Dimension Variants for S-Function Blocks”.

**Introduced in R2016a**

# ssRegisterSymbolicDimsString

Create `SymbDimsId` from identifier string

## Languages

C, C++

## Syntax

```
SymbDimsId ssRegisterSymbolicDimsString(SimStruct *S, const char_T* aString)
```

## Arguments

`S`

SimStruct representing an S-Function block.

`aString`

Identifier string

## Returns

A unique `SymbDimsId` (integer value) that represents a symbolic dimension specification.

## Example

This example creates a `SymbDimsId` for the string `"B"`.

```
const SymbDimsId symbolId = ssRegisterSymbolicDimsString(S, "B");
```

**Introduced in R2016a**

# ssRegisterSymbolicDimsIntValue

Create `SymbDimsId` from integer value

## Languages

C, C++

## Syntax

```
SymbDimsId ssRegisterSymbolicDimsIntValue(SimStruct *S, const int_T aIntValue)
```

## Arguments

`S`

SimStruct representing an S-Function block.

`aIntValue`

Dimensions value

## Returns

A unique `SymbDimsId` (integer value) that represents a symbolic dimension specification.

## Example

This example creates a `SymbDimsId` for the integer `2`.

```
const SymbDimsId symbolId = ssRegisterSymbolicDimsIntValue(S, 2);
```

**Introduced in R2016a**

# ssRegisterSymbolicDimsPlus

Create SymbDimsId by adding two symbolic dimensions

## Languages

C, C++

## Syntax

```
SymbDimsId ssRegisterSymbolicDimsPlus(SimStruct *S, const SymbDimsId aLHS,
 const SymbDimsId aRHS)
```

## Arguments

S

SimStruct representing an S-Function block.

aLHS

Unique integer value corresponding to a symbolic dimension specification.

aRHS

Unique integer value corresponding to a symbolic dimension specification.

## Returns

A unique SymbDimsId (integer value) that represents a symbolic dimension specification.

## Example

This example shows how to add the SymbDimsId symbDims to symbolId, and then sets the result equal to a new SymbDimsId called outputDimsId.

```
const SymbDimsId outputDimsId =
        ssRegisterSymbolicDimsPlus(S, symbDimsId, symbolId);
```

**Introduced in R2016a**

# ssRegisterSymbolicDimsMinus

Create `SymbDimsId` by subtracting two symbolic dimensions

## Languages

C, C++

## Syntax

```
SymbDimsId ssRegisterSymbolicDimsMinus(SimStruct *S, const SymbDimsId aLHS,
 const SymbDimsId aRHS)
```

## Arguments

`S`

SimStruct representing an S-Function block.

`aLHS`

Unique integer value corresponding to a symbolic dimension specification.

aRHS

Unique integer value corresponding to a symbolic dimension specification.

## Returns

A unique `SymbDimsId` (integer value) that represents a symbolic dimension specification.

## Example

For an example of how to use this function to configure an S-function that supports forward propagation of symbolic dimensions, see “Configure Dimension Variants for S-Function Blocks”.

**Introduced in R2016a**

# ssRegisterSymbolicDimsMultiply

Create `SymbDimsId` by multiplying two symbolic dimensions

## Languages

C, C++

## Syntax

```
SymbDimsId ssRegisterSymbolicDimsMultiply(SimStruct *S, const SymbDimsId aLHS,
 const SymbDimsId aRHS)
```

## Arguments

`S`

SimStruct representing an S-Function block.

`aLHS`

Unique integer value corresponding to a symbolic dimension specification.

aRHS

Unique integer value corresponding to a symbolic dimension specification.

## Returns

A unique `SymbDimsId` (integer value) that represents a symbolic dimension specification.

## Example

This example shows how to multiply the `SymbDimsId`s `symbDimsId` and `symbolId`. It sets the result equal to a new `SymbDimsId` called `outputDimsId`.

```
const SymbDimsId outputDimsId =
       ssRegisterSymbolicDimsMultiply(S, symbDimsId, symbolId);
```

**Introduced in R2016a**

# ssRegisterSymbolicDimsDivide

Create `SymbDimsId` by dividing two symbolic dimensions

## Languages

C, C++

## Syntax

```
SymbDimsId ssRegisterSymbolicDimsDivide(SimStruct *S, const SymbDimsId aLHS,
  const SymbDimsId aRHS)
```

## Arguments

`S`

SimStruct representing an S-Function block.

`aLHS`

Unique integer value corresponding to a symbolic dimension specification.

`aRHS`

Unique integer value corresponding to a symbolic dimension specification.

## Returns

A unique `SymbDimsId` (integer value) that represents a symbolic dimension specification.

## Example

This example shows how to divide the `SymbDimsId` `symbDimsId` by `symbolId`. It sets the result equal to a new `SymbDimsId` called `outputDimsId`.

```
const SymbDimsId outputDimsId =
        ssRegisterSymbolicDimsDivide(S, symbDimsId, symbolId);
```

**Introduced in R2016a**

# ssGetNumSymbolicDims

Get the number of dimensions for `SymbDimsId`

## Languages

C, C++

## Syntax

```
size_t  ssGetNumSymbolicDims(SimStruct *S, const SymbDimsId aSymbDimsId)
```

## Arguments

`S`

SimStruct representing an S-Function block.

`aSymbDimsId`

Unique integer value corresponding to a symbolic dimension specification.

## Returns

The number of dimensions for a `SymbDimsId`.

## Example

For an example of how to use this function to configure an S-function that supports forward and backward propagation of symbolic dimensions, see “Configure Dimension Variants for S-Function Blocks”.

**Introduced in R2016a**

# ssGetSymbolicDim

Get `SymbDimsId` from array of `SymbDimsId`s

## Languages

C, C++

## Syntax

```
SymbDimsId ssGetSymbolicDim(SimStruct *S, const SymbDimsId aSymbDimsId,
 const int_T aDimsIdx)
```

## Arguments

`S`

SimStruct representing an S-Function block.

`aSymbDimsId`

Unique integer corresponding to a symbolic dimension specification.

`aDimsIdx`

Array index

## Returns

A unique `SymbDimsId`.

## Example

For an example of how to use this function to configure an S-function that supports forward and backward propagation of symbolic dimensions, see "Configure Dimension Variants for S-Function Blocks".

**Introduced in R2016a**

# ssSetInputPortSymbolicDimsId

Set precompiled `SymbDimsId` of input port

## Languages

C, C++

## Syntax

```
void ssSetInputPortSymbolicDimsId(SimStruct *S, const int_T aPortIdx,
 const SymbDimsId aSymbDimsId)
```

## Arguments

`S`

SimStruct representing an S-Function block.

`aPortIdx`

Array index

`aSymbDimsId`

Unique integer value corresponding to a symbolic dimension specification.

## Returns

This function does not return a value.

## Example

You can call this function from inside the `mdlInitializeSizes` function. For an input port with an index of `0`, this example shows how to set the precompiled `SymbDimsId` equal to `inputDimsId`.

```
const SymbDimsId inputDimsId = ssRegisterSymbolicDimsExpr(S, "[A+3, B-2]");
    ssSetInputPortSymbolicDimsId(S, 0, inputDimsId);
```

## Introduced in R2016a

# ssGetCompInputPortSymbolicDimsId

Get compiled `SymbDimsId` of input port

## Languages

C, C++

## Syntax

```
SymbDimsId  ssGetCompInputPortSymbolicDimsId(SimStruct *S, const int_T aPortIdx)
```

## Arguments

`S`

SimStruct representing an S-Function block.

`aPortIdx`

Index of an input port

## Returns

`SymbDimsId` corresponding to symbolic dimensions of an input port.

## Example

For an example of how to use this function to configure an S-function that supports forward and backward propagation of symbolic dimensions, see “Configure Dimension Variants for S-Function Blocks”.

**Introduced in R2016a**

# ssSetCompInputPortSymbolicDimsId

Set compiled `SymbDimsId` of an input port

## Languages

C, C++

## Syntax

```
void ssSetCompInputPortSymbolicDimsId(SimStruct *S, const int_T aPortIdx,
  const SymbDimsId aSymbDimsId)
```

## Arguments

`S`

SimStruct representing an S-Function block.

`aPortIdx`

Index of an input port.

`aSymbDimsId`

Unique integer value corresponding to a symbolic dimension specification.

## Returns

This function does not return a value.

## Example

For examples of how to use this function to configure S-functions that support forward propagation of symbolic dimensions and forward and backward propagation of symbolic dimensions, see “Configure Dimension Variants for S-Function Blocks”.

**Introduced in R2016a**

# ssSetOutputPortSymbolicDimsId

Set precompiled `SymbDimsId` of an output port.

## Languages

C, C++

## Syntax

```
void ssSetOutputPortSymbolicDimsId(SimStruct *S, const int_T aPortIdx,
 const SymbDimsId aSymbDimsId)
```

## Arguments

`S`

SimStruct representing an S-Function block.

`aPortIdx`

Index of an input port.

`aSymbDimsId`

Unique integer value corresponding to a symbolic dimension specification.

## Returns

This function does not return a value.

## Example

You can call this function from inside the `mdlInitializeSizes` function. For an output port with an index of `0`, this example shows how to set the precompiled `SymbDimsId` equal to `outputDimsId`.

```
const SymbDimsId outputDimsId =
        ssRegisterSymbolicDimsExpr(S, "[ F / C , D * (B-3)]");
    ssSetOutputPortSymbolicDimsId(S, 0, outputDimsId);
```

**Introduced in R2016a**

# ssGetCompOutputPortSymbolicDimsId

Get compiled `SymbDimsId` of output port

## Languages

C, C++

## Syntax

```
SymbDimsId ssGetCompOutputPortSymbolicDimsId(SimStruct *S, const int_T aPortIdx)
```

## Arguments

`S`

SimStruct representing an S-Function block.

`aPortIdx`

Index of an output port.

## Returns

`SymbDimsId` corresponding to symbolic dimensions of an output port.

## Example

For an example of how to use this function to configure an S-function that supports forward and backward propagation of symbolic dimensions, see “Configure Dimension Variants for S-Function Blocks”.

**Introduced in R2016a**

# ssSetCompOutputPortSymbolicDimsId

Set compiled `SymbDimsId` of output port

## Languages

C, C++

## Syntax

```
void ssSetCompOutputPortSymbolicDimsId(SimStruct *S, const int_T aPortIdx,
 const SymbDimsId aSymbDimsId)
```

## Arguments

`S`

SimStruct representing an S-function block.

`aPortIdx`

Index of an output port.

`aSymbDimsId`

Unique integer value corresponding to a symbolic dimension specification.

## Returns

This function does not return a value.

## Example

For an example of how to use this function to configure an S-function that supports forward and backward propagation of symbolic dimensions, see “Configure Dimension Variants for S-Function Blocks”.

**Introduced in R2016a**

# ssSetCompDWorkSymbolicDimsId

Set compiled `SymbDimsId` of an index of a block's data type work (`DWork`) vector

## Languages

C, C++

## Syntax

```
void ssSetCompOutputPortSymbolicDimsId(SimStruct *S, const int_T aPortIdx,
 const SymbDimsId aSymbDimsId)
```

## Arguments

`S`

SimStruct representing an S-Function block.

`aPortIdx`

Index of an input port

`aSymbDimsId`

Unique integer value corresponding to a symbolic dimension specification.

## Returns

This function does not return a value.

**Introduced in R2016a**