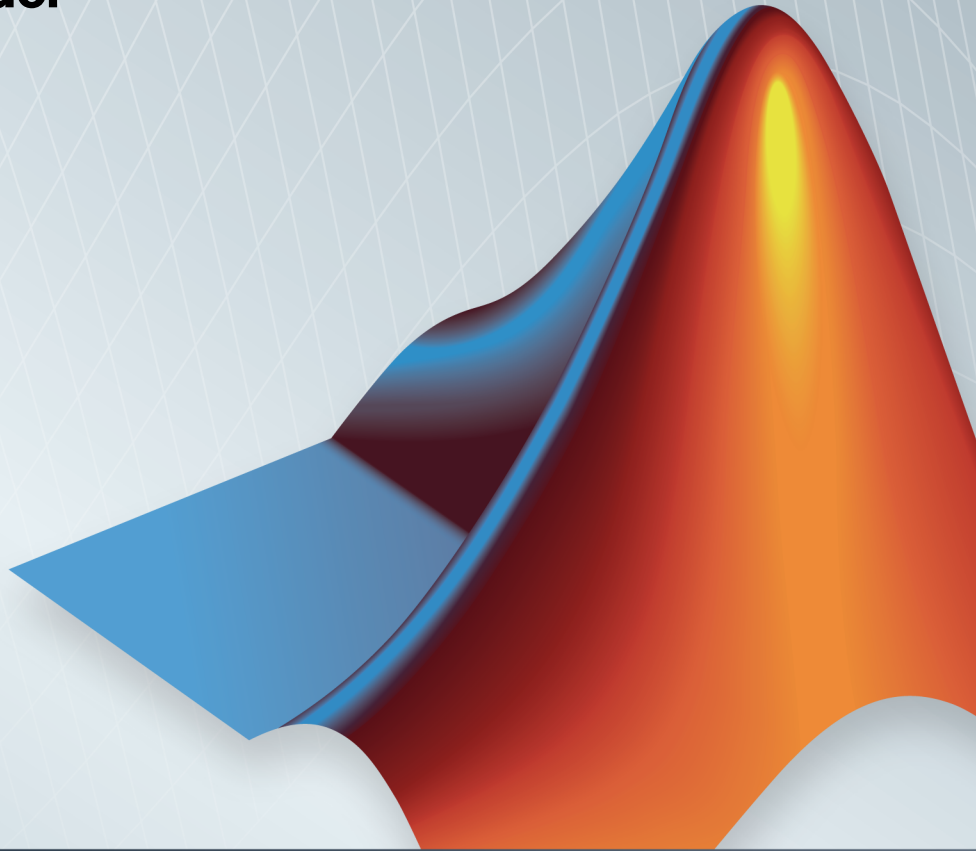


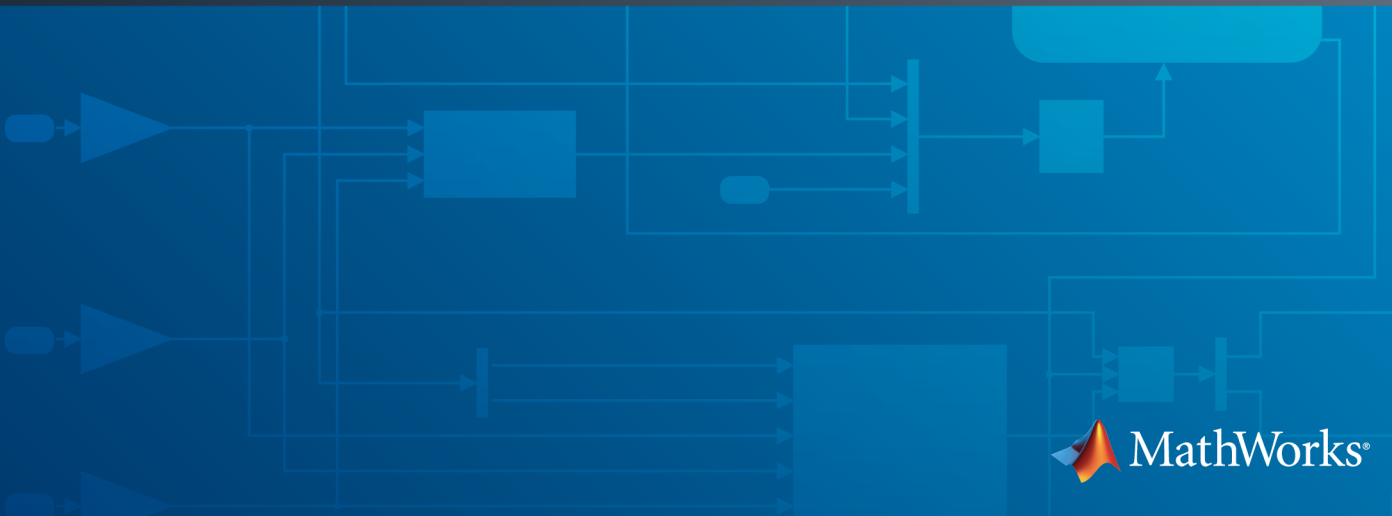
Embedded Coder[®]

User's Guide

R2014b



MATLAB[®] & SIMULINK[®]



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[®] User's Guide

© COPYRIGHT 2011–2014 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)

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.

Model Architecture and Design

1	Modeling Environment	
	Set Up Your Modeling Environment	1-2
2	Application Objectives	
3	Guidelines and Standards	
	What Are the Standards and Guidelines?	3-2
	MAAB Guidelines	3-4
	MISRA C Guidelines	3-5
	IEC 61508 Standard	3-7
	Apply Simulink and Embedded Coder to the IEC 61508 Standard	3-7
	Check for IEC 61508 Standard Compliance Using the Model Advisor	3-7
	Validate Traceability	3-8

ISO 26262 Standard	3-9
Apply Simulink and Embedded Coder to the ISO 26262 Standard	3-9
Check for ISO 26262 Standard Compliance Using the Model Advisor	3-9
Validate Traceability	3-8
EN 50128 Standard	3-11
Apply Simulink and Embedded Coder to the EN 50128 Standard	3-11
Check for EN 50128 Standard Compliance Using the Model Advisor	3-11
Validate Traceability	3-8
DO-178C Standard	3-13
Apply Simulink and Embedded Coder to the DO-178C Standard	3-13
Check for Standard Compliance Using the Model Advisor	3-13
Validate Traceability	3-8

Patterns for C Code

4

About Modeling Patterns	4-3
Prepare a Model for Code Generation	4-4
Configure a Signal	4-4
Configure Input and Output Ports	4-4
Initialize States	4-5
Set Up Configuration Parameters for Code Generation .	4-5
Set Up an Example Model With a Stateflow Chart	4-6
Set Up an Example Model With a MATLAB Function Block	4-7
Data Declaration	4-8
C Construct	4-8
Declare a Variable for a Block Parameter Using a Data Object	4-8
C Construct	4-9

Declare a Variable for a Signal using a Data Object . . .	4-10
Data Type Conversion	4-11
C Construct	4-11
Modeling Patterns	4-11
Modeling Pattern for Data Type Conversion — Simulink Block	4-11
Modeling Pattern for Data Type Conversion — Stateflow Chart	4-12
Modeling Pattern for Data Type Conversion — MATLAB Function Block	4-12
Other Type Conversions in Modeling	4-13
Type Qualifiers	4-14
Modeling Patterns for Type Qualifiers	4-14
Using a Tunable Parameter in the Base Workspace . . .	4-14
Use a Data Object of the Const Custom Storage Class	4-15
Relational and Logical Operators	4-16
Modeling Patterns for Relational and Logical Operators	4-16
Modeling Pattern for Relational or Logical Operators — Simulink Blocks	4-16
Modeling Pattern for Relational and Logical Operators — Stateflow Chart	4-17
Modeling Pattern for Relational and Logical Operators — MATLAB Function Block	4-18
Bitwise Operations	4-20
Simulink Bitwise-Operator Block	4-20
Stateflow Chart	4-21
MATLAB Function Block	4-22
If-Else	4-23
C Construct	4-23
Modeling Patterns	4-23
Modeling Pattern for If-Else: Switch block	4-23
Modeling Pattern for If-Else: Stateflow Chart	4-25
Modeling Pattern for If-Else: MATLAB Function Block	4-26
Switch	4-28
C Construct	4-28
Modeling Patterns	4-28
Modeling Pattern for Switch: Switch Case block	4-28

Modeling Pattern for Switch: MATLAB Function block	4-31
Convert If-Elseif-Else to Switch statement	4-32
For Loop	4-33
C Construct	4-33
Modeling Patterns:	4-33
Modeling Pattern for For Loop: For-Iterator Subsystem block	4-33
Modeling Pattern for For Loop: Stateflow Chart	4-36
Modeling Pattern for For Loop: MATLAB Function block	4-37
While Loop	4-39
C Construct	4-39
Modeling Patterns	4-39
Modeling Pattern for While Loop: While Iterator Subsystem block	4-39
Modeling Pattern for While Loop: Stateflow Chart	4-42
Modeling Pattern for While Loop: MATLAB Function Block	4-45
Do While Loop	4-48
C Construct	4-48
Modeling Patterns	4-48
Modeling Pattern for Do While Loop: While Iterator Subsystem block	4-48
Modeling Pattern for Do While Loop: Stateflow Chart	4-51
Function Call	4-54
C Construct	4-54
Procedure	4-54
Results	4-55
Function Prototyping	4-56
C Construct	4-56
Modeling Patterns	4-56
Function Call Using Graphical Functions	4-56
Control Function Prototype of the model_step Function	4-58
External C Functions	4-59
C Construct	4-59
Modeling Patterns	4-59
Use the Legacy Code Tool to Create S-functions	4-59

Use a Stateflow Chart to Make Calls to C Functions . . .	4-61
Using a MATLAB Function Block to Make Calls to C Functions	4-62
Macro Definitions (#define)	4-65
C Construct	4-65
Modeling Patterns	4-65
Use a 'Define' Custom Storage Class	4-65
Use a Custom Header File	4-66
Conditional Inclusions (#if / #endif)	4-68
Typedef	4-69
C Construct	4-69
Procedure	4-69
Results	4-70
Structures for Parameters	4-71
C Construct	4-71
Procedure	4-71
Results	4-72
Structures for Signals	4-73
C Construct	4-73
Modeling Patterns	4-73
Structure for Signals Using a 'Struct' Custom Storage Class	4-73
Structure for Signals Using a Simulink Non-Virtual Bus Object	4-74
Nested Structures	4-76
C Construct	4-76
Procedure	4-76
Results	4-78
Bitfields	4-80
C Construct	4-80
Procedure	4-80
Results	4-81
Arrays for Parameters	4-82
C Construct	4-82
Procedure	4-82

Results	4-83
Arrays for Signals	4-84
C Construct	4-84
Procedure	4-84
Results	4-84
Pointers for Signals	4-86
C Construct	4-86
Procedure	4-86
Results	4-86
Pointers Using Simulink Data Objects	4-87
C Construct	4-87
Procedure	4-87
Results	4-87

Variant Systems

5

About Variant Systems	5-2
Why Generate Code for Variant Systems?	5-3
Generate Preprocessor Conditionals for Variant Systems	5-4
Define Variant Controls	5-4
Configure Model for Generating Preprocessor Conditional Directives	5-5
Build Your Model	5-6
Review Code Variants in Code Generation Report	5-7
Generate Code for Model Variants	5-8
Generate Code for Variant Subsystems	5-10
Open Example Model	5-10
Define Variant Controls	5-10
Configure Model for Generating Preprocessor Conditional Directives	5-12

View Generated Code	5-13
Restrictions on Variant Subsystem Code Generation .	5-15
Special Considerations for Generating Preprocessor Conditionals	5-17
Limitations on Generating Code for Variants	5-18
Generated Code Components Not Compiled Conditionally	5-19

Scheduling Considerations

6

Use Discrete and Continuous Time	6-2
Support for Discrete and Continuous Time Blocks	6-2
Support for Continuous Solvers	6-2
Support for Stop Time	6-2
Optimize Multirate Multitasking Operation on RTOS	
Targets	6-4
Overview	6-4
Use rtmStepTask	6-5
Scheduling Code for Multirate Multitasking on VxWorks	6-5
Suppress Redundant Scheduling Calls	6-6

Data, Function, and File Definition

Data Types

7

What are User-Defined Data Types?	7-2
---	-----

Location of User-Defined Type Definitions	7-3
Omit a HeaderFile Value	7-3
Specify a HeaderFile Value	7-3
Create and Apply User-Defined Data Types	7-5
Specify Persistence Level for Signals and Parameters .	7-8
Buses	7-11
About Buses and Code Generation	7-11
Set Bus Diagnostics	7-12
Optimize Virtual and Nonvirtual Buses	7-12
Use Single-Rate and Multi-Rate Buses	7-15
Set Bus Signal Initial Values	7-18
Use Buses with Atomic Subsystems	7-22
Rename Built-In Data Types	7-26
Register mpt User Object Types	7-28
Introduction	7-28
Register mpt User Object Types Using	
sl_customization.m	7-28
mpt User Object Type Customization Using	
sl_customization.m	7-30
Data Type Replacement	7-32
Replace Built-In Data Type Names	7-32
Replace Built-In Data Type Names with Replacement	
Data Type Names	7-35
Programmatically Replace Built-In Data Type Names .	7-40
Replace <code>boolean</code> with an Integer Data Type	7-40
Data Type Replacement Limitations	7-42
Data Definition and Declaration Management	7-43
Overview of Data Objects	7-43
Create Simulink and mpt Data Objects	7-44
Create Data Objects for a Model	7-57
Define Global Data Objects in Separate File	7-63
Define Global Data Objects in Separate Files	7-64
Save and Load Data Objects	7-65
Apply Naming Rules to Identifiers Globally	7-65

Module Packaging Tool (MPT) Data Objects

8

MPT Data Object Properties	8-2
---	------------

Custom Storage Classes

9

Introduction to Custom Storage Classes	9-2
Custom Storage Class Memory Sections	9-3
Register Custom Storage Classes	9-3
Custom Storage Class Examples	9-3
Resources for Defining Custom Storage Classes	9-5
Simulink Package Custom Storage Classes	9-6
Design Custom Storage Classes and Memory Sections .	9-8
Create Packages for Custom Storage Class Definitions .	9-8
Use Custom Storage Class Designer	9-8
Edit Custom Storage Class Properties	9-14
Use Custom Storage Class References	9-19
Create and Edit Memory Section Definitions	9-23
Use Memory Section References	9-27
Apply Custom Storage Classes	9-29
About Applying Custom Storage Classes	9-29
Apply Custom Storage Classes to Parameters	9-30
Apply Custom Storage Classes to Signals	9-31
Custom Storage Classes Using Signal Objects	9-32
Custom Storage Classes Using Embedded Signal Objects	9-33
Specify Custom Storage Classes Using GUI	9-40
Specify Custom Storages Classes Using API	9-42
Generate Code with Custom Storage Classes	9-46
Code Generation Prerequisites	9-46
Code Generation With Custom Storage Classes	9-46

Define Advanced Custom Storage Classes Types	9-50
Introduction	9-50
Create Your Own Parameter and Signal Classes	9-50
Create Custom Attributes Classes for Custom Storage Classes	9-50
Write TLC Code for Custom Storage Classes	9-51
Register Custom Storage Class Definitions	9-51
GetSet Custom Storage Class	9-53
About GetSet Custom Storage Class	9-53
GetSet Custom Storage Class Properties	9-53
Apply the GetSet Custom Storage	9-54
GetSet Custom Storage Class Restrictions	9-54
Increase Code Efficiency With GetSet CSC	9-54
Custom Storage Class Implementation	9-56
Custom Storage Class Limitations	9-57

User Package Registration

10

About Data Object Wizard and User Packages	10-2
Register User Packages Using sl_customization.m	10-3
User Package Customization Using sl_customization.m	10-5

Function and Class Interfaces

11

Function Prototype Control	11-2
About Function Prototype Control	11-2
Configure Function Prototypes Using Graphical Interfaces	11-3

Sample Procedure for Configuring Function Prototypes	11-11
Configure Function Prototypes Programmatically	11-16
Sample Script for Configuring Function Prototypes	11-20
Verify Generated Code for Customized Functions	11-20
Function Prototype Control Limitations	11-21
C++ Class Interface Control	11-23
About C++ Class Interface Control	11-23
Simple Use of C++ Class Control	11-24
Customize C++ Class Interfaces Using Graphical Interfaces	11-30
Customize C++ Class Interfaces Programmatically	11-42
Configure Step Method for Model Class	11-45
Specify Custom Storage Class for C++ Class Code Generation	11-46
Model Class Copy Constructor and Assignment Operator	11-47
C++ Class Interface Control Limitations	11-48
Atomic Subsystem Code	11-50
About Nonvirtual Subsystem Code Generation	11-50
Configure Subsystem for Generating Modular Function Code	11-51
Modular Function Code for Nonvirtual Subsystems	11-56
Nonvirtual Subsystem Modular Function Code Limitations	11-61

Memory Sections

12

About Memory Sections	12-2
What Are Memory Sections?	12-2
Requirements for Defining Memory Sections	12-3
Define Memory Sections	12-5
Edit Memory Section Properties	12-5
Specify the Memory Section Name	12-6

Specify a Qualifier for Custom Storage Class Data	
Definitions	12-7
Specify Comment and Pragma Text	12-7
Surround Individual Definitions with Pragmas	12-7
Include Identifier Names in Pragmas	12-8
Configure Memory Sections	12-9
Declare Constant Data as Volatile	12-10
Apply Memory Sections	12-13
Assign Memory Sections to Custom Storage Classes	12-13
Apply Memory Sections to Model-Level Functions and	
Internal Data	12-15
Apply Memory Sections to Atomic Subsystems	12-17
Generated Code with Memory Sections	12-21
Sample ERT-Based Model with Subsystem	12-21
Model-Level Data Structures	12-23
Model-Level Functions	12-23
Subsystem Function	12-24

Code Generation

	Configuration
13	
Application Objectives	13-2
High-Level Code Generation Objectives	13-3
Determine Model Configuration for Specified	
Objectives	13-5
Check and Configure Model for Code Generation	
Objectives	13-6

Check and Configure Model for Code Generation Objectives Using Configuration Parameters Dialog Box	13-10
Configure Code Generation Objectives Programmatically	13-14
Check Objectives in Referenced Models	13-15
Checking Model and Configuration with Model Advisor	13-16
Check Model During Code Generation	13-17
Create Custom Objectives	13-18
Specify Parameters in Custom Objectives	13-18
Specify Checks in Custom Objectives	13-19
Determine Checks and Parameters in Existing Objectives	13-19
How to Create Custom Objectives	13-21
Code Generation Targets	13-24
About Target Selection	13-24
Select an ERT Target	13-25
Customize an ERT Target	13-26
Configure Support for Numeric Data	13-26
Configure Support for Time Values	13-26
Support for Non-inlined S-Functions	13-27
Configure Model Function Generation and Argument Passing	13-27
Set Up Support for Code Reuse	13-29
Configure a Code Replacement Library	13-30
Configuration Variations	13-31
Wizard	13-32
Configure and Optimize Model with Configuration Wizard Blocks	13-32
Add a Configuration Wizard Block	13-33
Use Configuration Wizard Blocks	13-35
Create a Custom Configuration Wizard Block	13-35

Add Custom Comments to Generated Code	14-3
Add Custom Comments for Signal or Parameter Identifiers	14-5
Add Global Comments	14-7
Use a Simulink DocBlock to Add a Comment	14-7
Use a Simulink Annotation to Add a Comment	14-10
Use a Stateflow Note to Add a Comment	14-10
Use Sorted Notes to Add Comments	14-11
Specify Comment Style	14-13
Customize Generated Identifier Naming Rules	14-14
Identifier Format Control	14-16
Control Name Mangling in Generated Identifiers	14-19
Minimize Name Mangling	14-19
Avoid Identifier Name Collisions with Referenced Models	14-21
Maintain Traceability for Generated Identifiers	14-22
Exceptions to Identifier Formatting Conventions	14-23
Identifier Format Control Parameters Limitations	14-24
Specify Simulink Data Object Naming Rules	14-26
Control Code Style	14-27
Control Parentheses in Generated Code	14-27
Control Indentation Style in Generated Code	14-28
Control Cast Expressions in Generated Code	14-29
Customize Code Organization and Format	14-35
Custom File Processing Components	14-35
Custom File Processing Configuration	14-36

Specify Templates For Code Generation	14-37
Code Generation Template (CGT) Files	14-38
Default CGT file	14-38
CGT File Structure	14-38
Built-In Tokens and Sections	14-39
Subsections	14-40
Custom File Processing (CFP) Templates	14-42
Custom File Processing (CFP) Template Structure ..	14-42
Change the Organization of a Generated File	14-44
Generate Source and Header Files with a Custom File Processing (CFP) Template	14-46
Generate Code with a CFP Template	14-46
Analysis of the Example CFP Template and Generated Code	14-48
Generate a Custom Section	14-51
Custom Tokens	14-53
Comparison of a Template and Its Generated File ..	14-54
Template and Generated File	14-55
Code Template API Summary	14-58
Generate Custom File and Function Banners	14-61
Create a Custom File and Function Banner Template	14-62
Customize a Code Generation Template (CGT) File for File and Function Banner Generation	14-63
Template Symbols and Rules	14-69
Introduction	14-69
Template Symbol Groups	14-69
Template Symbols	14-72
Rules for Modifying or Creating a Template	14-75
Code Annotation for Justifying Polyspace Checks ..	14-77
Manage Placement of Data Definitions and Declarations	14-79
Overview of Data Placement	14-79
Priority and Usage	14-80

Ownership Settings	14-85
Memory Section Settings	14-86
Data Placement Rules	14-86
Settings for a Data Object	14-86
Data Placement Rules and Results	14-94
Specify Delimiter for #Includes	14-105

Source Code Generation

15

Generating Code Using Embedded Coder®	15-2
Generate Code Modules	15-9
Introduction	15-9
Generated Code Modules	15-9
User-Written Code Modules	15-12
Customize Generated Code Modules	15-12
Generate Reentrant Code from MATLAB Code	15-15
What Is Reentrant Code?	15-15
When to Generate Reentrant Code	15-15
Generate Reentrant Code	15-16
Generated Code API	15-16
Call Reentrant Code in a Single-Thread Environment	15-17
Call Reentrant Code in a Multithreaded Environment	15-18
Call Reentrant Code with No Persistent or Global Data (UnIX Only)	15-19
Call Reentrant Code — Multithreaded with Persistent Data (Windows Only)	15-23
Call Reentrant Code — Multithreaded with Persistent Data (UNIX Only)	15-27
Generate Reentrant Code from Top-Level Models ...	15-33

Reports for Code Generation	16-2
HTML Code Generation Report Extensions	16-3
HTML Code Generation Report Location	16-5
HTML Code Generation Report for Referenced Models	16-6
Search Code Generation Report	16-7
Generate a Code Generation Report	16-8
Generate Code Generation Report After Build Process	16-9
Open Code Generation Report	16-11
Limitation	16-11
Generate Code Generation Report Programmatically	16-13
View Code Generation Report in Model Explorer ...	16-14
Package and Share the Code Generation Report	16-16
Package the Code Generation Report	16-16
View the Code Generation Report	16-17
Traceability in Code Generation Report	16-18
View Code Metrics and Definitions in the Generated Code	16-19
Web View of Model in Code Generation Report	16-20
About Model Web View	16-20
Generate HTML Code Generation Report with Model Web View	16-20
Model Web View Limitations	16-23

Analyze the Generated Code Interface	16-24
Code Interface Report Overview	16-24
Generating a Code Interface Report	16-25
Navigating Code Interface Report Subsections	16-27
Interpreting the Entry Point Functions Subsection ..	16-28
Interpreting the Inports and Outports Subsections ..	16-31
Interpreting the Interface Parameters Subsection ..	16-32
Interpreting the Data Stores Subsection	16-34
Code Interface Report Limitations	16-35
Static Code Metrics	16-36
About Static Code Metrics	16-36
Static Code Metrics Analysis	16-36
Generate Static Code Metrics Report for Simulink Model	16-39
Generate a Static Code Metrics Report for MATLAB Code	16-45
In a MATLAB Coder Project	16-45
At the Command Line	16-49
Analyze Code Replacements in the Generated Code ..	16-51
Document Generated Code with Simulink Report Generator	16-53
Generate Code for the Model	16-54
Open the Report Generator	16-54
Set Report Name, Location, and Format	16-56
Include Models and Subsystems in a Report	16-57
Customize the Report	16-58
Generate the Report	16-59

Code Replacement for Simulink Models

17

What Is Code Replacement?	17-2
Code You Can Replace From Simulink Models	17-4
About Code You Can Replace	17-4

Math Functions – Simulink Support	17-4
Math Functions – Stateflow Support	17-10
Memory Functions	17-15
Nonfinite Functions	17-16
Mutex and Semaphore Functions	17-17
Lookup Table Functions	17-18
Operators	17-18
Code Replacement Libraries	17-23
Code Replacement Terminology	17-25
Code Replacement Limitations	17-28
Replace Code Generated from Simulink Models	17-29
Choose a Code Replacement Library	17-32
About Choosing a Code Replacement Library	17-32
Explore Available Code Replacement Libraries	17-32
Explore Code Replacement Library Contents	17-32

Deployment

18

Desktops

Shared Object Libraries	18-2
About Host-Based Shared Libraries	18-2
Generate Shared Library Version of Model Code	18-3
Create Application Code to Use Shared Library	18-3
Host-Based Shared Library Limitations	18-7

19

Standalone Programs (No Operating System)	19-2
About Standalone Program Execution	19-2
Generate a Standalone Program	19-2
Standalone Program Components	19-3
Main Program	19-3
rt_OneStep and Scheduling Considerations	19-4
Static Main Program Module	19-10
Rate Grouping Compliance and Compatibility Issues	19-17
Operating System Integration	19-21
Processor Support Packages	19-22

Export Code Generated from Model to External Application

20

Export Function-Call Subsystems	20-2
Exporting Function-Call Subsystems	20-2
Requirements for Exporting Function-Call Subsystems	20-3
Techniques for Exporting Function-Call Subsystems	20-5
Optimize Exported Function-Call Subsystems	20-7
Export Functions That Depend on Elapsed Time	20-7
Function-Call Subsystem Export	20-8
Function-Call Subsystems Export Limitations	20-11
Control Generation of Function Prototypes	20-12
C++ Class Interface Control	20-14

Code Replacement Customization for Simulink Models

21

What Is Code Replacement Customization?	21-3
Code You Can Replace From Simulink Models	21-4
About Code You Can Replace	21-4
Math Functions – Simulink Support	21-4
Math Functions – Stateflow Support	21-10
Memory Functions	21-15
Nonfinite Functions	21-16
Mutex and Semaphore Functions	21-17
Operators	21-18
Code Replacement Match and Replacement Process	21-22
Code Replacement Customization Limitations	21-24
Develop a Code Replacement Library	21-26
Quick Start Library Development	21-27
Identify Code Replacement Requirements	21-37
Mapping Information Requirements	21-37
Build Information Requirements	21-38
Registration Information Requirements	21-38
Prepare for Code Replacement Library Development	21-40
Define Code Replacement Mappings	21-42
Defining Code Replacement Mappings	21-42
Define Mappings Interactively with the Code Replacement Tool	21-43
Define Mappings Programmatically	21-46
Specify Build Information for Replacement Code	21-58
Specifying Build Information	21-58
Specify Build Information Interactively with the Code Replacement Tool	21-59
Specify Build Information Programmatically	21-61

Register Code Replacement Mappings	21-66
Code Replacement Library Registration	21-66
Create Registration File Interactively with the Code Replacement Tool	21-67
Create Registration File Programmatically	21-69
Register a Code Replacement Library	21-71
Registration Files That Define Multiple Code Replacement Libraries	21-71
Registration Files That Define Code Replacement Library Hierarchies	21-72
Review and Test Code Replacements	21-74
Code Replacement Table Validation	21-74
Validate a Table Definition File	21-74
Review Table Content	21-75
Verify Library Registration and Content	21-76
Review and Trace Code Replacements Generated Using a Code Replacement Library	21-77
Determine Why Code Replacement Functions Are Not Used	21-81
Deploy Code Replacement Library	21-87
Math Function Code Replacement	21-88
Memory Function Code Replacement	21-90
Nonfinite Function Code Replacement	21-93
Semaphore and Mutex Function Replacement	21-96
Algorithm-Based Code Replacement	21-103
Lookup Table Function Code Replacement	21-106
Lookup Table Algorithm Replacement	21-106
Lookup Table and Interpolation Algorithm Parameters	21-106
Lookup Table Function Signatures	21-107
Interactive Mapping with Code Replacement Tool ..	21-111
Programmatic Specification	21-116
Sample Code Replacement Definition for the lookup2D Function	21-121

Data Alignment for Code Replacement	21-125
Code Replacement Data Alignment	21-125
Specify Data Alignment Requirements for Function Arguments	21-125
Provide Data Alignment Specifications for Compilers	21-127
Basic Example of Code Replacement Data Alignment	21-132
Replace MATLAB Functions with Custom Code Using <code>coder.replace</code>	21-136
Replace MATLAB Functions Specified in MATLAB Function Blocks	21-138
Reserved Identifiers and Code Replacement	21-142
Customize Matching and Replacement Process for Functions	21-144
Customize Code Matching and Replacement for Functions	21-145
Scalar Operator Code Replacement	21-149
Addition and Subtraction Operator Code Replacement	21-152
Algorithm Options	21-152
Interactive Specification with Code Replacement Tool	21-153
Programmatic Specification	21-153
Algorithm Classification	21-153
Limitations	21-155
Small Matrix Operation to Processor Code Replacement	21-157
Matrix Multiplication Operation to MathWorks BLAS Code Replacement	21-162
Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement	21-170
Remap Operator Output to Function Input	21-177

Customize Matching and Replacement Process for Operators	21-180
Create the Entry	21-180
Test the Entry	21-185
Fixed-Point Operator Code Replacement	21-187
Fixed-Point Operator Entries	21-187
Fixed-Point Numbers and Arithmetic	21-191
Addition	21-191
Subtraction	21-192
Multiplication	21-192
Division	21-193
Data Type Conversion (Cast)	21-194
Shift	21-194
Binary-Point-Only Scaling Code Replacement	21-197
Slope Bias Scaling Code Replacement	21-201
Net Slope Scaling Code Replacement	21-205
Multiplication and Division with Saturation	21-205
Multiplication and Division with Rounding Mode and Additional Implementation Arguments	21-208
Equal Slope and Zero Net Bias Code Replacement .	21-212
Data Type Conversions (Casts) and Operator Code Replacement	21-216
Casts From int32 To int16	21-216
Casts Using Net Slope	21-217
Shift Left Operations and Code Replacement	21-221
Shift Lefts for int16 Data	21-221
Shift Lefts Using Net Slope	21-222

Code Replacement Customization for MATLAB Code

22

What Is Code Replacement Customization?	22-3
--	-------------

Code You Can Replace from MATLAB Code	22-4
About Code You Can Replace	22-4
Math Functions	22-4
Memory Functions	22-9
Operators	22-9
Code Replacement Match and Replacement Process	22-14
Code Replacement Customization Limitations	22-15
Develop a Code Replacement Library	22-16
Quick Start Library Development	22-17
Identify Code Replacement Requirements	22-27
Mapping Information Requirements	22-37
Build Information Requirements	22-38
Registration Information Requirements	22-38
Prepare for Code Replacement Library Development	22-30
Define Code Replacement Mappings	22-31
Defining Code Replacement Mappings	22-42
Define Mappings Interactively with the Code Replacement Tool	22-43
Define Mappings Programmatically	22-46
Specify Build Information for Replacement Code ...	22-47
Specifying Build Information	22-58
Specify Build Information Interactively with the Code Replacement Tool	22-59
Specify Build Information Programmatically	22-61
Register Code Replacement Mappings	22-55
Code Replacement Library Registration	22-55
Create Registration File Interactively with the Code Replacement Tool	22-56
Create Registration File Programmatically	22-58
Register a Code Replacement Library	22-60
Registration Files That Define Multiple Code Replacement Libraries	22-60
Registration Files That Define Code Replacement Library Hierarchies	22-61

Review and Test Code Replacements	22-63
Code Replacement Table Validation	22-63
Validate a Table Definition File	22-63
Review Table Content	22-64
Verify Library Registration and Content	22-65
Review and Trace Code Replacements Generated Using a Code Replacement Library	22-66
Determine Why Code Replacement Functions Are Not Used	22-68
Deploy Code Replacement Library	22-72
Math Function Code Replacement	22-73
Memory Function Code Replacement	22-75
Specify In-Place Code Replacement	22-77
In-Place Code Replacement	22-77
Argument Specification Requirements	22-77
Interactive Argument Replacement Specification with Code Replacement Tool	22-77
Programmatic Argument Replacement Specification .	22-81
Replace MATLAB Functions with Custom Code Using coder.replace	22-84
Reserved Identifiers and Code Replacement	22-86
Customize Matching and Replacement Process for Functions	22-88
Scalar Operator Code Replacement	22-90
Addition and Subtraction Operator Code Replacement	22-92
Algorithm Options	22-92
Interactive Specification with Code Replacement Tool	22-93
Programmatic Specification	22-93
Algorithm Classification	22-93
Limitations	22-95
Small Matrix Operation to Processor Code Replacement	22-97

Matrix Multiplication Operation to MathWorks BLAS Code Replacement	22-101
Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement	22-108
Remap Operator Output to Function Input	22-115
Customize Matching and Replacement Process for Operators	22-118
Fixed-Point Operator Code Replacement	22-124
Fixed-Point Operator Entries	22-124
Fixed-Point Numbers and Arithmetic	22-128
Addition	22-128
Subtraction	22-129
Multiplication	22-129
Division	22-130
Data Type Conversion (Cast)	22-131
Shift	22-131
Binary-Point-Only Scaling Code Replacement	22-134
Slope Bias Scaling Code Replacement	22-137
Net Slope Scaling Code Replacement	22-140
Multiplication and Division with Saturation	22-140
Multiplication and Division with Rounding Mode and Additional Implementation Arguments	22-143
Equal Slope and Zero Net Bias Code Replacement .	22-147
Data Type Conversions (Casts) and Operator Code Replacement	22-150
Shift Left Operations and Code Replacement	22-154

	Configuration
23	
Configure Code Optimizations	23-2
Specify Global Variable Localization	23-5
Set Hardware Implementation Parameters	23-6
Use External Mode with the ERT Target	23-7
Memory Management	23-7
Generation of Pure Integer Code with External Mode .	23-7
	Code Execution Profiling
24	
Execution Profiling for Generated Code	24-2
Code Execution Profiling for SIL and PIL	24-5
Configure Code Execution Profiling for SIL and PIL .	24-6
Execution Profiling for Atomic Subsystems and Model Reference Hierarchies	24-8
View and Compare Code Execution Times	24-10
Analyze Code Execution Data	24-16
Tips and Limitations	24-18
Triggered Model Block	24-18
Outliers in Execution Time Profiles	24-18
Hardware-Specific Timer	24-18
Task Context Switching Due to Preemption	24-18

25 | **Code Execution Profiling for MATLAB Coder**

Execution Time Profiling for SIL and PIL	25-2
Generate Execution Time Profile	25-3
View Execution Time Profile	25-4
Analyze Execution Time Data	25-7
Extract Execution Time Data for Kalman Estimator Code	25-7

26 | **Data Copy Reduction**

Optimize Global Variable Usage	26-2
Minimize Global Data Access	26-3
Use Global to Hold Temporary Results	26-8
Reuse Block Outputs in the Generated Code	26-12
Reuse Global Block Outputs	26-12
Virtualized Output Ports Optimization	26-16
Control Signal Storage	26-18
Signal Reuse for Root-Level Model Inputs and Outputs	26-19
Reuse Root-Level I/O Signals	26-19

27

Remove Initialization Code	27-2
Generate Pure Integer Code If Possible	27-3
Disable MAT-File Logging	27-4
Simplify Multiply Operations In Array Indexing	27-5
Generated Code Results	27-5

28

Optimize Generated Code Using Specified Minimum and Maximum Values	28-2
How to Configure Your Model	28-2
How to Enable Optimization	28-3
Optimize Generated Code Using Specified Minimum and Maximum Values	28-3
Limitations	28-6
Reduce Global Variables in Nonreusable Subsystem Functions	28-8
Generate void-void Function	28-8
Generate Function with Arguments	28-9

Verification

29

About Code Tracing	29-2
---------------------------------	------

Format of Traceability Tags	29-3
Examples of Tagged Code	29-3
Trace Code to Model Objects Using Hyperlinks	29-4
Trace Model Objects to Generated Code	29-6
Reload Existing Traceability Information	29-9
Customize Traceability Reports	29-10
Generate a Traceability Matrix	29-12
Traceability Limitations	29-13

Component Verification

30

Component Verification in the Target Environment ..	30-2
Goals of Component Verification	30-3
Maximizing Code Portability and Configurability ...	30-4
Simplifying Code Integration and Maximizing Code Efficiency	30-5
Running Component Tests	30-6

Component Verification With a Real-Time Target Environment

31

About Real-Time Software Component Verification ..	31-2
Real-Time Software Component Testing	31-4

About SIL and PIL Simulations	32-2
What are SIL and PIL Simulations?	32-2
Why Use SIL and PIL	32-3
How SIL and PIL Simulations Work	32-4
Comparison of SIL and PIL Simulation	32-5
Choose a SIL or PIL Approach	32-7
Verify Top Model Code	32-8
Verify Referenced Model Code	32-9
Verify Subsystem Code	32-9
Configure a SIL or PIL Simulation	32-10
Top-Model SIL or PIL Simulation	32-10
Model Block SIL or PIL Simulation	32-12
Use a SIL or PIL Block	32-13
Check the SIL or PIL Configuration	32-14
Top Model Simulation Using SIL or PIL	32-17
Referenced Model Simulation Using SIL or PIL	32-18
Verify Internal Signals of a Component	32-20
Simulation Mode Override Behavior in Model Reference Hierarchy	32-21
Code Interfaces for SIL and PIL	32-23
Code Interface for Top-Model SIL or PIL	32-23
Code Interface for Model Block SIL or PIL	32-24
Configure Hardware Implementation Settings for SIL	32-25
Choose Hardware Implementation Approach	32-25
Portable Word Sizes	32-25
Test Hardware	32-28
Production hardware	32-29
Debug Code During SIL Simulations	32-30

PIL Customization for Target Environment	32-32
Target Connectivity Configurations for PIL	32-32
Target Connectivity API Components	32-32
Communications <code>rtiostream</code> API	32-33
Create PIL Target Connectivity Configuration	32-35
Create a Connectivity API Implementation	32-35
Test an <code>rtiostream</code> Driver	32-36
Synchronize Host and Target	32-38
Specify Hardware Timer	32-39
Register a Connectivity API Implementation	32-41
Target Connectivity API Examples	32-41
View Test Harness in Code Generation Report	32-43
SIL and PIL Simulation Support and Limitations ...	32-45
About SIL and PIL Simulation Support and Limitations	32-45
Code Source Support	32-45
Block Support	32-48
Configuration Parameters Support	32-50
I/O Support	32-56
Hardware Implementation Support	32-66
Other Feature Support	32-67
Programmatic Code Generation Verification	32-69
Code Generation Verification API Overview	32-69
Verify Numerical Equivalence with CGV	32-69
Verify Numerical Equivalence Between Two Modes of Execution of a Model	32-70
Plot Output Signals	32-75

Software-in-the-Loop Execution for MATLAB Coder

Code Verification Through Software-in-the-Loop and Processor-in-the-Loop Execution	33-2
---	-------------

Software-in-the-Loop Execution Through Project Interface	33-4
Software-in-the-Loop Execution From Command Line	33-6
SIL Execution of Code Generated for a Kalman Estimator	33-6
Code Debugging During SIL Execution	33-9
PIL Customization for Target Environment	33-12
Target Connectivity Configurations for PIL	33-12
Target Connectivity PIL API Components	33-12
Communications <code>rtiostream</code> API	33-13
Create PIL Target Connectivity Configuration	33-15
Create a Connectivity API Implementation	33-15
Test an <code>rtiostream</code> Driver	33-16
Synchronize Host and Target	33-38
Specify Hardware Timer	33-19
Register a Connectivity API Implementation	33-41
Processor-in-the-Loop Execution Through Project Interface	33-22
Processor-in-the-Loop Execution From Command Line	33-24
PIL Execution of Code Generated for a Kalman Estimator	33-24
SIL/PIL Execution Support and Limitations	33-29

Code Coverage

34

Code Coverage in SIL and PIL Simulations	34-2
Configure SIL and PIL Code Coverage	34-3

View Code Coverage Information at the End of SIL or PIL Simulations	34-5
Configure Code Coverage Programmatically	34-8
Code Coverage Summary and Annotations	34-10
LDRA Testbed Coverage	34-10
BullseyeCoverage Information	34-12
Code Coverage Tool Support	34-15
Code Coverage for PIL	34-16
PIL Support for LDRA Testbed	34-16
PIL Support for BullseyeCoverage	34-17
Tips and Limitations	34-18
Compiler and Platform Support for SIL	34-18
Right-Click Subsystem Build Unsupported for Code Coverage	34-18
BullseyeCoverage License Wait	34-18
Current Working Folder Cannot be UNC Path	34-18
Characters in <code>matlabroot</code> and File Path	34-18
Header Files with Identical Names	34-19
Code Coverage for Source Files in Shared Utility Folders	34-19
BullseyeCoverage Behavior with Inline Macros	34-19
SIL and PIL Simulations with Open LDRA Testbed ..	34-20
PIL Zero Coverage LDRA Testbed Annotations	34-20
Modify Legacy Code	34-20
IDE Link Does Not Support LDRA Testbed	34-20

Embedded IDEs and Embedded Targets

35	Getting Started with Embedded Targets	
	Embedded Coder Supported Hardware	35-2

Project and Build Configurations for Embedded Targets

36

Model Setup	36-2
Block Selection	36-2
Configure Target Hardware Resources	36-3
Configuration Parameters	36-5
Model Reference	36-12
IDE Projects	36-13
Support for Third Party Products	36-13
Code Generation and Build	36-13
Makefiles for Software Build Tool Chains	36-15
What is the XMakefile Feature	36-15
Using Makefiles to Generate and Build Software	36-18
Making an XMakefile Configuration Operational	36-21
Creating a New XMakefile Configuration	36-21
XMakefile User Configuration dialog	36-27

Verification and Profiling Generated Code

37

PIL Simulation for IDE and Toolchain Targets	37-2
Overview	37-2
PIL Approaches	37-3
Communications	37-7
Running Your PIL Application to Perform Simulation and Verification	37-13
Performing a Model Block PIL Simulation via SCI Using Makefiles	37-13
Definitions	37-16
PIL Issues and Limitations	37-17
Code Execution Profiling for IDE and Toolchain Targets	37-19
Execution Time Profiling	37-19
Stack Profiling	37-19

Perform Execution Time Profiling for IDE and Toolchain	
Targets	37-22
Execution Profiling During Standalone Execution . . .	37-22
Execution Profiling During PIL Simulation	37-25
Perform Stack Profiling with IDE and Toolchain	
Targets	37-27

Processor-Specific Optimizations for Embedded Targets

38

Replace Code for Embedded Targets	38-2
Using a Processor-Specific Code Replacement Library to Optimize Code	38-2
Process of Determining Optimization Effects Using Real-Time Profiling Capability	38-2

Working with Texas Instruments Code Composer Studio 3.3 IDE

39

Set Up	39-2
Code Composer Studio	39-3
Using Code Composer Studio with Embedded Coder Software	39-3
Default Project Configuration	39-3
Getting Started	39-5
Overview	39-5
Verifying Your Code Composer Studio Installation . . .	39-8
IDE Automation Interface	39-9
Getting Started with IDE Automation Interface	39-9
Getting Started with RTDX	39-25

Constructing tics Objects	39-42
tics Properties and Property Values	39-43
Overloaded Functions for tics Objects	39-43
tics Object Properties	39-44
Function List	39-50
IDE Project Generator	39-52
Introducing IDE Project Generator	39-52
IDE Project Generator and Board Selection	39-52
Generate an IDE Project	39-54
Model Reference	39-57
Exporting Filter Coefficients from FDATool	39-61
About FDATool	39-61
Preparing to Export Filter Coefficients to Code Composer Studio Projects	39-62
Exporting Filter Coefficients to Your Code Composer Studio Project	39-65
Preventing Memory Corruption When You Export Coefficients to Processor Memory	39-70
Using Makefiles with Code Composer Studio 3.x	39-76
Introduction	39-76
Set Up XMakefile for CCSv3	39-76
Prepare Your Model for CCSv3 and Makefiles	39-77
Create Target Configuration File in CCSv3	39-78
Load and Run the Embedded Software	39-78
Reported Limitations and Tips	39-80
Example Programs Do Not Run Well with Incorrect GEL Files	39-80
Changing Values of Local Variables Does Not Work	39-81
Code Composer Studio Cannot Find a File After You Halt a Program	39-81
C54x XPC Register Can Be Modified Only Through the PC Register	39-83
Working with More Than One Installed Version of Code Composer Studio	39-83
Changing CCS Versions During a MATLAB Session	39-84
MATLAB Hangs When Code Composer Studio Cannot Find a Board	39-84
Using Mapped Drives	39-85
Uninstalling Code Composer Studio 3.3 Prevents Embedded Coder From Connecting	39-85

PostCodeGenCommand Commands Do Not Apply to IDE Projects	39-86
---	-------

Setting Up Code Composer Studio (ert.tlc System Target File)	39-87
Prepare Your Model for CCSv3.3	39-87
Prepare Your Model for CCSv4/5	39-87

IDE Link Frequently Asked Question: Why do I get an error when I invoke TICCS?	39-89
Why do I get an error when I invoke TICCS?	39-89
How can I fix this problem?	39-89
What happens if I click Deselect All when CCS prompts that 'New components were detected'?	39-91
How do I use CCS Component Manager to enable IDE Link Components?	39-91

Working with Texas Instruments Code Composer Studio 4 & 5 IDE

40

Set Up	40-2
Code Composer Studio	40-3
Feature Support	40-3
Getting Started	40-4
Verifying Your Code Composer Studio Installation ...	40-4
Learning About Makefiles	40-4
Using Makefiles with Code Composer Studio 4/5	40-5
Introduction	40-5
Set Up XMakefile for CCSv4/5	40-5
Prepare Your Model for CCSv4/5 and Makefiles	40-6
Create Target Configuration File in CCSv4/5	40-7
Configure Windows Path for TI Debug Server Scripting (DSS)	40-7
Load and Run the Embedded Software Using DSS ...	40-8

Reported Limitations and Tips	40-11
Example Programs Do Not Run well with Incorrect GEL Files	40-11
PostCodeGenCommand Commands Do Not Apply to IDE Projects	40-11

Working with Texas Instruments C2000 Processors

41

Install Support for C2000 Processors	41-2
Workflows for Texas Instruments C2000	41-2
Install Code Composer Studio and control SUITE	41-2
Install Texas Instruments C2000	41-3
Configuring and Verifying	41-5
Data Type Support	41-6
Scheduling and Timing	41-7
Overview	41-7
Timer-Based Interrupt Processing	41-7
Asynchronous Scheduling	41-8
Sharing General Purpose Timers between C281x Peripherals	41-13
Sharing General Purpose Timers between CAP and eCAN	41-14
Sharing General Purpose Timers between CAP and SPI	41-17
Overview of Creating Models for C2000 Processors .	41-21
Accessing the Embedded Coder Block Library	41-21
Building Your Model	41-21
Using the c2000lib Blockset	41-23
Introduction	41-23
Hardware Setup	41-23
Starting the c2000lib Library	41-24
Setting Up the Model	41-24
Adding Blocks to the Model	41-25

Generating Code from the Model	41-26
Configuring Timing Parameters for CAN Blocks	41-28
The CAN Blocks	41-28
Setting Timing Parameters	41-28
Parameter Tuning and Signal Logging	41-33
ADC-PWM Synchronization via ADC Interrupt	41-44
Run the model using CCSv3	41-46
Run the model using CCSv4	41-46
Run the model using CCSv5	41-47
Configuring Acquisition Window Width for ADC Blocks	41-49
What Is an Acquisition Window?	41-49
Configuring ADC Parameters for Acquisition Window Width	41-51
Real-Time Data Exchange via RTDX™	41-55
SPI-Based Control of PWM Duty Cycle	41-57
HIL Verification of IIR Filter via SCI	41-60
CAN-Based Control of PWM Duty Cycle	41-66
CAN Calibration Protocol and External Mode	41-69
Using the I2C Bus to Access a Connected EEPROM .	41-77
Using the IQmath Library	41-81
About the IQmath Library	41-81
Fixed-Point Numbers	41-82
Building Models	41-86
Programming Flash Memory	41-89
Introduction	41-89
Installing TI Flash APIs	41-89
Configuring the DSP Board Bootloader	41-90
Configuring the Software for Automatic Flash Programming	41-90
Selectively Erase, Program, or Verify Specific Flash Sectors	41-91

Placing Additional Code or Data on Unused Flash Sectors	41-91
Configuring LIN Communications	41-94
Overview	41-94
Configuring Your Model	41-94
Open Examples for Embedded Coder Texas Instruments C2000	41-96
Tips and Limitations	41-98
RTDX Simulation Blocks	41-99

Code Generation from MATLAB Code

Build Configuration for Code Generation from MATLAB Code

42

Specify Comment Style	42-2
Specify Comment Style Using the Project Interface ...	42-2
Specify Comment Style at the Command-Line Interface	42-2
Generate Custom File and Function Banners for C and C++ Code	42-4
Code Generation Template (CGT) Files for MATLAB ..	42-7
Default CGT file	42-7
CGT File Structure	42-7
Components of the CGT File Sections	42-9
Customize Generated Identifiers	42-18
Customize Identifiers Using the Project Interface ...	42-18
Customize Generated Identifiers Using the Command Line Interface	42-19

What Is Code Replacement?	43-2
Code You Can Replace from MATLAB Code	43-4
About Code You Can Replace	43-4
Math Functions	43-4
Memory Functions	43-9
Operators	43-9
Code Replacement Libraries	43-15
Code Replacement Terminology	43-17
Code Replacement Limitations	43-20
Replace Code Generated from MATLAB Code	43-21
Choose a Code Replacement Library	43-24
About Choosing a Code Replacement Library	43-24
Explore Available Code Replacement Libraries	43-24
Explore Code Replacement Library Contents	43-32

Verification of Code Generated from MATLAB Code

Highlight Potential Data Type Issues in a Report	44-2
Enable Highlight Option Using a Project	44-3
Enable Highlight Option Using the Command Line Interface	44-4
Find Potential Data Type Issues in Generated Code ..	44-5
Data Type Issues Overview	44-5
Enable Highlighting of Potential Data Type Issues ...	44-5
Find and Address Cumbersome Operations	44-6
Find and Address Expensive Rounding	44-8
Find and Address Expensive Comparison Operations .	44-9

Model Architecture and Design

Modeling Environment

Set Up Your Modeling Environment

When developing a system, use a combination of products to model each system component based on the domain to which it applies.

The following table guides you to information and examples that pertain to use of the Embedded Coder[®] product to meet goals for specific domains.

Goals	Related Product Information	Examples
Generate a software design description	“Code Generation” in the Simulink Report Generator™ documentation	rtwdemo_codegenrpt
Trace model requirements to generated code	“Requirements Traceability” in the Simulink Verification and Validation™ documentation	rtwdemo_requirements
Implement application on fixed-point processors	“Data Types and Scaling” and “Fixed-Point Code Generation” in the Fixed-Point Designer™ documentation	rtwdemo_fixpt1 rtwdemo_fuelsys_fxp_publish rtwdemo_dspanc_fixpt
Use an integrated development environment (IDE) to integrate an application on a target processor automatically	“Program Building, Interaction, and Debugging” topics in the Embedded Coder documentation “Program Building, Interaction, and Debugging” and “Desktop Targets” topics in the Simulink Coder documentation	In rtwdemos, select one of the following folders: Desktop IDEs , Desktop Targets , Embedded IDEs , or Embedded Targets

Application Objectives

The first step in applying Embedded Coder configuration options to the application development process is to consider how your application objectives, particularly with respect to efficiency, traceability, and safety, map to code generation options in a model configuration set.

Parameters that you set in the **Solver**, **Data Import/Export**, **Diagnostics**, and **Code Generation** panes of the Configuration Parameters dialog box affect the behavior of a model in simulation and the code generated for the model.

Consider questions such as the following:

- What settings might help you debug your application?
- What is the highest objective for your application — efficiency, traceability, debugging, safety precaution, or some other criteria?
- What is the second highest objective?
- Can the objective at the start of the project differ from the objective required for the end result? What tradeoffs can you make?

After you answer these questions:

- 1 Define your objectives in the configuration set. For more information, see “High-Level Code Generation Objectives” on page 13-3.
- 2 Use the Code Generation Advisor to identify parameter values that are not configured for the objectives that you selected. For more information, see “Determine Model Configuration for Specified Objectives” on page 13-5.

Guidelines and Standards

- “What Are the Standards and Guidelines?” on page 3-2
- “MAAB Guidelines” on page 3-4
- “MISRA C Guidelines” on page 3-5
- “IEC 61508 Standard” on page 3-7
- “ISO 26262 Standard” on page 3-9
- “EN 50128 Standard” on page 3-11
- “DO-178C Standard” on page 3-13

What Are the Standards and Guidelines?

If your application has mission-critical development and certification goals, your models or subsystems and the code generated for them might need to comply with one or more of the standards and guidelines listed in the following table.

Standard or Guidelines	Organization	For More Information, See...
Guidelines: Use of MATLAB®, Simulink, and Stateflow® software for control algorithm modeling – MathWorks Automotive Advisory Board (MAAB) Guidelines	MAAB	<ul style="list-style-type: none"> • Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow Software: PDF, Word • Develop Models and Code That Comply with “MAAB Guidelines” on page 3-4
Guidelines: Use of the C Language in Critical Systems (MISRA C ^{®a})	Motor Industry Software Reliability Association (MISRA)	<ul style="list-style-type: none"> • MISRA C Web site • Technical Solution 1-1IFP0W on the MathWorks Web site • Develop Models and Code That Comply with “MISRA C Guidelines” on page 3-5
Standard: AUTomotive Open System ARchitecture (AUTOSAR)	AUTOSAR Development Partnership	<ul style="list-style-type: none"> • Publications and specifications available from the AUTOSAR Web site • Technical Solution 1-2WFS27 on the MathWorks Web site • “AUTOSAR Standard”
Standard: IEC 61508, Functional safety of electrical/ electronic/ programmable electronic safety-related systems	International Electrotechnical Commission	<ul style="list-style-type: none"> • IEC functional safety zone Web site • Model-Based Design for IEC 61508 (Excerpts) — For the complete document, see Technical Solution 1-32COJP on the MathWorks Web site.

Standard or Guidelines	Organization	For More Information, See...
		<ul style="list-style-type: none"> • Develop Models and Code That Comply with “IEC 61508 Standard” on page 3-7
Standard: ISO 26262, Road Vehicles - Functional Safety	International Organization for Standardization	<ul style="list-style-type: none"> • ISO 26262 Support in MATLAB and Simulink • Develop Models and Code That Comply with “ISO 26262 Standard” on page 3-9
Standard: EN 50128, Railway applications — Software for railway control and protection systems	European Committee for Electrotechnical Standardization	<ul style="list-style-type: none"> • Develop Models and Code That Comply with “EN 50128 Standard” on page 3-11
Standard: DO-178C, Software Considerations in Airborne Systems and Equipment Certification	Radio Technical Commission for Aeronautics (RTCA)	<ul style="list-style-type: none"> • Develop Models and Code That Comply with “DO-178C Standard” on page 3-13

a. MISRA[®] and MISRA C are registered trademarks of MISRA Ltd., held on behalf of the MISRA Consortium.

For information on whether Simulink Coder technology is certified or qualified and whether safety-critical software has been developed with MathWorks tools, see Embedded Coder — Code Certification with MathWorks Tools.

MAAB Guidelines

The MathWorks Automotive Advisory Board (MAAB) involves major automotive OEMs and suppliers in the process of evolving MathWorks controls, simulation, and code generation products, including Simulink, Stateflow, and Simulink Coder. An important result of the MAAB has been the “MAAB Control Algorithm Modeling” guidelines.

If you have a Simulink Verification and Validation product license, you can check that your Simulink model or subsystem, and the code that you generate from it, complies with MAAB guidelines. To check your model or subsystem, open the “Simulink Model Advisor”. Navigate to **By Product > Simulink Verification and Validation > Modeling Standards > MathWorks Automotive Advisory Board Checks** and run the “MathWorks Automotive Advisory Board checks”.

For more information on using the Model Advisor, see “Run Model Checks” in the Simulink documentation.

MISRA C Guidelines

The Motor Industry Software Reliability Association (MISRA²) has established “Guidelines for the Use of the C Language in Critical Systems” (MISRA C). For general information about MISRA C, see www.misra-c.com.

In 1998, MIRA Ltd. published MISRA C (MISRA C:1998) to provide a restricted subset of a standardized, structured language that met Safety Integrity Level (SIL) 2 and higher. A major update based on feedback was published in 2004 (MISRA C:2004), followed by a minor update in 2007 known as Technical Corrigendum (TC1). In 2007, MISRA also published the MISRA AC AGC standard, “MISRA AC AGC: Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation.” MISRA AC AGC does not change MISRA C:2004 rules, rather it modifies the adherence recommendation. For more information about MISRA C, see www.misra-c.com.

Embedded Coder and Simulink offer capabilities to minimize the potential for MISRA C rule violations, especially rules deemed applicable by MISRA AC AGC.

To configure a model or subsystem so that the code generator is most likely to produce MISRA C:2004 compliant code, use the Code Generation Advisor. For more information, refer to:

- “High-Level Code Generation Objectives”
- “Determine Model Configuration for Specified Objectives”

The “Model Advisor” also checks that you developed your model or subsystem to increase the likelihood of generating MISRA C:2004 compliant code. To check your model or subsystem:

- 1 Open the “Model Advisor”.
- 2 Navigate to **By Product > Embedded Coder**.
- 3 Run the following checks:
 - “Check for blocks not recommended for MISRA-C:2004 compliance”
 - “Check configuration parameters for MISRA-C:2004 compliance”

For more information about using the Model Advisor, see “Run Model Checks” in the Simulink documentation.

2. MISRA and MISRA C are registered trademarks of MIRA Ltd., held on behalf of the MISRA Consortium.

For information about using Embedded Coder software within MISRA C guidelines, see Technical Solution 1-1IFPOW on the MathWorks Web site.

IEC 61508 Standard

In this section...

“Apply Simulink and Embedded Coder to the IEC 61508 Standard” on page 3-7

“Check for IEC 61508 Standard Compliance Using the Model Advisor” on page 3-7

“Validate Traceability” on page 3-8

Apply Simulink and Embedded Coder to the IEC 61508 Standard

Applying Model-Based Design to a safety-critical system requires extra consideration and rigor so that the system adheres to defined safety standards. IEC 61508, Functional safety of electrical/electronic/programmable electronic safety related systems, is such a standard. Because the standard was published when most software was coded by hand, the standard needs to be mapped to Model-Based Design technologies. Model-Based Design for IEC 61508 (Excerpts) provides a sampling of information available from a document that offers recommendations on how to apply Simulink, Simulink Coder, and third-party products for Model-Based Design to IEC 61508 measures and techniques. For the complete version of Model-Based Design for IEC 61508, see Technical Solution 1-32COJP on the MathWorks Web site.

MathWorks provides an IEC Certification Kit product that you can use to certify MathWorks code generation and verification tools for projects based on the IEC 61508 standard. For more information, see <http://www.mathworks.com/products/iec-61508/>.

Check for IEC 61508 Standard Compliance Using the Model Advisor

If you have a Simulink Verification and Validation product license, you can check that your Simulink model or subsystem and the code that you generate from it complies with selected aspects of the IEC 61508 standard by running the “Simulink Model Advisor”. Navigate to **By Product > Simulink Verification and Validation > Modeling Standards > IEC 61508, ISO 26262, and EN 50128 Checks** or **By Task > Modeling Standards for IEC 61508** and run the “IEC 61508, ISO 26262, and EN 50128 Checks”.

For more information on using the Model Advisor, see “Run Model Checks” in the Simulink documentation.

Validate Traceability

Typically, applications that require certification require some level of traceability between requirements, models, and corresponding code.

To...	Use...
Associate requirements documents with objects in Simulink models	The “Requirements Traceability” that is available if you have a Simulink Verification and Validation license.
Trace model blocks and subsystems to generated code	The “ Model-to-code traceability option” when generating an HTML report during the code generation or build process.
Trace generated code to model blocks and subsystems	The “ Code-to-model traceability option” when generating an HTML report during the code generation or build process.

ISO 26262 Standard

In this section...

“Apply Simulink and Embedded Coder to the ISO 26262 Standard” on page 3-9

“Check for ISO 26262 Standard Compliance Using the Model Advisor” on page 3-9

“Validate Traceability” on page 3-8

Apply Simulink and Embedded Coder to the ISO 26262 Standard

Applying Model-Based Design to a safety-critical system requires extra consideration and rigor so that the system adheres to defined functional safety standards. ISO 26262, Road Vehicles - Functional Safety, is such a standard. For further information about MathWorks support for ISO 26262, see ISO 26262 Support in MATLAB and Simulink.

MathWorks provides an IEC Certification Kit product that you can use to qualify MathWorks code generation and verification tools for projects based on the ISO 26262 standard. For more information, see <http://www.mathworks.com/products/iso-26262/>.

Check for ISO 26262 Standard Compliance Using the Model Advisor

If you have a Simulink Verification and Validation product license, you can check that your Simulink model or subsystem and the code that you generate from it complies with selected aspects of the ISO 26262 standard by running the “Simulink Model Advisor”. Navigate to **By Product > Simulink Verification and Validation > Modeling Standards > IEC 61508, ISO 26262, and EN 50128 Checks** or **By Task > Modeling Standards for ISO 26262** and run the “IEC 61508, ISO 26262, and EN 50128 Checks”.

For more information on using the Model Advisor, see “Run Model Checks” in the Simulink documentation.

Validate Traceability

Typically, applications that require certification require some level of traceability between requirements, models, and corresponding code.

To...	Use...
Associate requirements documents with objects in Simulink models	The “Requirements Traceability” that is available if you have a Simulink Verification and Validation license.

To...	Use...
Trace model blocks and subsystems to generated code	The “ Model-to-code traceability option” when generating an HTML report during the code generation or build process.
Trace generated code to model blocks and subsystems	The “ Code-to-model traceability option” when generating an HTML report during the code generation or build process.

EN 50128 Standard

In this section...

“Apply Simulink and Embedded Coder to the EN 50128 Standard” on page 3-11
 “Check for EN 50128 Standard Compliance Using the Model Advisor” on page 3-11
 “Validate Traceability” on page 3-8

Apply Simulink and Embedded Coder to the EN 50128 Standard

Applying Model-Based Design to a safety-critical system requires extra consideration and rigor so that the system adheres to defined safety standards. EN 50128, Railway applications — Software for railway control and protection systems, is such a standard.

MathWorks provides an IEC Certification Kit product that you can use to certify MathWorks code generation and verification tools for projects based on the EN 50128 standard. For more information, see <http://www.mathworks.com/products/iec-61508/>.

Check for EN 50128 Standard Compliance Using the Model Advisor

If you have a Simulink Verification and Validation product license, you can check that your Simulink model or subsystem and the code that you generate from it complies with selected aspects of the EN 50128 standard by running the “Simulink Model Advisor”. Navigate to **By Product > Simulink Verification and Validation > Modeling Standards > IEC 61508, ISO 26262, and EN 50128 Checks** or **By Task > Modeling Standards for EN 50128** and run the “IEC 61508, ISO 26262, and EN 50128 Checks”.

For more information on using the Model Advisor, see “Run Model Checks” in the Simulink documentation.

Validate Traceability

Typically, applications that require certification require some level of traceability between requirements, models, and corresponding code.

To...	Use...
Associate requirements documents with objects in Simulink models	The “Requirements Traceability” that is available if you have a Simulink Verification and Validation license.

To...	Use...
Trace model blocks and subsystems to generated code	The “ Model-to-code traceability option” when generating an HTML report during the code generation or build process.
Trace generated code to model blocks and subsystems	The “ Code-to-model traceability option” when generating an HTML report during the code generation or build process.

DO-178C Standard

In this section...

“Apply Simulink and Embedded Coder to the DO-178C Standard” on page 3-13

“Check for Standard Compliance Using the Model Advisor” on page 3-13

“Validate Traceability” on page 3-8

Apply Simulink and Embedded Coder to the DO-178C Standard

Applying Model-Based Design to a high-integrity system requires extra consideration and rigor so that the system adheres to defined safety standards. DO-178C Software Considerations in Airborne Systems and Equipment Certification is such a standard. A supplement to DO-178C, DO-331, provides guidance on the use of Model-Based Design technologies. MathWorks provides a DO Qualification Kit product that you can use to qualify MathWorks verification tools for projects based on the DO-178C, DO-331, and related standards. For more information, see <http://www.mathworks.com/products/do-178/>.

For information about Model-Based Design and MathWorks support of aerospace and defense industry standards, see <http://www.mathworks.com/aerospace-defense/>.

Check for Standard Compliance Using the Model Advisor

If you have a Simulink Verification and Validation product license, you can check that your Simulink model or subsystem and the code that you generate from it complies with selected aspects of the DO-178C standard by running the “Simulink Model Advisor”. Navigate to **By Product > Simulink Verification and Validation > Modeling Standards > DO-178C/DO-331 Checks** or **By Task > Modeling Standards for DO-178C/DO-331** and run the “DO-178C/DO-331 checks”.

For more information on using the Model Advisor, see “Run Model Checks” in the Simulink documentation.

Validate Traceability

Typically, applications that require certification require some level of traceability between requirements, models, and corresponding code.

To...	Use...
Associate requirements documents with objects in Simulink models	The “Requirements Traceability” that is available if you have a Simulink Verification and Validation license.
Trace model blocks and subsystems to generated code	The “ Model-to-code traceability option” when generating an HTML report during the code generation or build process.
Trace generated code to model blocks and subsystems	The “ Code-to-model traceability option” when generating an HTML report during the code generation or build process.

Patterns for C Code

- “About Modeling Patterns” on page 4-3
- “Prepare a Model for Code Generation” on page 4-4
- “Data Declaration” on page 4-8
- “Data Type Conversion” on page 4-11
- “Type Qualifiers” on page 4-14
- “Relational and Logical Operators” on page 4-16
- “Bitwise Operations” on page 4-20
- “If-Else” on page 4-23
- “Switch” on page 4-28
- “For Loop” on page 4-33
- “While Loop” on page 4-39
- “Do While Loop” on page 4-48
- “Function Call” on page 4-54
- “Function Prototyping” on page 4-56
- “External C Functions” on page 4-59
- “Macro Definitions (#define)” on page 4-65
- “Conditional Inclusions (#if / #endif)” on page 4-68
- “Typedef” on page 4-69
- “Structures for Parameters” on page 4-71
- “Structures for Signals” on page 4-73
- “Nested Structures” on page 4-76
- “Bitfields” on page 4-80
- “Arrays for Parameters” on page 4-82
- “Arrays for Signals” on page 4-84
- “Pointers for Signals” on page 4-86

- “Pointers Using Simulink Data Objects” on page 4-87

About Modeling Patterns

Several standard methods are available for setting up a model to generate specific C constructs in your code. For preparing your model for code generation, some of these methods include: configuring signals and ports, initializing states, and setting up configuration parameters for code generation. Depending on the components of your model, some of these methods are optional. Methods for configuring a model to generate specific C constructs are organized by category, for example, the Control Flow category includes constructs `if-else`, `switch`, `for`, and `while`. Refer to the name of a construct to see how you should configure blocks and parameters in your model. Different modeling methodologies are available, such as Simulink blocks, Stateflow charts, and MATLAB Function blocks, to implement a C construct.

Model examples have the following naming conventions:

Model Components	Naming Convention
Inputs	u1, u2, u3, and so on
Outputs	y1, y2, y3, and so on
Parameters	p1, p2, p3, and so on
States	x1, x2, x3, and so on

Input ports are named to reflect the signal names that they propagate.

Prepare a Model for Code Generation

In this section...

“Configure a Signal” on page 4-4

“Configure Input and Output Ports” on page 4-4

“Initialize States” on page 4-5

“Set Up Configuration Parameters for Code Generation” on page 4-5

“Set Up an Example Model With a Stateflow Chart” on page 4-6

“Set Up an Example Model With a MATLAB Function Block” on page 4-7

Configure a Signal

- 1 Create a model in Simulink. For more information, see “Modeling Basics”.
- 2 Right-click a signal line. Select **Properties**. For more information about the Signal Properties dialog box, see “Signal Properties”.
- 3 Enter a signal name for the **Signal name** parameter.
- 4 On the same Signal Properties dialog box, select the **Code Generation** tab. Use the drop down menu for the **Storage class** parameter to specify a storage class. Examples in this chapter use `Exported Global`.

Note: Alternatively, on the Signal Properties dialog box, select **Signal name must resolve to Simulink signal object**. Then create a signal data object in the base workspace with the same name as the signal. See “Create Simulink and mpt Data Objects” for more information on creating data objects in the base workspace. (Examples use `mpt.Signal` and specify the **Storage class** as `ExportedGlobal`.)

Configure Input and Output Ports

- 1 In your model,

Double-click an Inport or Outport block. A Block Parameters dialog box opens.
- 2 Select the **Signal Attributes** tab.
- 3 Specify the **Port dimensions** and **Data type**. Examples leave the default value for **Port dimensions** as `-1` (for inherited) and **Data type** as `Inherit: auto`.

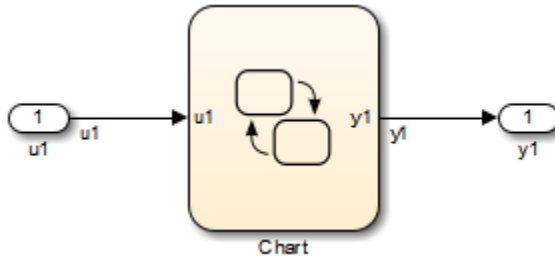
Initialize States

- 1 Double-click a block.
- 2 In the Block Parameters dialog box, select the **Main** tab.
- 3 Specify the **Initial conditions** and **Sample time**. For more information, see “Specify Sample Time”.
- 4 Select the **State Attributes** pane. Specify the state name. See “States”.
- 5 You can also use the Data Object Wizard for creating data objects. A part of this process initializes states. See “Create Data Objects with Data Object Wizard”.

Set Up Configuration Parameters for Code Generation

- 1 Open the Configuration Parameter dialog box by selecting **Simulation > Model Configuration parameters**. You can also use the keyboard shortcut **Ctrl+E**.
- 2 Open the **Solver** pane and select
 - **Solver type:** Fixed-Step
 - **Solver:** discrete (no continuous states)
- 3 Open the **Optimization > Signals and Parameters** pane, and select the **Inline parameters** parameter.
- 4 Open the **Code Generation** pane, and specify `ert.tlc` as the **System Target File**.
- 5 Clear **Generate makefile**.
- 6 Select **Generate code only**.
- 7 Enable the HTML report generation by opening the **Code Generation > Report** pane and selecting **Create code generation report**, **Launch report automatically**, and **Code-to-model**. Enabling the HTML report generation is optional.
- 8 Click **Apply** and then **OK** to exit.

Set Up an Example Model With a Stateflow Chart



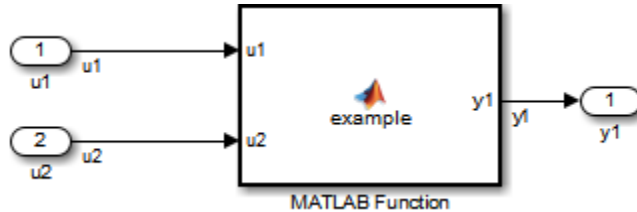
Follow this general procedure to create a simple model containing a Stateflow chart.

- 1 From the **Stateflow > Chart** library, add a Stateflow chart to your model .
- 2 Add Inport blocks and Outport blocks according to the example model.
- 3 Open the **Stateflow Editor** by performing one of the following:
 - Double-click the Stateflow chart.
 - Press **Ctrl+R**.
- 4 Select **Chart > Add Inputs & Outputs > Data Input from Simulink** to add the inputs to the chart. A Data dialog box opens for each input.
- 5 Specify the **Name** (u1, u2, ...) and the **Type** (Inherit: Same as Simulink) for each input, unless specified differently in the example. Click **OK**.

Click **Apply** and close each dialog box.

- 6 Select **Chart > Add Inputs & Outputs > Data Output from Simulink** to add the outputs to the chart. A Data dialog opens for each output.
- 7 Specify the **Name** (y1, y2, ...) and **Type** (Inherit: Same as Simulink) for each output, unless specified differently in the example. Click **OK**.
- 8 Click **Apply** and close each dialog box.
- 9 In the **Stateflow Editor**, create the Stateflow diagram specific to the example.
- 10 The inputs and outputs appear on the chart in your model.
- 11 Connect the Inport and Outport blocks to the Stateflow Chart.
- 12 Configure the input and output signals; see “Configure a Signal” on page 4-4.

Set Up an Example Model With a MATLAB Function Block



- 1 Add the number of Inport and Outport blocks according to a C construct example included in this chapter.
- 2 From the Simulink User-defined Functions library drag a MATLAB Function block into the model.
- 3 Double-click the block. The MATLAB Function Block Editor opens. Edit the function to implement your application.
- 4 Click **File** > **Save** and close the MATLAB Function Block Editor.
- 5 Connect the Inport and Outport blocks to the MATLAB Function block. See “Configure a Signal” on page 4-4.
- 6 Save your model.

Data Declaration

C Construct

```
int32 p1 = 3;
```

Declare a Variable for a Block Parameter Using a Data Object

You can specify certain block parameters as a variable. If you define the variable as a data object, the variable is global. Where the variable is declared in the generated code depends on the custom storage class that you choose (and whether you select **Inline Parameters** on the **Optimization > Signals and Parameters** pane). If you choose **Inline Parameters**, then the data object name is used in the generated code. If you did not choose **Inline Parameters**, the generated code creates a global structure that stores all of the parameters. For more information on how to create a data object, see “Create and Apply User-Defined Data Types”.

Block	Parameter
Constant	Value
Gain	Value
For Iterator	Iteration Limit

There are several methods for configuring data objects:

- For a model with many parameters, use the Data Object Wizard, which analyzes your model and finds the unresolved data objects and data types. You can then create the data objects in the Data Object Wizard. The procedure for using the Data Object Wizard for a parameter is similar to the procedure for a signal. For an example, see “Declare a Variable for a Signal using a Data Object” on page 4-10.
- To add, delete, edit, and configure data objects, use the base workspace in the Model Explorer.
- To create and configure data objects, use the MATLAB command line.

The following example demonstrates how to create a data object using the Model Explorer. The declaration statement in the generated code is as follows:

```
int Kp = 3;
```


- 1 Create a model containing a Constant block and a Gain block.
- 2 Press **Ctrl+E** to open the Configuration Parameters dialog box.
- 3 On the **Optimization > Signals and Parameters** pane of the Configuration Parameter dialog box, select **Inline parameters**.
- 4 Click **Apply** and **OK**. The Configuration Parameter dialog box closes.
- 5 In your model, double-click the Constant block. The Block Parameters dialog box opens.
- 6 In the **Value** field, enter a variable name. In this example, the variable name is **p1**.
- 7 In your model, double-click the Gain block. The Block Parameters dialog box opens.
- 8 In the **Value** field, enter a variable name. In this example, the variable name is **p2**.
- 9 Press **Ctrl+H** to open the Model Explorer. On the Model Hierarchy pane, select the base workspace.
- 10 Add two `mpt` parameter objects. Select **Add > Add Custom**. On the **Contents of: Base Workspace** pane, you see the parameters.
- 11 Double-click each `mpt.Parameter` object and change their names to **p1** and **p2**.
- 12 Click the **p1** parameter. The data object parameters are displayed in the right pane of the Model Explorer.
- 13 In the **Value** field, enter **3** for **p1**. For the **Data type**, select `int32`. Because you chose an `mpt` parameter, the **Storage Class** is already set to `Global(Custom)`.
- 14 In the **Value** field, enter **5** for **p2**. For the **Data type**, select `int32`.
- 15 Press **Ctrl+B** to generate code.

In the `model.c` file you see:

```
int32 p1 = 3;  
int32 p2 = 5;
```

Note: Depending on the storage class, the global variable is represented differently in the generated code. For more information, see “Parameter Objects”.

C Construct

```
int p1 = 3;
```

Declare a Variable for a Signal using a Data Object

- 1 Create a model and label the signals.
- 2 Open the **Data Object Wizard**. In the Simulink Editor, select **Code > Data Objects > Data Object Wizard**. If you are not familiar with creating Simulink Data Objects using the wizard, refer to “Data Object Wizard” .
- 3 Click **Find**. The list of unresolved parameters and objects populates the Data Object Wizard. You can do mass edits for identical data objects.
- 4 Select the signals individually or select all signals by clicking **Check All**.
- 5 From the parameter **Choose package for selected data objects** drop-down list, select the `mpt` package. Click **Apply Package**. When you open the Model Explorer the data objects appear in the base workspace.
- 6 In the base workspace, click the `p1` data object . The data object parameters appear in the right pane of the Model Explorer.
- 7 From the **Data type** drop-down list, select `int16`.
- 8 You can also specify the storage class. The data object is an `mpt.Parameter` object, therefore the Storage Class is automatically set to `Global (Custom)`.

Note: The Storage class alters the data object implementation in the generated code. For more information, see “Signal Objects”.

Data Type Conversion

C Construct

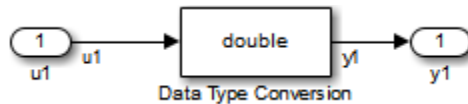
```
y1 = (double)u1;
```

Modeling Patterns

- “Modeling Pattern for Data Type Conversion — Simulink Block” on page 4-11
- “Modeling Pattern for Data Type Conversion — Stateflow Chart” on page 4-12
- “Modeling Pattern for Data Type Conversion — MATLAB Function Block” on page 4-12

Modeling Pattern for Data Type Conversion — Simulink Block

One method to create a data type conversion is to use a Data Type Conversion block from the **Simulink > Commonly Used Blocks** library.



ex_data_type_SL

- 1 From the **Commonly Used Blocks** library, drag a Data Type Conversion block into your model and connect to the Inport and Output blocks.
- 2 Double-click on the Data Type Conversion block to open the Block Parameters dialog box.
- 3 Select the **Output data type** parameter as **double**.
- 4 Press **Ctrl+B** to build the model and generate code.

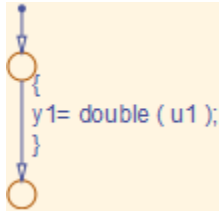
The generated code appears in `ex_data_type_SL.c`, as follows:

```
int32_T u1;
real_T y1;

void ex_data_type_SL_step(void)
{
    y1 = (real_T)u1;
```

```
}
The Embedded Coder type definition for double is real_T.
```

Modeling Pattern for Data Type Conversion — Stateflow Chart



Stateflow Chart Type Conversion

Procedure

- 1 Follow the steps for “Set Up an Example Model With a Stateflow Chart” on page 4-6 . This example contains one Inport block and one Outport block.
- 2 Name the example model `ex_data_type_SF`.
- 3 Double-click the Inport block and select the **Signal Attributes** tab. Specify the **Data Type** as `int32` from the drop down menu.
- 4 Double-click the Outport block and select the **Signal Attributes** tab. Specify the **Data Type** as `Inherit: auto` from the drop down menu.
- 5 Press **Ctrl+B** to build the model and generate code.

Results

The generated code appears in `ex_data_type_SF.c`, as follows:

```
int32_T u1;
real_T y1;
void ex_data_type_SF_step(void)
{
    y1 = (real_T)u1;
}
```

Modeling Pattern for Data Type Conversion — MATLAB Function Block

Procedure

- 1 Follow the steps for “Set Up an Example Model With a MATLAB Function Block” on page 4-7 . This example model contains one Inport block and one Outport block.

- 2 Name the model `ex_data_type_ML_Func`.
- 3 In the MATLAB Function Block Editor enter the function, as follows:

```
function y1 = typeconv(u1)
y1 = double(u1);
end
```

- 4 Press **Ctrl+B** to build the model and generate code.

Results

The generated code appears in `ex_data_type_ML_func.c`, where `real32_T` is a float and `real_T` is a double. Type conversion occurs across assignments.

```
real32_T u1;
real_T y1;

void ex_data_type_ML_func_step(void)
{
    y1 = u1;
}
```

Other Type Conversions in Modeling

Type conversions can also occur on the output of blocks where the output variable is specified as a different data type. For example, in the Gain block, you can select the **Inherit via internal rule** parameter to control the output signal data type. Another example of type conversion can occur at the boundary of a Stateflow chart. You can specify the output variable as a different data type.

Type Qualifiers

Modeling Patterns for Type Qualifiers

- “Using a Tunable Parameter in the Base Workspace” on page 4-14
- “Use a Data Object of the Const Custom Storage Class” on page 4-15

Using a Tunable Parameter in the Base Workspace

A tunable parameter is a block parameter whose value can be changed at runtime. The storage class property of a parameter specifies how the parameter is declared in the generated code.



ex_type_qual

Procedure

- 1 Create a model containing a Constant block and an Outport block.
- 2 Double-click the Constant block. In the **Constant value** field, enter the parameter name p1 .
- 3 In the base workspace, create a MATLAB variable for p1 and specify its **Value** as 9.8 and its **Data type** as double.
- 4 Press **Ctrl+E** to open the Configuration Parameters dialog box and select the **Optimization > Signals and Parameters** pane.
- 5 Select the **Inline parameters** parameter, which activates the **Configure** button.
- 6 Click the **Configure** button to open the Model Parameter Configuration dialog box.
- 7 To declare a tunable parameter, from the **Source list**, select the variable p1.
- 8 Click the **Add to table** button to add p1 to the **Global (tunable) parameters** section.
- 9 Click the **Storage Class** and select Exported Global.
- 10 Click the **Storage Type Qualifier** arrow and select const.

- 11 Click **Apply** to save the changes.
- 12 Press **Ctrl+B** to build the model and generate code.

Results

The generated code appears in `ex_type_qual.c` as follows:

```
/* Exported block parameters */
const real_T p1 = 9.8;          /* Variable: p1
                               * Referenced by: '<Root>/Constant'
                               */
```

Use a Data Object of the Const Custom Storage Class

One way to create a type qualifier in the generated code is to create a data object and specify a custom storage class. Use the previous model, `ex_type_qual`, for this example. Specify `p1` differently as follows:

Procedure

- 1 Press **Ctrl+H** to open the Model Explorer. On the Model Hierarchy pane, select the base workspace.
- 2 Select **Add > Add Custom** to add an `mpt` parameter object. The parameter is displayed in the **Contents of: Base Workspace** pane.
- 3 Double-click the `mpt.Parameter` object and change the **Name** to **p1**.
- 4 Click the **p1** parameter which displays the data object parameters on the right pane of the Model Explorer.
- 5 In the **Value** field, enter **9.8** for `p1`. Specify the **Data type** as **auto** for 64-bit double.
- 6 You can use the different type qualifiers by selecting a custom storage class from the **Storage class** list. For this example, select **ConstVolatile (custom)**.
- 7 In the Configuration Parameters dialog box, on the **Optimization > Signals and Parameters** pane, select the **Inline parameters**.
- 8 Press **Ctrl+B** to build the model and generate code.

Results

The generated code produces the type qualifier in `ex_type_qual.c`:

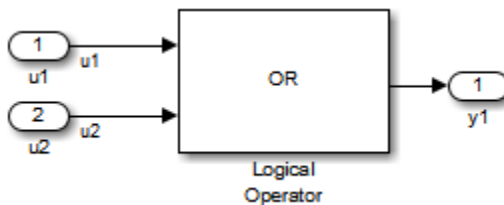
```
const volatile real_T p1 = 9.8;
```

Relational and Logical Operators

Modeling Patterns for Relational and Logical Operators

- “Modeling Pattern for Relational or Logical Operators — Simulink Blocks” on page 4-16
- “Modeling Pattern for Relational and Logical Operators — Stateflow Chart” on page 4-17
- “Modeling Pattern for Relational and Logical Operators — MATLAB Function Block” on page 4-18

Modeling Pattern for Relational or Logical Operators — Simulink Blocks



ex_logical_SL

Procedure

- 1 From the **Logic and Bit Operations** library, drag a Logical Operator block into your model.
- 2 Double-click the block to configure the logical operation. Set the **Operator** field to OR.
- 3 Name the blocks, as shown in the model ex_logical_SL.
- 4 Connect the blocks and name the signals, as shown in the model ex_logical_SL.
- 5 Press **Ctrl+B** to build the model and generate code.

Note: You can use the above procedure to implement relational operators by replacing the Logical Operator block with a Relational Operator block.

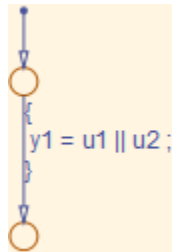
Results

Code implementing the logical operator OR is in the `ex_logical_SL_step` function in `ex_logical_SL.c`:

```
/* Exported block signals */
boolean_T u1;                               /* '<Root>/u1' */
boolean_T u2;                               /* '<Root>/u2' */
boolean_T y1;                               /* '<Root>/Logical Operator' */

/* Logic: '<Root>/Logical Operator' incorporates:
 * Inport: '<Root>/u1'
 * Inport: '<Root>/u2'
 */
y1 = (u1 || u2);
```

Modeling Pattern for Relational and Logical Operators — Stateflow Chart



ex_logical_SF/Logical Operator Stateflow Chart

Procedure

- 1 Follow the steps for “Set Up an Example Model With a Stateflow Chart” on page 4-6. This example model contains two Inport blocks and one Outport block.
- 2 Name the example model `ex_logical_SF`.
- 3 In the **Stateflow Editor**, specify the **Data Type** for `y1` as **Boolean**.
- 4 In the **Stateflow Editor**, create the Stateflow diagram as shown. The relational or logical operation actions are on the transition from one junction to another. Relational statements specify conditions to conditionally allow a transition. In that case, the statement would be within square brackets.
- 5 Press **Ctrl+B** to build the model and generate code.

Results

Code implementing the logical operator OR is in the `ex_logical_SF_step` function in `ex_logical_SF.c`:

```
boolean_T u1;           /* '<Root>/u1' */
boolean_T u2;           /* '<Root>/u2' */
boolean_T y1;           /* '<Root>/Chart' */

void ex_logical_SF_step(void)
{
    y1 = (u1 || u2);
}
```

Modeling Pattern for Relational and Logical Operators — MATLAB Function Block

This example demonstrates the MATLAB Function block method for incorporating operators into the generated code using a relational operator.

Procedure

- 1 Follow the steps for “Set Up an Example Model With a MATLAB Function Block” on page 4-7 . This example model contains two Inport blocks and one Outport block.
- 2 Name the example model `ex_rel_operator_ML`.
- 3 In the MATLAB Function Block Editor enter the function, as follows:

```
function y1 = fcn(u1, u2)
y1 = u1 > u2;
end
```

- 4 Press **Ctrl+B** to build the model and generate code.

Results

Code implementing the relational operator '>' is in the `ex_rel_operator_ML_step` function in `ex_rel_operator_ML.c`:

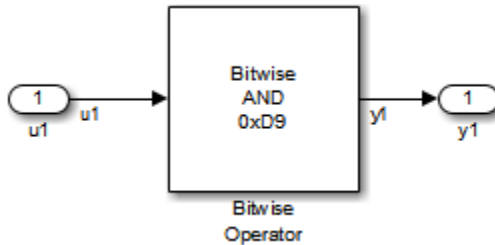
```
real_T u1;           /* '<Root>/u1' */
real_T u2;           /* '<Root>/u2' */
boolean_T y;         /* '<Root>/MATLAB Function' */

void ex_rel_operator_ML_step(void)
```

```
{  
  y = (u1 > u2);  
}
```

Bitwise Operations

Simulink Bitwise-Operator Block



ex_bit_logic_SL

Procedure

- 1 Drag a Bitwise Operator block from the **Logic and Bit Operations** library into your model.
- 2 Double-click the block to open the Block Parameters dialog.
- 3 Select the type of **Operator**. In this example, select **AND**.
- 4 In order to perform Bitwise operations with a bit-mask, select **Use bit mask**.

Note: If another input uses Bitwise operations, clear the **Use bit mask** parameter and enter the number of input ports.

- 5 In the **Bit Mask** field, enter a decimal number. Use `bin2dec` or `hex2dec` to convert from binary or hexadecimal. In this example, enter `hex2dec('D9')`.
- 6 Name the blocks, as shown in, model `ex_bit_logic_SL`.
- 7 Connect the blocks and name the signals, as shown in, model `ex_bit_logic_SL`.
- 8 Press **Ctrl+B** to build the model and generate code.

Results

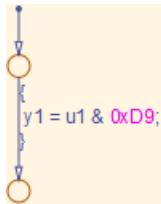
Code implementing the logical operator OR is in the `ex_bit_logic_SL_step` function in `ex_bit_logic_SL.c`:

```
uint8_T u1;
```

```
uint8_T y1;

void ex_bit_logic_SL_step(void)
{
    y1 = (uint8_T)(u1 & 217);
}
```

Stateflow Chart



ex_bit_logic_SF/Bit_Logic Stateflow Chart

Procedure

- 1 Follow the steps for “Set Up an Example Model With a Stateflow Chart” on page 4-6. This example contains one Inport block and one Outport block.
- 2 Name the example model `ex_bit_logic_SF`.
- 3 From the **Stateflow Editor**, select **Tools > Explore** to open the Model Explorer.
- 4 In the Model Explorer, on the right pane, select **Enable C-bit operations**.
- 5 In the **Stateflow Editor**, create the Stateflow diagram, `ex_bit_logic_SF/Bit_Logic`.
- 6 Press **Ctrl+B** to build the model and generate code.

Results

Code implementing the logical operator OR is in the `ex_bit_logic_SF_step` function in `ex_bit_logic_SF.c`:

```
uint8_T u1;
uint8_T y1;

void bit_logic_SF_step(void)
{
    y1 = (uint8_T)(u1 & 0xD9);
}
```

```
}
```

MATLAB Function Block

In this example, to demonstrate the MATLAB Function block method for implementing bitwise logic into the generated code, use the bitwise OR, '|'.

Procedure

- 1 Follow the steps for “Set Up an Example Model With a MATLAB Function Block” on page 4-7. This example model contains two Inport blocks and one Outport block.
- 2 Name your model `ex_bit_logic_ML`.
- 3 In the MATLAB Function Block Editor enter the function, as follows:

```
function y1 = fcn(u1, u2)

y1 = bitor(u1, u2);
end
```

- 4 Press **Ctrl+B** to build the model and generate code.

Results

Code implementing the bitwise operator OR is in the `ex_bit_logic_ML_step` function in `ex_bit_logic_ML.c`:

```
uint8_T u1;
uint8_T u2;
uint8_T y1;

void ex_bit_logic_ML_step(void)
{
    y1 = (uint8_T)(u1 | u2);
}
```

If-Else

C Construct

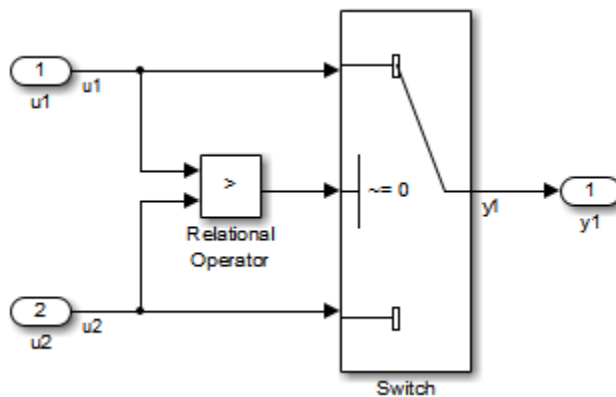
```
if (u1 > u2)
{
    y1 = u1;
}
else
{
    y1 = u2;
}
```

Modeling Patterns

- “Modeling Pattern for If-Else: Switch block” on page 4-23
- “Modeling Pattern for If-Else: Stateflow Chart” on page 4-25
- “Modeling Pattern for If-Else: MATLAB Function Block” on page 4-26

Modeling Pattern for If-Else: Switch block

One method to create an if-else statement is to use a Switch block from the **Simulink** > **Signal Routing** library.



Model `ex_if_else_SL`

Procedure

- 1 Drag the Switch block from the **Simulink>Signal Routing** library into your model.
- 2 Connect the data inputs and outputs to the block.
- 3 Drag a Relational Operator block from the Logic & Bit Operations library into your model.
- 4 Connect the signals that are used in the if-expression to the Relational Operator block. The order of connection determines the placement of each signal in the if-expression.
- 5 Configure the Relational Operator block to be a greater than operator.
- 6 Connect the controlling input to the middle input port of the Switch block.
- 7 Double-click the Switch block and set **Criteria for passing first input** to `u2~=0`. The software selects `u1` if `u2` is TRUE; otherwise `u2` passes.
- 8 Enter `Ctrl+B` to build the model and generate code.

Results

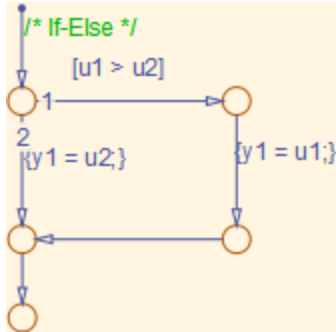
The generated code includes the following `ex_if_else_SL_step` function in the file `ex_if_else_SL.c`:

```
/* External inputs (root inport signals with auto storage) */
ExternalInputs U;

/* External outputs (root outports fed by signals with auto storage) */
ExternalOutputs Y;

/* Model step function */
void ex_if_else_SL_step(void)
{
    /* Switch: '<Root>/Switch' incorporates:
     * Inport: '<Root>/u1'
     * Inport: '<Root>/u2'
     * Output: '<Root>/y1'
     * RelationalOperator: '<Root>/Relational Operator'
     */
    if (U.u1 > U.u2) {
        Y.y1 = U.u1;
    } else {
        Y.y1 = U.u2;
    }
}
```


Modeling Pattern for If-Else: Stateflow Chart



ex_if_else_SF/Chart

Procedure

- 1 Follow the steps for “Set Up an Example Model With a Stateflow Chart” on page 4-6. This example model contains two Inport blocks and one Output block.
- 2 Name your model `ex_if_else_SF`.
- 3 When configuring your Stateflow chart, select **Chart > Add Patterns > Decision > If-Else**. The Stateflow Pattern dialog opens. Fill in the fields as follows:

Description	If-Else (optional)
If condition	<code>u1 > u2</code>
If action	<code>y1 = u1</code>
Else action	<code>y1 = u2</code>

- 4 Press **Ctrl+B** to build the model and generate code.

Results

The generated code includes the following `ex_if_else_SF_step` function in the file `If_Else_SF.c`:

```

/* External inputs (root inport signals with auto storage) */
ExternalInputs U;

/* External outputs (root outports fed by signals with auto storage) */
ExternalOutputs Y;

/* Model step function */

```

```
void ex_if_else_SF_step(void)
{
  /* Stateflow: '<Root>/Chart' incorporates:
   * Inport: '<Root>/u1'
   * Inport: '<Root>/u2'
   * Outport: '<Root>/y1'
   */
  /* Gateway: Chart */
  /* During: Chart */
  /* Transition: '<S1>:14' */
  /* If-Else */
  if (U.u1 > U.u2) {
    /* Transition: '<S1>:13' */
    /* Transition: '<S1>:12' */
    Y.y1 = U.u1;

    /* Transition: '<S1>:11' */
  } else {
    /* Transition: '<S1>:10' */
    Y.y1 = U.u2;
  }

  /* Transition: '<S1>:9' */
}
```

Modeling Pattern for If-Else: MATLAB Function Block

Procedure

- 1 Follow the steps for “Set Up an Example Model With a MATLAB Function Block” on page 4-7. This example model contains two Inport blocks and one Outport block.
- 2 Name your model `ex_if_else_ML`.
- 3 In the MATLAB Function Block Editor enter the function, as follows:

```
function y1 = fcn(u1, u2)
if u1 > u2;
    y1 = u1;
else y1 = u2;
end
```

- 4 Press **Ctrl+B** to build the model and generate code.

Results

The generated code includes the following `ex_if_else_ML_step` function in the file `ex_if_else_ML.c`:

```
/* External inputs (root inport signals with auto storage) */
ExternalInputs U;

/* External outputs (root outports fed by signals with auto storage) */
ExternalOutputs Y;
```

```
/* Model step function */
void ex_if_else_ML_step(void)
{
    /* MATLAB Function Block: '<Root>/MATLAB Function' incorporates:
     * Inport: '<Root>/u1'
     * Inport: '<Root>/u2'
     * Outport: '<Root>/y1'
     */
    /* MATLAB Function 'MATLAB Function': '<S1>:1' */
    if (U.u1 > U.u2) {
        /* '<S1>:1:4' */
        /* '<S1>:1:5' */
        Y.y1 = U.u1;
    } else {
        /* '<S1>:1:6' */
        Y.y1 = U.u2;
    }
}
```

Switch

C Construct

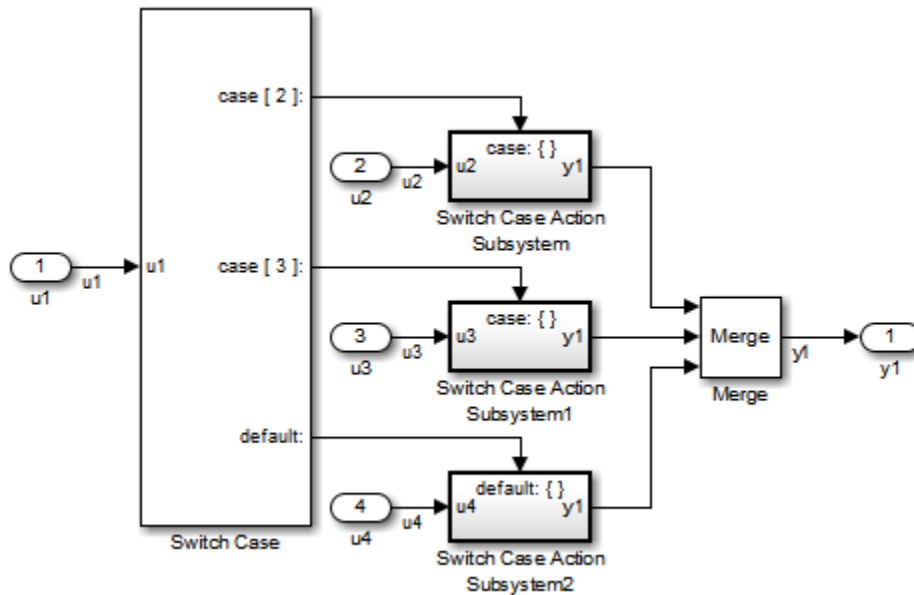
```
switch (u1)
{
  case 2:
    y1 = u2;
    break;
  case 3:
    y1 = u3;
    break;
  default:
    y1 = u4;
    break;
}
```

Modeling Patterns

- “Modeling Pattern for Switch: Switch Case block” on page 4-28
- “Modeling Pattern for Switch: MATLAB Function block” on page 4-31
- “Convert If-Elseif-Else to Switch statement” on page 4-32

Modeling Pattern for Switch: Switch Case block

One method for creating a `switch` statement is to use a Switch Case block from the **Simulink > Ports and Subsystems** library.



Model ex_switch_SL

Procedure

- 1 Drag a Switch Case block from the **Simulink > Ports and Subsystems** library into your model.
- 2 Double-click the block. In the Block Parameters dialog box, fill in the **Case Conditions** parameter. In this example, the two cases are: {2,3}.
- 3 Select the **Show default case** parameter. The default case is optional in a switch statement.
- 4 Connect the condition input u1 to the input port of the Switch block.
- 5 Drag Switch Case Action Subsystem blocks from the **Simulink>Ports and Subsystems** library to correspond with the number of cases.
- 6 Configure the “Switch Case Action Subsystem” subsystems.
- 7 Drag a Merge block from the **Simulink > Signal Routing** library to merge the outputs.
- 8 The Switch Case block takes an integer input, therefore, the input signal u1 is type cast to an int32.

9 Enter Ctrl+B to build the model and generate code.

Results

The generated code includes the following `ex_switch_SL_step` function in the file `ex_switch_SL.c`:

```
/* Exported block signals */
int32_T u1;                                /* '<Root>/u1' */

/* External inputs (root inport signals with auto storage) */
ExternalInputs U;

/* External outputs (root outports fed by signals with auto storage) */
ExternalOutputs Y;

/* Model step function */
void ex_switch_SL_step(void)
{
    /* SwitchCase: '<Root>/Switch Case' incorporates:
     * ActionPort: '<S1>/Action Port'
     * ActionPort: '<S2>/Action Port'
     * ActionPort: '<S3>/Action Port'
     * Inport: '<Root>/u1'
     * SubSystem: '<Root>/Switch Case Action Subsystem'
     * SubSystem: '<Root>/Switch Case Action Subsystem1'
     * SubSystem: '<Root>/Switch Case Action Subsystem2'
     */
    switch (u1) {
    case 2:
        /* Inport: '<S1>/u2' incorporates:
         * Inport: '<Root>/u2'
         * Output: '<Root>/y1'
         */
        Y.y1 = U.u2;
        break;

    case 3:
        /* Inport: '<S2>/u3' incorporates:
         * Inport: '<Root>/u3'
         * Output: '<Root>/y1'
         */
        Y.y1 = U.u3;
        break;

    default:
        /* Inport: '<S3>/u4' incorporates:
         * Inport: '<Root>/u4'
         * Output: '<Root>/y1'
         */
        Y.y1 = U.u4;
        break;
    }
}
```

Modeling Pattern for Switch: MATLAB Function block

Procedure

- 1 Follow the steps for “Set Up an Example Model With a MATLAB Function Block” on page 4-7. This example model contains four Inport blocks and one Outport block.
- 2 Name your model `ex_switch_ML`.
- 3 In the MATLAB Function Block Editor enter the function, as follows:

```
function y1 = fcn(u1, u2, u3, u4)

switch u1
    case 2
        y1 = u2;
    case 3
        y1 = u3;
    otherwise
        y1 = u4;
end
```

- 4 Press **Ctrl+B** to build the model and generate code.

Results

The generated code includes the following `ex_switch_ML_step` function in the file `ex_switch_ML.c`:

```
/* External inputs (root inport signals with auto storage) */
ExternalInputs U;

/* External outputs (root outports fed by signals with auto storage) */
ExternalOutputs Y;

/* Model step function */
void ex_switch_ML_step(void)
{
    /* MATLAB Function Block: '<Root>/MATLAB Function' incorporates:
     * Inport: '<Root>/u1'
     * Inport: '<Root>/u2'
     * Inport: '<Root>/u3'
     * Inport: '<Root>/u4'
     * Outport: '<Root>/y1'
     */
    /* MATLAB Function 'MATLAB Function': '<S1>:1' */
    /* '<S1>:1:4' */
    switch (U.u1) {
        case 2:
            /* '<S1>:1:6' */
            Y.y1 = U.u2;
            break;
    }
```

```
case 3:
    /* '<S1>:1:8' */
    Y.y1 = U.u3;
    break;

default:
    /* '<S1>:1:10' */
    Y.y1 = U.u4;
    break;
}
}
```

Convert If-Elseif-Else to Switch statement

If a MATLAB Function block or a Stateflow chart uses `if-elseif-else` decision logic, you can convert it to a `switch` statement by using a configuration parameter. In the Configuration Parameters dialog box, on the **Code Generation > Code Style** pane, select the “Convert if-elseif-else patterns to switch-case statements” parameter. For more information, see “Converting If-Elseif-Else Code to Switch-Case Statements” in the Simulink documentation. For more information on this conversion using a Stateflow chart, see “Convert If-Elseif-Else Code to Switch-Case Statements” and “Example of Converting Code to Switch-Case Statements” in the Stateflow documentation.

For Loop

C Construct

```

y1 = 0;
for(inx = 0; inx <10; inx++)
{
    y1 = u1[inx] + y1;
}

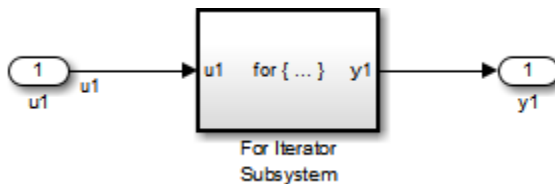
```

Modeling Patterns:

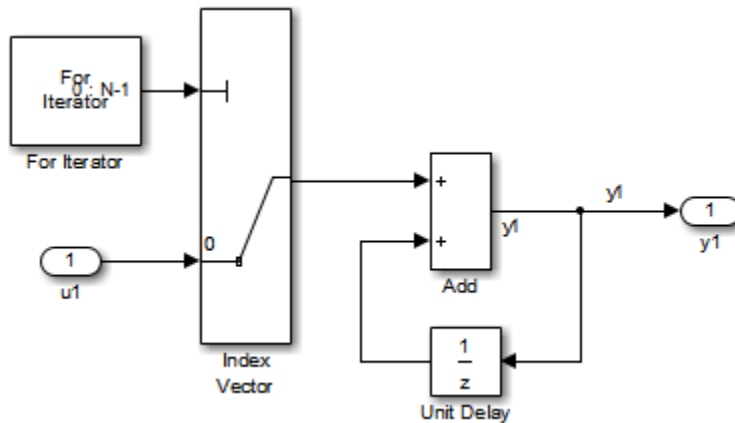
- “Modeling Pattern for For Loop: For-Iterator Subsystem block” on page 4-33
- “Modeling Pattern for For Loop: Stateflow Chart” on page 4-36
- “Modeling Pattern for For Loop: MATLAB Function block” on page 4-37

Modeling Pattern for For Loop: For-Iterator Subsystem block

One method for creating a for loop is to use a For Iterator Subsystem block from the **Simulink > Ports and Subsystems** library.



Model ex_for_loop_SL



For Iterator Subsystem

Procedure

- 1 Drag a “For Iterator Subsystem” block from the **Simulink > Ports and Subsystems** library into your model.
- 2 Connect the data inputs and outputs to the For Iterator Subsystem block.
- 3 Open the Inport block.
- 4 In the Block Parameters dialog box, select the **Signal Attributes** pane and set the **Port dimensions** parameter to 10.
- 5 Double-click the For Iterator Subsystem block to open the subsystem.
- 6 Drag an Index Vector block from the Signal-Routing library into the subsystem.
- 7 Open the For Iterator block. In the Block Parameters dialog box set the **Index-mode** parameter to **Zero-based** and the **Iteration limit** parameter to 10.
- 8 Connect the controlling input to the topmost input port of the Index Vector block, and the other input to the second port.
- 9 Drag an Add block from the **Math Operations** library into the subsystem.
- 10 Drag a Unit Delay block from **Commonly Used Blocks** library into the subsystem.
- 11 Double-click the Unit Delay block and set the **Initial Conditions** parameter to 0. This parameter initializes the state to zero.
- 12 Connect the blocks as shown in the model diagram.
- 13 Save the subsystem and the model.

14 Enter Ctrl+B to build the model and generate code.

Results

The generated code includes the following `ex_for_loop_SL_step` function in the file `ex_for_loop_SL.c`:

```

    /* External inputs (root inport signals with auto storage) */
    ExternalInputs U;

    /* External outputs (root outports fed by signals with auto storage) */
    ExternalOutputs Y;

    /* Model step function */
    void ex_for_loop_SL_step(void)
    {
        int32_T s1_iter;
        int32_T rtb_y1;

        /* Outputs for iterator SubSystem: '<Root>/For Iterator Subsystem' incorporates:
        * ForIterator: '<S1>/For Iterator'
        */
        for (s1_iter = 0; s1_iter < 10; s1_iter++) {
            /* Sum: '<S1>/Add' incorporates:
            * Inport: '<Root>/u1'
            * MultiPortSwitch: '<S1>/Index Vector'
            * UnitDelay: '<S1>/Unit Delay'
            */
            rtb_y1 = U.u1[s1_iter] + DWork.UnitDelay_DSTATE;

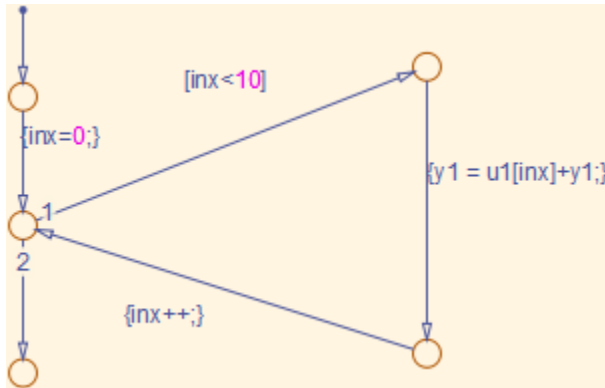
            /* Update for UnitDelay: '<S1>/Unit Delay' */
            DWork.UnitDelay_DSTATE = rtb_y1;
        }

        /* end of Outputs for SubSystem: '<Root>/For Iterator Subsystem' */

        /* Outport: '<Root>/y1' */
        Y.y1 = rtb_y1;
    }

```

Modeling Pattern for For Loop: Stateflow Chart



Procedure

- 1 Follow the steps for “Set Up an Example Model With a Stateflow Chart” on page 4-6. This example model contains one Inport block and one Outport block.
- 2 Name the model `ex_for_loop_SF`.
- 3 Enter `Ctrl+R` to open the Model Explorer.
- 4 In the Model Explorer, select the output variable, `u1`, and in the right pane, select the **General** tab and set the **Initial Value** to 0.
- 5 In the **Stateflow Editor**, select **Chart > Add Patterns > Loop > For**. The Stateflow Pattern dialog opens.
- 6 Fill in the fields in the Stateflow Pattern dialog box as follows:

Description	For Loop (optional)
Initializer expression	<code>inx = 0</code>
Loop test expression	<code>inx < 10</code>
Counting expression	<code>inx++</code>
For loop body	<code>y1 = u1[inx] + y1</code>

The Stateflow diagram is shown.

- 7 Press `Ctrl+B` to build the model and generate code.

Results

The generated code includes the following `ex_for_loop_SF_step` function in the file `ex_for_loop_SF.c`:

```

/* Block signals (auto storage) */
BlockIO B;

/* External inputs (root inport signals with auto storage) */
ExternalInputs U;

/* External outputs (root outports fed by signals with auto storage) */
ExternalOutputs Y;

/* Model step function */
void ex_for_loop_SF_step(void)
{
    int32_T sf_inx;

    /* Stateflow: '<Root>/Chart' incorporates:
     * Inport: '<Root>/u1'
     */
    /* Gateway: Chart */
    /* During: Chart */
    /* Transition: '<S1>:24' */
    /* For Loop */
    /* Transition: '<S1>:25' */
    for (sf_inx = 0; sf_inx < 10; sf_inx++) {
        /* Transition: '<S1>:22' */
        /* Transition: '<S1>:23' */
        B.y1 = U.u1[sf_inx] + B.y1;

        /* Transition: '<S1>:21' */
    }

    /* Transition: '<S1>:20' */

    /* Outport: '<Root>/y1' */
    Y.y1 = B.y1;
}

```

Modeling Pattern for For Loop: MATLAB Function block

Procedure

- 1 Follow the directions for “Set Up an Example Model With a MATLAB Function Block” on page 4-7. This example model contains one Inport block and one Outport block.
- 2 Name your model `ex_for_loop_ML`.
- 3 In the MATLAB Function Block Editor enter the function, as follows:

```
function y1 = fcn(u1)
```

```
y1 = 0;

for inx=1:10
    y1 = u1(inx) + y1 ;
end
```

4 Press **Ctrl+B** to build the model and generate code.

Results

The generated code includes the following `ex_for_loop_ML_step` function in the file `ex_for_loop_ML.c`:

```
/* Exported block signals */
real_T u1[10];                /* '<Root>/u1' */
real_T y1;                    /* '<Root>/MATLAB Function' */

/* Model step function */
void ex_for_loop_ML_step(void)
{
    int32_T inx;

    /* MATLAB Function Block: '<Root>/MATLAB Function' incorporates:
     * Inport: '<Root>/u1'
     */
    /* MATLAB Function 'MATLAB Function': '<S1>:1' */
    /* '<S1>:1:3' */
    y1 = 0.0;
    for (inx = 0; inx < 10; inx++) {
        /* '<S1>:1:5' */
        /* '<S1>:1:6' */
        y1 = u1[inx] + y1;
    }
}
```

While Loop

C Construct

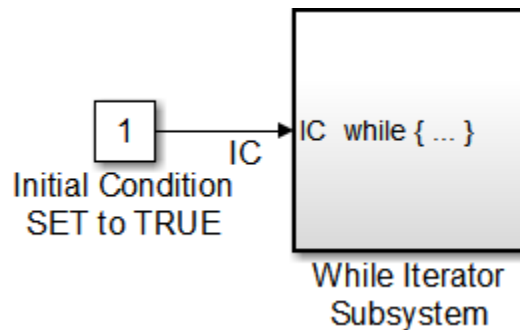
```
while(flag && (num_iter <= 100)
{
    flag = func ();
    num_iter ++;
}
```

Modeling Patterns

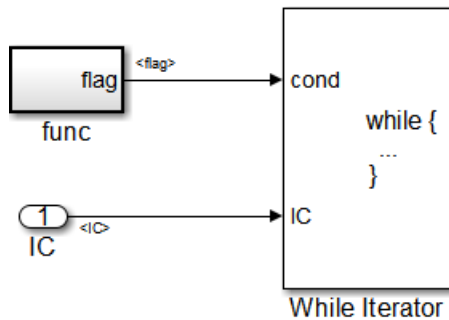
- “Modeling Pattern for While Loop: While Iterator Subsystem block” on page 4-39
- “Modeling Pattern for While Loop: Stateflow Chart” on page 4-42
- “Modeling Pattern for While Loop: MATLAB Function Block” on page 4-45

Modeling Pattern for While Loop: While Iterator Subsystem block

One method for creating a `while` loop is to use a While Iterator Subsystem block from the **Simulink > Ports and Subsystems** library.



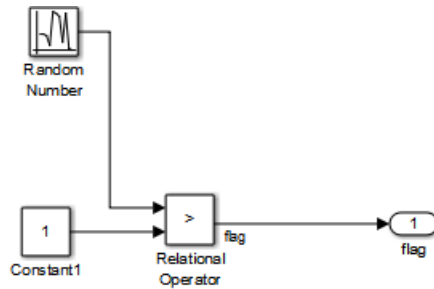
Model `ex_while_loop_SL`



ex_while_loop_SL/While Iterator Subsystem

Procedure

- 1 Drag a While Iterator Subsystem block from the **Simulink > Ports and Subsystems** library into the model.
- 2 Drag a Constant block from the **Simulink > Commonly Used Blocks** library into the model. In this case, set the **Initial Condition** to 1 and the **Data Type** to **Boolean**. You do not have to set the initial condition to **FALSE**. The initial condition can be dependent on the input to the block.
- 3 Connect the Constant block to the While Iterator Subsystem block.
- 4 Double-click the While Iterator Subsystem block to open the subsystem.
- 5 Place a Subsystem block next to the While Iterator block.
- 6 Right-click the subsystem and select **Block Parameters (Subsystem)**. The Block Parameters dialog box opens.
- 7 Select the **Treat as atomic unit** parameter to configure the subsystem to generate a function. This enables parameters on the **Code Generation** tab.
- 8 Select the **Code Generation** tab. From the **Function packaging** list, select the option, **Nonreusable function**.
- 9 From the **Function name options** list, select the option, **User specified**. The **Function name** parameter is displayed.
- 10 Specify the name as **func**.
- 11 Click **Apply**.
- 12 Double-click the **func** subsystem block. In this example, function **func()** has an output **flag** set to 0 or 1 depending on the result of the algorithm in **func()**. Create the **func()** algorithm as shown in the following diagram:



func

- 13 Double-click the While Iterator block to set the **Maximum number of iterations** to 100.
- 14 Connect blocks as shown in the model and subsystem diagrams.

Results

The generated code includes the following `ex_while_loop_SL_step` function in the file `ex_while_loop_SL.c`:

```

/* Exported block signals */
boolean_T IC; /* '<Root>/Initial Condition SET to TRUE' */
boolean_T flag; /* '<S2>/Relational Operator' */

/* Block states (auto storage) */
D_Work DWork;

/* Start for atomic system: '<S1>/func( ) Is a function that updates the flag' */
void func_Start(void)
{
  /* Start for RandomNumber: '<S2>/Random Number' */
  DWork.RandSeed = 1144108930U;
  DWork.NextOutput = rt_NormalRand(&DWork.RandSeed) * 1.7320508075688772E+000;
}

/* Output and update for atomic system:
 * '<S1>/func( ) Is a function that updates the flag' */
void func(void)
{
  /* RelationalOperator: '<S2>/Relational Operator' incorporates:
   * Constant: '<S2>/Constant1'
   * RandomNumber: '<S2>/Random Number'
   */
  flag = (DWork.NextOutput > 1.0);

  /* Update for RandomNumber: '<S2>/Random Number' */
  DWork.NextOutput = rt_NormalRand(&DWork.RandSeed) * 1.7320508075688772E+000;
}

/* Model step function */

```

```

void ex_while_loop_SF_step(void)
{
    int32_T s1_iter;
    boolean_T loopCond;

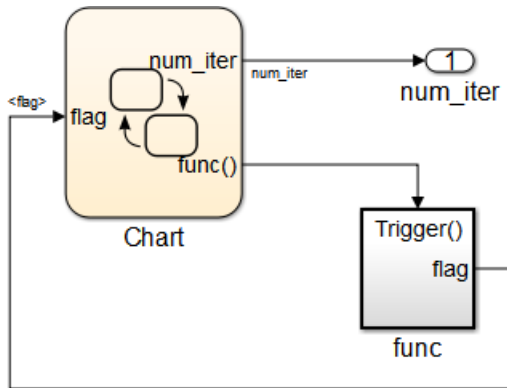
    /* Outputs for iterator SubSystem:
     * '<Root>/While Iterator Subsystem' incorporates:
     * WhileIterator: '<S1>/While Iterator'
     */
    s1_iter = 1;
    loopCond = IC;
    while (loopCond && (s1_iter <= 100)) {
        /* Outputs for atomic SubSystem:
         * '<S1>/func( ) Is a function that updates the flag' */
        func();

        /* end of Outputs for SubSystem:
         * '<S1>/func( ) Is a function that updates the flag' */
        loopCond = flag;
        s1_iter++;
    }

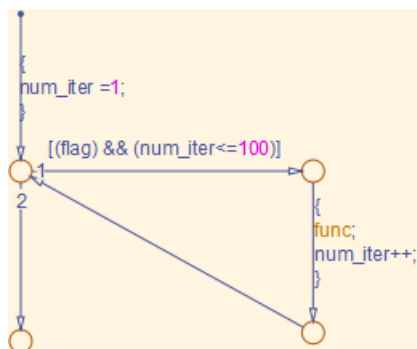
    /* end of Outputs for SubSystem: '<Root>/While Iterator Subsystem' */
}

```

Modeling Pattern for While Loop: Stateflow Chart



Model ex_while_loop_SF



ex_while_loop_SF/Chart Executes the desired while-loop

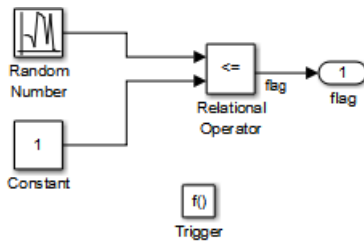
Procedure

- 1 Add a Stateflow Chart to your model from the **Stateflow > Chart** library.
- 2 Double-click the chart.
- 3 Add the input, `flag`, and output, `func`, to the chart and specify their data type.
- 4 Connect the data input and output to the Stateflow chart as shown in the model diagram.
- 5 In the Model Explorer, select the output variable, then, in the right pane, select the **General** tab and set the **Initial Value** to 0.
- 6 Select **Chart > Add Patterns > Loop > While**. The Stateflow Pattern dialog opens.
- 7 Fill in the fields for the Stateflow Pattern dialog box as follows:

Description	While Loop (optional)
While condition	<code>(flag) && (num_iter<=100)</code>
Do action	<code>func; num_iter++;</code>

- 8 Place a Subsystem block in your model.
- 9 Right-click the subsystem and select **Block Parameters (Subsystem)**. The Block Parameters dialog box opens.
- 10 Select the **Treat as atomic unit** parameter to configure the subsystem to generate a function. This enables parameters on the **Code Generation** tab.
- 11 Select the **Code Generation** tab. From the **Function packaging** list, select the option, **Nonreusable function**.

- 12 From the **Function name options** list, select the option, **User specified**. The **Function name** parameter is displayed.
- 13 Specify the name as **func**.
- 14 Click **Apply** to apply the changes.
- 15 Double-click the **func** subsystem block. In this example, function **func** has an output **flag** set to 0 or 1 depending on the result of the algorithm in **func()**. The Trigger block parameter **Trigger type** is **function-call**. Create the **func()** algorithm, as shown in the following diagram:



ex_while_loop_SF/func A function that updates the flag

- 16 Save and close the subsystem.
- 17 Connect blocks to the Stateflow chart as shown in the model diagram for **ex_while_loop_SF**.
- 18 Save your model.

Results

The generated code includes the following **ex_while_loop_SF_step** function in the file **ex_while_loop_SF.c**:

```

/* Exported block signals */
int32_T num_iter; /* '<Root>/Chart Executes the desired while-loop' */
boolean_T flag; /* '<S2>/Relational Operator' */

/* Block states (auto storage) */
D_Work DWork;

/* Model step function */
void ex_while_loop_SF_step(void)
{
/* Stateflow: '<Root>/Chart Executes the desired
 * while-loop' incorporates:
 * SubSystem: '<Root>/func() A function that
 * updates the flag'
 */
/* Gateway: Chart
Executes the desired while-loop */

```

```

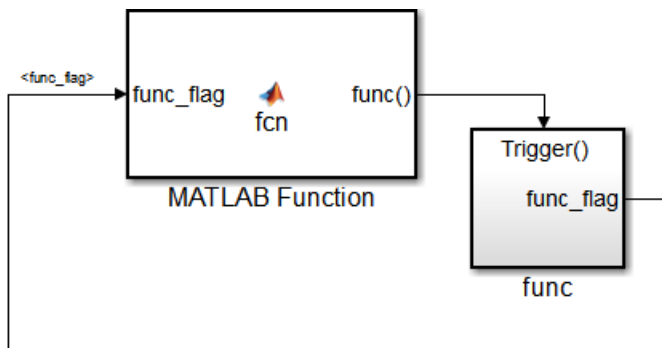
/* During: Chart
   Executes the desired while-loop */
/* Transition: '<S1>:2' */
num_iter = 1;
while (flag && (num_iter <= 100)) {
  /* Transition: '<S1>:3' */
  /* Transition: '<S1>:4' */
  /* Event: '<S1>:12' */
  func();
  num_iter = num_iter + 1;

  /* Transition: '<S1>:5' */
}

/* Transition: '<S1>:1' */
}

```

Modeling Pattern for While Loop: MATLAB Function Block



Model ex_while_loop_ML

Procedure

- 1 In the Simulink Library Browser, click **Simulink > User Defined Functions**, and drag a MATLAB Function block into your model.
- 2 Double-click the MATLAB Function block. The MATLAB Function Block Editor opens.
- 3 In the MATLAB Function Block Editor enter the function, as follows:

```

function fcn(func_flag)

flag = true;
num_iter = 1;

while(flag && (num_iter<=100))

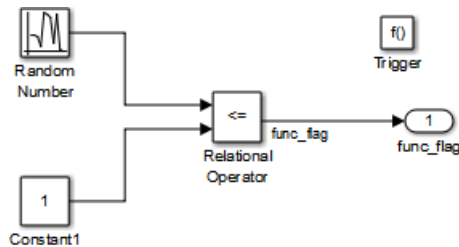
```

```

func;
flag = func_flag;
num_iter = num_iter + 1;
end

```

- 4 Click **Save** and close the MATLAB Function Block Editor.
- 5 Place a Subsystem block in your model, right-click the subsystem and select **Block Parameters (Subsystem)**. The Block Parameters dialog box opens.
- 6 Select the **Treat as atomic unit** parameter to configure the subsystem to generate a function. This enables parameters on the **Code Generation** tab.
- 7 Select the **Code Generation** tab. From the **Function packaging** list, select the option, **Nonreusable function**.
- 8 From the **Function name options** list, select the option, **User specified**. The **Function name** parameter is displayed.
- 9 Specify the name as `func`.
- 10 Click **Apply**.
- 11 Double-click the `func()` subsystem block. In this example, function `func()` has an output `flag` set to 0 or 1 depending on the result of the algorithm in `func()`. The Trigger block parameter **Trigger type** is **function-call**. Create the `func()` algorithm, as shown in the following diagram:



- 12 Save and close the subsystem.
- 13 Connect the MATLAB Function block to the `func()` subsystem.
- 14 Save your model.

Results

The generated code includes the following `while_loop_ML_step` function in the file `while_loop_EML.c`. In some cases an equivalent `for` loop might be generated instead of a `while` loop.

```
/* Exported block signals */
boolean_T func_flag;          /* '<S2>/Relational Operator' */

/* Block states (auto storage) */
D_Work DWork;

/* Model step function */
void while_loop_ML_step(void)
{
    boolean_T func_flag_0;
    boolean_T flag;
    int32_T num_iter;

    /* MATLAB Function Block: '<Root>/MATLAB Function Executes
     * the desired While-Loop' incorporates:
     * SubSystem: '<Root>/func()' updates the "flag"
     */
    func_flag_0 = func_flag;

    /* MATLAB Function 'MATLAB Function
     * Executes the desired While-Loop': '<S1>:1' */
    /* '<S1>:1:3' */
    flag = TRUE;

    /* '<S1>:1:4' */
    num_iter = 1;
    while (flag && (num_iter <= 100);
           num_iter++) {
        /* '<S1>:1:6' */
        /* '<S1>:1:7' */
        func();

        /* '<S1>:1:8' */
        flag = func_flag_0;

        /* '<S1>:1:9' */
        num_iter++;
    }
}
```

Do While Loop

C Construct

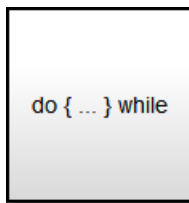
```
num_iter = 1;
do {
    flag = func();
    num_iter++;
}
while (flag && num_iter <= 100)
```

Modeling Patterns

- “Modeling Pattern for Do While Loop: While Iterator Subsystem block” on page 4-48
- “Modeling Pattern for Do While Loop: Stateflow Chart” on page 4-51

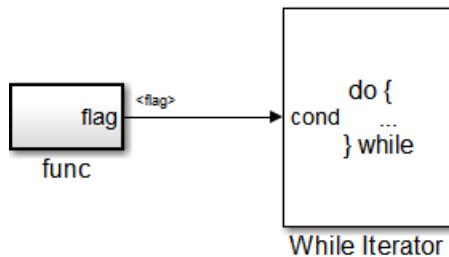
Modeling Pattern for Do While Loop: While Iterator Subsystem block

One method for creating a `while` loop is to use a While Iterator Subsystem block from the **Simulink > Ports and Subsystems** library.



While Iterator
Subsystem

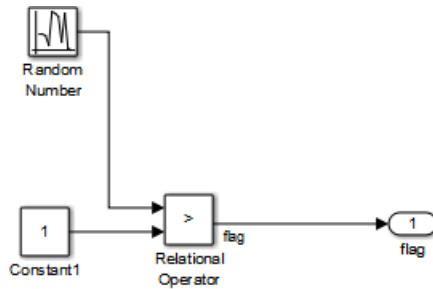
`ex_do_while_loop_SL`



ex_do_while_loop_SL/While Iterator Subsystem

Procedure

- 1 Drag a While Iterator Subsystem block from the **Simulink > Ports and Subsystems** library into the model.
- 2 Double-click the While Iterator Subsystem block to open the subsystem.
- 3 Place a Subsystem block next to the While Iterator block.
- 4 Right-click the subsystem and select **Block Parameters (Subsystem)**. The Block Parameters dialog box opens.
- 5 Select the **Treat as atomic unit** parameter to configure the subsystem to generate a function. This enables parameters on the **Code Generation** tab.
- 6 Select the **Code Generation** tab. From the **Function packaging** list, select the option, Nonreusable function.
- 7 From the **Function name options** list, select the option, User specified. The **Function name** parameter is displayed.
- 8 Specify the name as **func**.
- 9 Click **Apply**.
- 10 Double-click the **func** subsystem block. In this example, function **func** has an output **flag** set to 0 or 1 depending on the result of the algorithm in **func**. Create the **func** algorithm as shown in the following diagram:



ex_do_while_loop_SL/While Iterator Subsystem/func

- 11 Double-click the While Iterator block. This opens the Block Parameters dialog.
- 12 Set the **Maximum number of iterations** to 100.
- 13 Specify the **While loop type** as do-while.
- 14 Connect blocks as shown in the model and subsystem diagrams.
- 15 Enter **Ctrl+B** to generate code.

Results

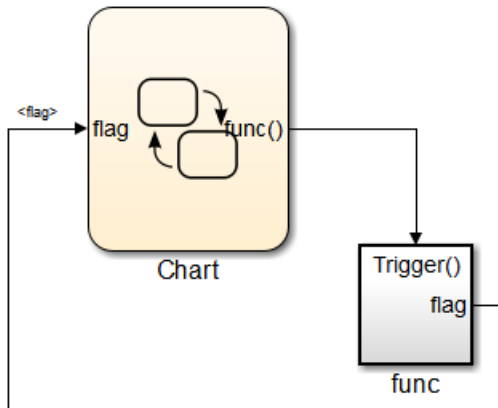
```

void func(void)
{
    flag = (DWork.NextOutput > (real_T)P.Constant1_Value);
    DWork.NextOutput =
        rt_NormalRand(&DWork.RandSeed) * P.RandomNumber_StdDev +
        P.RandomNumber_Mean;
}

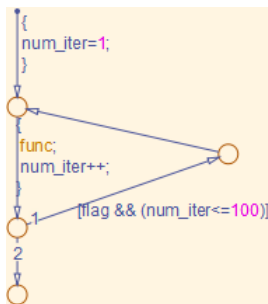
void ex_do_while_loop_SL_step(void)
{
    int32_T s1_iter;

    s1_iter = 1;
    do {
        func();
        s1_iter++;
    } while (flag && (s1_iter <= 100));
}
  
```

Modeling Pattern for Do While Loop: Stateflow Chart



ex_do_while_loop_SF



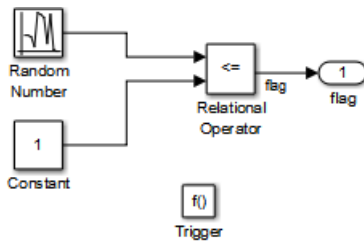
ex_do_while_loop_SF/Chart

- 1 Add a Stateflow Chart to your model from the **Stateflow > Chart** library.
- 2 Double-click the chart to open it.
- 3 Add the inputs and outputs to the chart and specify their data type.
- 4 Connect the data input and output to the Stateflow chart.
- 5 In the Model Explorer, select the output variable, then, in the right pane, select the **General** tab and set the **Initial Value** to 0.

- 6 Select **Chart > Add Patterns > Loop > While**. The Stateflow Pattern dialog opens.
- 7 Fill in the fields for the Stateflow Pattern dialog box as follows:

Description	While Loop (optional)
While condition	(flag) && (num_iter<=100)
Do action	func; num_iter++;

- 8 Place a Subsystem block in your model.
- 9 Right-click the subsystem and select **Block Parameters (Subsystem)**. The Block Parameters dialog box opens.
- 10 Select the **Treat as atomic unit** parameter to configure the subsystem to generate a function. This enables parameters on the **Code Generation** tab.
- 11 Select the **Code Generation** tab. From the **Function packaging** list, select the option, Nonreusable function.
- 12 From the **Function name options** list, select the option, User specified. The **Function name** parameter is displayed.
- 13 Specify the name as func.
- 14 Click **Apply** to apply the changes.
- 15 Double-click the func subsystem block. In this example, function func has an output **flag** set to 0 or 1 depending on the result of the algorithm in func. The Trigger block parameter **Trigger type** is **function-call**. Create the func algorithm, as shown in the following diagram:



ex_do_while_loop_SF/func Updates the flag

- 16 Save and close the subsystem.
- 17 Connect blocks to the Stateflow chart as shown in the model diagram for ex_do_while_loop_SF.
- 18 Save your model.

Results

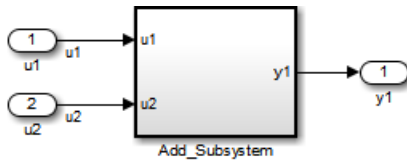
```
void ex_do_while_loop_SF_step(void)
{
    int32_T sf_num_iter;
    num_iter = 1;
    do {
        func();
        num_iter++;
    } while (flag && (sf_num_iter <= 100));
}
```

Function Call

To generate a function call, add a subsystem, which implements the operations that you want.

C Construct

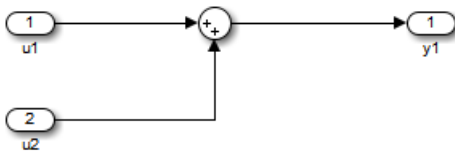
```
void add_function(void)
{
    y1 = u1 + u2;
}
```



ex_function_call

Procedure

- 1 Create a model containing a subsystem. In this example, the subsystem has two inputs and returns one output.
- 2 Double-click the subsystem. Create `Add_Subsystem`, as shown.



ex_function_call/Add_Subsystem

- 3 Right-click the subsystem and select **Block Parameters (Subsystem)** to open the Subsystem Parameters dialog box.
- 4 Select the **Treat as atomic unit** parameter. This enables parameters on the **Code Generation** tab.

Select the **Code Generation** tab. For the **Function packaging** parameter, from the drop-down list, select **Nonreusable function**.

- 5** For the **Function name options** parameter, from the drop-down list, select **User specified**.
- 6** In the **Function name** field, enter the subsystem name, `add_function`.
- 7** Click **Apply** and **OK**.
- 8** Press **Ctrl+B** to build and generate code.

Results

In `ex_function_call.c`, the function is called from `ex_function_call_step`:

```
void ex_function_call_step(void)
{
    add_function();
}
```

The function prototype is externed through the subsystem file, `add_function.h`.

```
extern void add_function(void);
```

The function definition is in the subsystem file `add_function.c`:

```
void add_function(void)
{
    function_call_Y.y1 = u1 + u2;
}
```

Function Prototyping

C Construct

```
double add_function(double u1, double u2)
{
    return u1 + u2;
}
```

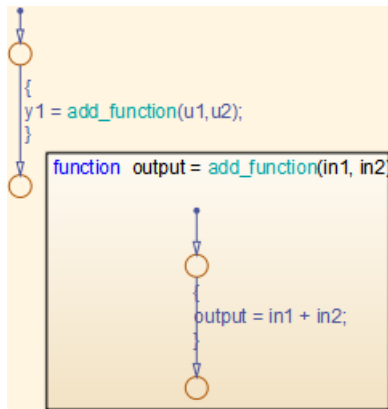
Modeling Patterns

- “Function Call Using Graphical Functions” on page 4-56
- “Control Function Prototype of the model_step Function” on page 4-58

Function Call Using Graphical Functions

Procedure

- 1 Follow the steps for “Set Up an Example Model With a Stateflow Chart” on page 4-6. This example model contains two Inport blocks and one Outport block.
- 2 Name the example model `ex_func_SF`.
- 3 In the **Stateflow Editor**, create a graphical function by clicking the *fx* button and placing a graphical function into the Stateflow chart.
- 4 Edit the graphical function signature to: `output = add_function(u1, u2)`.
- 5 Add the transition action, as shown in the following diagram.



ex_func_SF/Chart

In the Stateflow chart is an example of a simple transition that calls `add_function`.

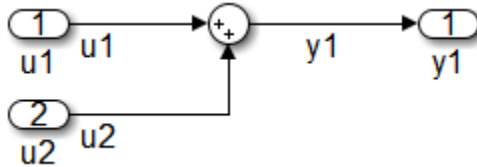
- 6 Open the Model Explorer. From the Model Hierarchy tree, select **ex_func_SF > Chart > f()add_function**. On the right pane, specify the **Function Inline Option** as **Function**.
- 7 From the Model Hierarchy tree, click **Chart** and on the right pane select the **Export Chart Level Functions (Make Global)** parameter. This makes the function available globally to the entire model.
- 8 Press **Ctrl+B** to build the model and generate code.

Results

`ex_func_SF.c` contains the generated code:

```
extern real_T add_function(real_T sf_in1, real_T sf_in2)
{
    return sf_in1 + sf_in2;
}
.
.
.
void ex_func_SF_step(void)
{
    ex_func_SF_B.y1 = add_function(u1, u2);
    ex_func_SF_Y.y1 = ex_func_SF_B.y1;
}
```

Control Function Prototype of the *model_step* Function



ex_control_step_function

Procedure

- 1 Create the model, `ex_control_step_function`. See “Configure a Signal” on page 4-4 and “Configure Input and Output Ports” on page 4-4, for more information.
- 2 Press **Ctrl+E** to open the Configuration Parameters dialog box.
- 3 On the **Code Generation > Interface** pane, click **Configure Model Functions** to open the Model Interface dialog box.
- 4 Specify the **Function specification** parameter as `Model specific C prototypes`.
- 5 Click **Get Default Configuration** to update the **Configure model initialize and step functions** section and list the input and output arguments.
- 6 To configure the function output argument to pass a pointer, in the **Step function arguments** table, specify the **Category** for the Outport as a **Pointer**. In addition, you can specify the step function arguments order and type qualifiers.
- 7 To validate your changes, click **Validate**.
- 8 Press **Ctrl+B** to build the model and generate code.

Results

`ex_control_step_function.c` contains the generated code:

```

void ex_control_step_function_custom(real_T arg_u1, real_T arg_u2, ...
                                   real_T *arg_y1)
{
    (*arg_y1) = arg_u1 + arg_u2;
}
  
```

External C Functions

C Construct

```
extern double add(double, double);

#include "add.h"
double add(double u1, double u2)
{
    double y1;
    y1 = u1 + u2;
    return (y1);
}
```

Modeling Patterns

There are several methods for integrating legacy C functions into the generated code. These methods either create an S-function or make a call to an external C function. For more information on S-functions, see “Insert S-Function Code”.

- “Use the Legacy Code Tool to Create S-functions” on page 4-59
- “Use a Stateflow Chart to Make Calls to C Functions” on page 4-61
- “Using a MATLAB Function Block to Make Calls to C Functions” on page 4-62

Use the Legacy Code Tool to Create S-functions

This method uses the Legacy Code Tool to create an S-function and generate a TLC file. The code generation software uses the TLC file to generate code from this S-function. The advantage of using the Legacy Code Tool is that the generated code is fully inlined and does not need wrapper functions to access the custom code.

Procedure

- 1 Create a C header file named `add.h` that contains the function signature:

```
extern double add(double, double);
```
- 2 Create a C source file named `add.c` that contains the function body:

```
double add(double u1, double u2)
```

```
{
    double y1;
    y1 = u1 + u2;
    return (y1);
}
```

- 3** To build an S-function for use in both simulation and code generation, Run the following script or execute each of these commands at the MATLAB command line:

```
%% Initialize legacy code tool data structure
def = legacy_code('initialize');

%% Specify Source File
def.SourceFiles = {'add.c'};

%% Specify Header File
def.HeaderFiles = {'add.h'};

%% Specify the Name of the generated S-function
def.SFunctionName = 'add_function';

%% Create a c-mex file for S-function
legacy_code('sfcn_cmex_generate', def);

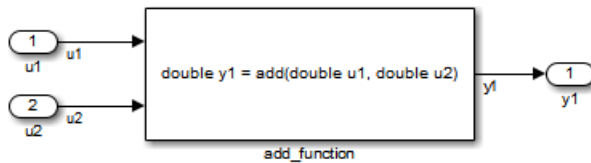
%% Define function signature and target the Output method
def.OutputFcnSpec = ['double y1 = add(double u1, double u2)'];

%% Compile/Mex and generate a block that can be used in simulation
legacy_code('generate_for_sim', def);

%% Create a TLC file for Code Generation
legacy_code('sfcn_tlc_generate', def);

%% Create a Masked S-function Block
legacy_code('slblock_generate', def);
The output of this script produces:
```

- A new model containing the S-function block
 - A TLC file named `add_function.tlc`.
 - A C source file named `add_function.c`.
 - A mexw32 dll file named `add_function.mexw32`
- 4** Add inport blocks and an outport block and make the connections, as shown in the model.



ex_function_call_lct

- 5 Name and save your model. In this example, the model is named `ex_function_call_lct`.
- 6 Press **Ctrl+B** to build the model and generate code.

Results

The following code is generated in `ex_function_call_lct.c`:

```
real_T u1;
real_T u2;
real_T y1;
void ex_function_call_lct_step(void)
{
    y1 = add(u1, u2);
}
```

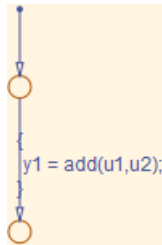
The user-specified header file, `add.h`, is included in `ex_function_call_lct.h`:

```
#include "add.h"
```

Use a Stateflow Chart to Make Calls to C Functions

Procedure

- 1 Create a C header file named `add.h` that contains the example function signature.
- 2 Create a C source file named `add.c` that contains the function body.
- 3 Follow the steps for “Set Up an Example Model With a Stateflow Chart” on page 4-6. This example model contains two Inport blocks and one Outport block.
- 4 Name the example model `ex_exfunction_call_SF`.
- 5 Double-click the Stateflow chart and edit the chart as shown. Place the call to the `add` function within a transition action.



ex_exfunction_call_SF/Chart

- 6 On the **Stateflow Editor**, select **Simulation > Model Configuration Parameters**.
- 7 On the Configuration Parameters dialog box, select **Simulation Target > Custom Code**. In the **Include custom C code in generated** section, on the left pane, select **Header file** and in the **Header file** field, enter the `#include` statement:

```
#include "add.h"
```
- 8 In the **Include list of additional** section, select **Source files** and in the **Source files** field, enter `add.c`.
- 9 Press **Ctrl+B** to build the model and generate code.

Results

ex_exfunction_call_SF.c contains the following code in the step function:

```
real_T u1;  
real_T u2;  
real_T y1;  
  
void exfunction_call_SF_step(void)  
{  
    y1 = (real_T)add(u1, u2);  
}
```

ex_exfunction_call_SF.h contains the include statement for `add.h`:

```
#include "add.h"
```

Using a MATLAB Function Block to Make Calls to C Functions

Procedure

- 1 Create a C header file named `add.h` that contains the example function signature.

- 2 Create a C source file named `add.c` that contains the function body.
- 3 In the Simulink Library Browser, click **Simulink > User Defined Functions**, and drag a MATLAB Function block into your model.
- 4 Double-click the MATLAB Function block. The MATLAB Function Block Editor opens.
- 5 Edit the function to include the statement:

```
function y1 = add_function(u1, u2)

%Set the class and size of output
y1 = u1;

%Call external C function
y1 = coder.ceval('add',u1,u2);

end
```

- 6 Open the Configuration Parameters dialog box, and select **Simulation Target > Custom Code**.
- 7 In the **Include custom C code in generated** section, on the left pane, select **Header file** and in the **Header file** field, enter the statement, :


```
#include "add.h"
```
- 8 In the **Include list of additional** section, select **Source files** and in the **Source files** field, enter `add.c`.
- 9 Add two Inport blocks and one Outport block to the model and connect to the MATLAB Function block.
- 10 Configure the signals: `u1`, `u2`, and `y1`, as described in “Configure a Signal” on page 4-4.
- 11 Save the model as `ex_exfunction_call_ML`.
- 12 Press **Ctrl+B** to build the model and generate code.

Results

`ex_exfunction_call_ML.c` contains the following code:

```
real_T u1;
real_T u2;
real_T y1;

void ex_exfunction_call_ML_step(void)
{
```

```
    y1 = add(u1, u2);  
}
```

`ex_exfunction_call_ML.h` contains the `#include` statement for `add.h`:

```
#include "add.h"
```


Macro Definitions (#define)

C Construct

```
#define p_1 9.8;
```

Modeling Patterns

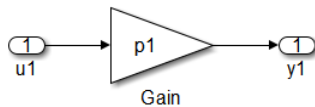
“Use a 'Define' Custom Storage Class” on page 4-65

“Use a Custom Header File” on page 4-66

Use a 'Define' Custom Storage Class

Procedure

- 1 Create a model containing a Gain block.



- 2 Press **Ctrl+E** to open the Configuration Parameters dialog box.
- 3 In the Configuration Parameter dialog box, on the **Optimization > Signals and Parameters** pane, select **Inline parameters**.
- 4 Click **Apply** and **OK**.
- 5 In your model, double-click the Gain block. The Block Parameters dialog box opens.
- 6 In the **Value** field, enter a variable name. In this example, the variable name is **p1**.
- 7 Press **Ctrl+H** to open the Model Explorer. On the Model Hierarchy pane, select the Base Workspace.
- 8 To add an **mpt** parameter object, in the menu bar, select **Add > Add Custom**. The parameter appears in the **Contents of: Base Workspace** pane.
- 9 Double-click the **mpt.Parameter** object and change its name to **p1**.
- 10 Click the **p1** parameter. The data object parameters are displayed in the right pane of the Model Explorer.

- 11 In the **Value** field, enter **9.8**. In the **Code generation options** section, click the **Storage Class** drop-down list and select **Define (Custom)**.
- 12 Press **Ctrl+B** to generate code.

Results

The generated code includes the inlined parameter, `p1`, in `ex_define_data_object.c`:

```
/* Model step function */
void ex_define_data_object_step(void)
{
    rtY.y1 = p1 * rtU.u1;
}
```

Use a Custom Header File

Procedure

- 1 Follow steps 1 through 10 of “Use a 'Define' Custom Storage Class” on page 4-65.
- 2 In the Simulink.Parameter dialog box for `p1`, in the **Value** field, enter **9.8**. In the **Code generation options** section, click the **Storage Class** drop-down list and select **ImportFromFile (Custom)**.
- 3 In the **Header file** parameter, enter the name of the header file, in this example, `external_params.h`.
- 4 Click **Apply** and **OK**.
- 5 Create the C header file, `external_params.h` that contains the `#define` statement:

```
#ifndef _EXTERNAL_PARAMS
#define _EXTERNAL_PARAMS

#define p1 9.8

#endif

/* EOF */
```

- 6 Press **Ctrl+B** to generate code.

Results

The generated code includes the inlined parameter, `p1`, in `ex_define_data_object.c`:

```
/* Model step function */
void ex_define_data_object_step(void)
{
```

```
ex_define_data_object_Y.Out1 = p1 * ex_define_data_object_U.In1;  
}
```

Conditional Inclusions (`#if / #endif`)

You can generate preprocessor conditional directives in your code by implementing variant blocks (Model Variants block or Variant Subsystem block) in your model. In the generated code, preprocessor conditional directives select a section of code to execute at compile time. To implement variants in your model, see “Workflow for Implementing Variants”. To generate code for variants, see “Generate Preprocessor Conditionals for Variant Systems”.

Typedef

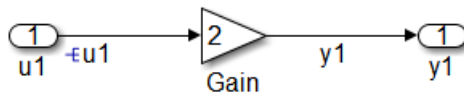
To generate a typedef definition, use a `Simulink.AliasType` data object.

C Construct

```
typedef double float_64;
```

Procedure

- 1 Create the `ex_get_typedef` model with a Gain block.



- 2 In the Gain block parameter dialog box, select the **Parameter Attributes** tab, and specify the **Parameter data type** as `double`.
- 3 Right-click the `u1` signal and select **Properties**. In the Signal Properties dialog box, select **Signal name must resolve to Simulink signal object**.
- 4 Right-click the `y1` signal and select **Properties**. In the Signal Properties dialog box, select the **Code Generation** tab, and specify the **Storage class** parameter as `ExportedGlobal`.
- 5 Create a new alias type by using a “`Simulink.AliasType`” data object. At the MATLAB command line, enter:


```
float_64 = Simulink.AliasType;
```
- 6 In the base workspace, double-click `float_64`. The `Simulink.AliasType` dialog box opens.
- 7 Specify the **Base type** parameter as `double`. Click **Apply** and **OK**.
- 8 Create a data object for the `u1` signal. In the base workspace, select **Add > Simulink Signal**, and name it `u1`. Specify the **Data type** parameter as `float_64` and the **Storage class** parameter as `Global (custom)`.

Note: You can also specify an output data type for Simulink blocks using the new alias type.

- 9 Click **Apply** and **OK**.

10 Press **Ctrl+B** to generate code.

Note: An alternative method for defining a **typedef** is to import the alias type from a custom header file. If you want to import all of the **typedefs** from a C header file, using this alternative method is useful.

Results

The generated code includes the **typedef** definition, which is declared within **#ifndef** and **#endif** statements in the **ex_get_typedef_types.h** file.

```
#ifndef _DEFINED_TYPEDEF_FOR_float_64_
#define _DEFINED_TYPEDEF_FOR_float_64_

typedef real_T float_64;
typedef creal_T cfloat_64;

#endif
```

Note: **real_T** is the Embedded Coder **typedef** for **double** .

The generated code also includes the declaration of the Simulink data objects of the alias type in **ex_get_typedef.c**.

```
float_64 y1;
float_64 u1;
```

Structures for Parameters

To generate a structure containing parameters, use a `mpt.Parameter` object with a `Struct (custom)` storage class.

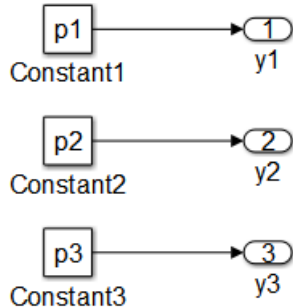
C Construct

```
typedef struct {
    double p1;
    double p2;
    double p3;
} my_struct_type;

my_struct_type my_struct={1.0,2.0,3.0};
```

Procedure

- 1 Create the `ex_struct_param` model with three Constant blocks and three Outport blocks.



- 2 Create a data object for each parameter, `p1`, `p2`, and `p3`. At the MATLAB command line, enter:

```
p1 = mpt.Parameter;
p2 = mpt.Parameter;
p3 = mpt.Parameter;
```

- 3 In the base workspace, double-click one of the parameter data objects to open the `mpt.Parameter` dialog box.
- 4 Specify a **Value** parameter for each parameter object.
- 5 Specify the **Storage class** parameter as `Struct (Custom)` for each parameter object.

- 6 In the **Custom Attributes** section, specify the **Struct name** as `my_struct`. Click **Apply** and **OK**.
- 7 Press **Ctrl+E** to open the Configuration Parameters dialog box.
- 8 Open the **Optimization > Signals and Parameters** pane, and select the **Inline parameters** parameter.
- 9 Click **Apply** and **OK**.
- 10 Press **Ctrl+B** to generate code.

Results

The generated code includes the `typedef` definition for a structure, which is declared in the `ex_struct_param_types.h` file.

```
/* Type definition for custom storage class: Struct */
typedef struct my_struct_tag {
    real_T p1;
    real_T p2;
    real_T p3;
} my_struct_type;
```

The generated code also includes the declaration of `my_struct` in `ex_struct_param.c`.

```
/* Definition for custom storage class: Struct */
my_struct_type my_struct = {
    /* p1 */
    1.0,

    /* p2 */
    2.0,

    /* p3 */
    3.0
};
```


Structures for Signals

To generate a structure containing parameters, use a `mpt.Signal` object with a `Struct` (custom) storage class or a Simulink non-virtual bus object.

C Construct

```
typedef struct {
    double u1;
    double u2;
    double u3;
} my_signals;
```

Modeling Patterns

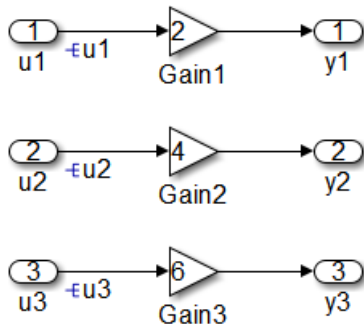
“Structure for Signals Using a 'Struct' Custom Storage Class” on page 4-73

“Structure for Signals Using a Simulink Non-Virtual Bus Object” on page 4-74

Structure for Signals Using a 'Struct' Custom Storage Class

Procedure

- 1 Create the `ex_signal_struct_csc` model using the blocks shown and follow the steps to configure the signals and model.



- 2 Double-click a Gain block to open the block parameter dialog box. Set the values of the Gain blocks as shown in the model diagram.
- 3 Right-click the `u1` signal and select **Properties**. In the Signal Properties dialog box, select **Signal name must resolve to Simulink signal object**. Repeat for signals `u2` and `u3`.

- 4 At the MATLAB command line, create a `mpt.Signal` data object for each input signal.

```
u1 = mpt.Signal;  
u2 = mpt.Signal;  
u3 = mpt.Signal;
```

Note: You can also create a data object in the Model Explorer base workspace, by selecting **Add > Add Custom**.

- 5 In the base workspace, configure each of the data objects, `u1`, `u2`, and `u3`. Double-click a data object, to open the `mpt.Signal` parameter dialog box.
- 6 Specify the **Data type** parameter as `auto` and the **Storage class** parameter as `Struct (custom)`.
- 7 Click **Apply** and **OK**.
- 8 Press **Ctrl+B** to generate code.

Results

The generated code includes the `typedef` definition for a structure, which is declared in the `ex_signal_struct_csc_types.h` file.

```
/* Type definition for custom storage class: Struct */  
typedef struct my_signal_struct_tag {  
    real_T u1;  
    real_T u2;  
    real_T u3;  
} my_signal_struct_type;
```

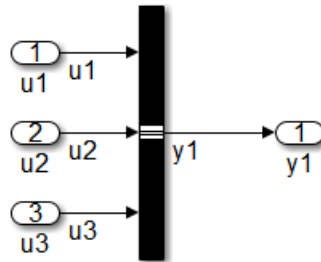
The generated code also includes the declaration of `my_signal_struct` in `ex_signal_struct_csc.c`.

```
/* Definition for custom storage class: Struct */  
my_signal_struct_type my_signals;
```

Structure for Signals Using a Simulink Non-Virtual Bus Object

Procedure

- 1 Create the `ex_signal_struct_bus` model using the blocks shown and follow the steps to configure the bus object and model.



This block creates a bus signal from its inputs.

- 2 Add the Inport blocks, an Output block, and a Bus Creator block to your diagram.
- 3 Double-click the Bus Creator block to open the block parameter dialog box.
- 4 Specify the **Number of inputs** parameter as **3**. Click **Apply**.
- 5 In your model diagram, connect the three Inport blocks to the three inports of the Bus Creator block. Also, connect the output of the Bus Creator block to the Output block.
- 6 Label the signals as shown in the model diagram.
- 7 In the Bus Creator block parameter dialog box, **Signals in bus** now displays the signals connected to the Bus Creator block.
- 8 Create a bus object named **MySignals** that includes signals **u1**, **u2**, and **u3**. For more information on creating bus objects, see “Manage Bus Objects with the Bus Editor”. Once the bus object, **MySignals**, is created, it appears in the base workspace.
- 9 In the Bus Creator block parameter dialog box, select the **Output as nonvirtual bus** parameter, which specifies that bus signals must be grouped into a structure in the generated code.
- 10 Click **Apply** and **OK**.
- 11 Press **Ctrl+B** to generate code.

Results

The generated code includes the `typedef` definition for a structure, which is declared in the `signal_struct_bus_types.h` file.

```
typedef struct {
    real_T u1;
    real_T u2;
    real_T u3;
} MySignals;
```

Nested Structures

One way to create nested structures of signals in the generated code is by using multiple non-virtual bus objects. When nesting bus objects, all of the bus objects must either be non-virtual, or all of them must be virtual.

C Construct

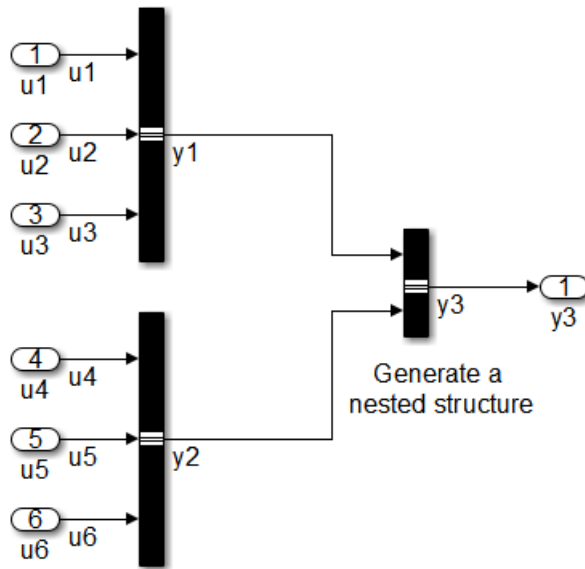
```
typedef struct {
    double u1;
    double u2;
    double u3;
} my_signals123;

typedef struct {
    double u4;
    double u5;
    double u6;
} my_signals456;

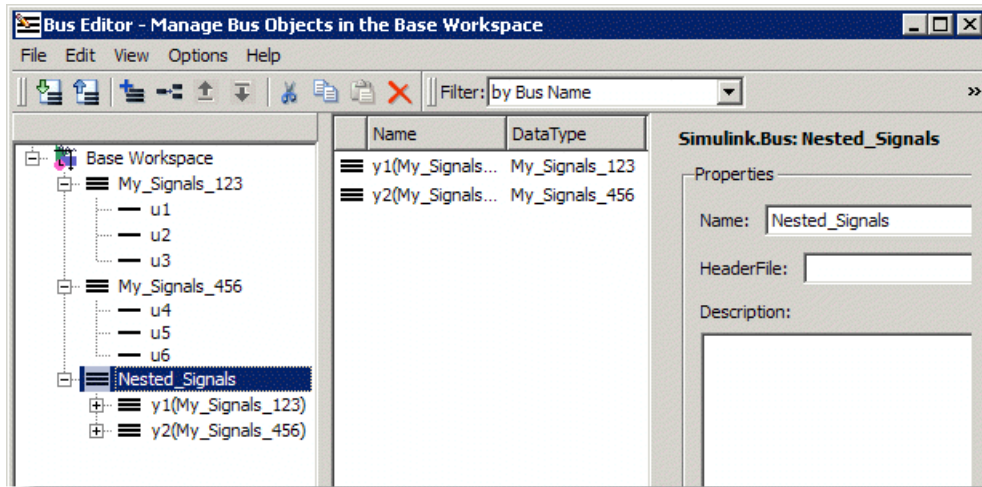
typedef struct {
    my_signals123 y1;
    my_signals456 y2;
} nested_signals;
```

Procedure

- 1 Create the `ex_nested_structure` model using the blocks shown and follow the steps to configure the bus objects and model.



- 2 For each bus in the model, follow the instructions for “Structure for Signals Using a Simulink Non-Virtual Bus Object” on page 4-74, creating bus objects My_Signals_123 and My_Signals_456.
- 3 Drag a Bus Creator block into your model. Configure the Bus Creator block so that it takes in signals from different buses.
- 4 Double-click the Bus Creator block to open the block parameter dialog box.
- 5 Specify the **Number of inputs** parameter as 2. Click **Apply**.
- 6 In your model diagram, connect the two bus outputs to the inports of the new Bus Creator block.
- 7 Label the signals as shown in the model diagram.
- 8 In the Bus Creator block parameter dialog box, **Signals in bus** now displays the signals, y1 and y2, connected to the Bus Creator block.
- 9 Create a bus object named Nested_Signals that includes signals y1 and y2, where the **Data Type** for y1 is My_Signals_123 and the **Data Type** for y2 is My_Signals_456.



For more information on creating bus objects, see “Manage Bus Objects with the Bus Editor”. Once the bus object, `Nested_Signals`, is created, it appears in the base workspace.

- 10 In the Bus Creator block parameter dialog box, select the **Output as nonvirtual bus** parameter, which specifies that bus signals must be grouped into a structure in the generated code.
- 11 Click **Apply** and **OK**.
- 12 Press **Ctrl+B** to generate code.

Results

The generated code includes the `typedef` definitions for structures, which are declared in the `ex_nested_structure_types.h` file.

```
#ifndef _DEFINED_TYPEDEF_FOR_My_Signals_123_
#define _DEFINED_TYPEDEF_FOR_My_Signals_123_

typedef struct {
    real_T u1;
    real_T u2;
    real_T u3;
} My_Signals_123;

#endif
```

```
#ifndef _DEFINED_TYPEDEF_FOR_My_Signals_456_  
#define _DEFINED_TYPEDEF_FOR_My_Signals_456_  
  
typedef struct {  
    real_T u4;  
    real_T u5;  
    real_T u6;  
} My_Signals_456;  
  
#endif  
  
#ifndef _DEFINED_TYPEDEF_FOR_Nested_Signals_  
#define _DEFINED_TYPEDEF_FOR_Nested_Signals_  
  
typedef struct {  
    My_Signals_123 y1;  
    My_Signals_456 y2;  
} Nested_Signals;  
  
#endif
```

Bitfields

One way to create bitfields in the generated code is by using a `mpt.Parameter` object with `Bitfield (Custom)` storage class.

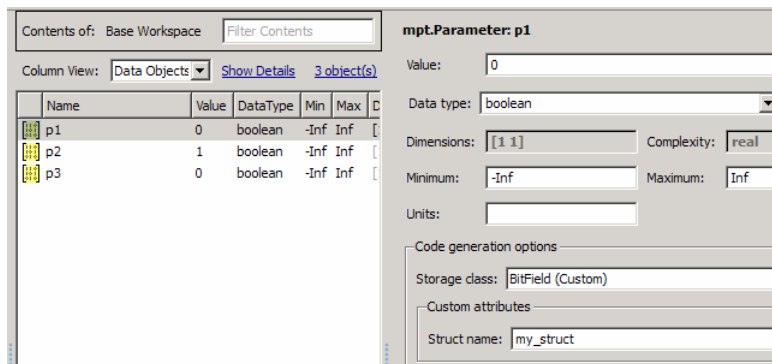
C Construct

```
typedef struct {  
    unsigned int p1 : 1;  
    unsigned int p2 : 1;  
    unsigned int p3 : 1;  
} my_struct_type
```

Procedure

- 1 Using the model, `ex_struct_param`, in “Structures for Parameters” on page 4-71, rename the model as `ex_struct_bitfield_GSC`.
- 2 Create a data object for each parameter, `p1`, `p2`, and `p3`. At the MATLAB command line, enter:

```
p1 = mpt.Parameter;  
p2 = mpt.Parameter;  
p3 = mpt.Parameter;
```
- 3 In the base workspace, double-click one of the parameter data objects to open the `mpt.Parameter` dialog box.
- 4 Specify the **Value** parameter for each parameter object.
- 5 Specify the **Storage class** parameter as `Bitfield (Custom)` for each parameter object.
- 6 In the **Custom Attributes** section, specify the **Struct name** as `my_struct`. Click **Apply** and **OK**.
- 7 Specify the data objects for each parameter.



- 8 Press **Ctrl+E** to open the Configuration Parameters dialog box.
- 9 Open the **Optimization > Signals and Parameters** pane, and select the **Inline parameters** parameter.
- 10 Click **Apply** and **OK**.
- 11 Press **Ctrl+B** to generate code.

Results

The generated code of the model, `ex_struct_bitfield_CSC`, includes the typedef definition for a Bitfield, which is declared in the `ex_struct_bitfield_CSC_types.h` file.

```
/* Type definition for custom storage class: BitField */
typedef struct my_struct_tag {
    uint_T p1 : 1;
    uint_T p2 : 1;
    uint_T p3 : 1;
} my_struct_type;
```

Arrays for Parameters

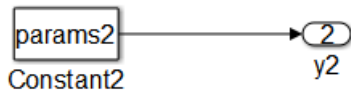
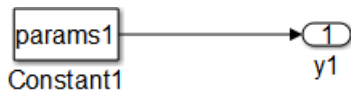
To create an array in the generated code, you can use a constant parameter in the base workspace, or a `mpt.Parameter`.

C Construct

```
int params[5]= {1,2,3,4,5};
```

Procedure

- 1 Create a model, `ex_array_params`, containing the Constant blocks and Output blocks and label the blocks as shown in the model diagram.



- 2 Double-click the Constant1 block and give the **Constant value** the name of a parameter, `params1`.
- 3 Double-click the Constant2 block and give the **Constant value** the name of a parameter, `params2`.
- 4 To create the parameters in the base workspace, at the MATLAB command line, enter:


```
params1 = [1,2,3,4,5];
params2 = mpt.Parameter;
```
- 5 In the base workspace, double-click `params2` to open the `mpt.Parameter` dialog box.
- 6 In the **Value** field, specify the array, `[1 2 3 4 5]`.
- 7 Press **Ctrl+E** to open the Configuration Parameters dialog box.
- 8 Open the **Optimization > Signals and Parameters** pane, and select the **Inline parameters** parameter.
- 9 Click **Apply** and **OK**.

10 Press **Ctrl+B** to generate code.

Results

The generated code includes the array, `params2`, in the `ex_array_params.c` file:

```
int16_T params2[5] = { 1, 2, 3, 4, 5 } ;
```

The data object, `params1`, is defined in the `array_params_data.c` file:

```
/* Constant parameters (auto storage) */
const ConstParam_array_params array_params_ConstP = {
  /* Computed Parameter: Constant1_Value
   * Referenced by: '/Constant1'
   */
  { 1, 2, 3, 4, 5 }
};
```

where `ConstParam_array_params` is a structure containing the array and defined in the `array_params.h` file.

```
typedef struct {
  /* Computed Parameter: Constant1_Value
   * Referenced by: '/Constant1'
   */
  int16_T Constant1_Value[5];
} ConstParam_array_params;
```

Arrays for Signals

To create an array in the generated code for signal data, you can specify a signal as `ExportedGlobal`, or use a `mpt.Signal` object.

C Construct

```
int u1[5];
int y1[5];
```

Procedure

- 1 Create the `ex_array_signals` model using the blocks shown and follow the steps to configure the signals and model.



- 2 Double-click the Inport block to open the Inport block parameter dialog box.
- 3 Select the **Signal Attributes** tab and specify the **Port dimensions** parameter as 5, for an array of length 5.
- 4 Click **OK**.
- 5 Right-click the `u1` signal line and select **Properties**.
- 6 Select the **Code Generation** tab and specify the **Storage Class** parameter as `ExportedGlobal`.
- 7 Repeat steps 5 and 6 for signal `y1`.
- 8 Press **Ctrl+B** to generate code.

Note: Alternatively, you can use Simulink data objects (`mpt.Signal`) to specify the storage class and dimensions for the signals, `u1` and `y1`.

Results

The generated code includes arrays for `u1` and `y1` in the `ex_array_signals.c` file:

```
int16_T u1[5];
int16_T y1[5];
```

In this case, a `for` loop is generated to carry out the gain operations on elements of the input signal.

```
int32_T i;
for (i = 0; i < 5; i++) {
    y1[i] = (int16_T)(5 * u1[i]);
}
```

However, if the dimension of the array is less than a threshold value (typically 5), code generation might not include a `for` loop for array operations.

Pointers for Signals

To create a pointer in the generated code, you can configure a signal to use the `ImportedExternPointer` storage class or use an `mpt.Signal` (or `mpt.Parameter` for parameters) object with an `ImportedExternPointer` storage class.

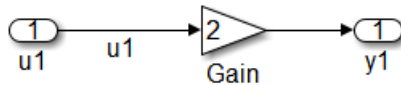
C Construct

```
extern double *u1;
```

Procedure

This is a quick method to obtain pointers in the generated code. You cannot control the data type, which is decided by the model compilation process.

- 1 Create the `ex_pointer_signal` model using the blocks shown and follow the steps to configure the signals and model.



- 2 Label the signal to be imported as a pointer, in this example, `u1`.
- 3 Right-click the `u1` signal line and select **Properties**.
- 4 Select the **Code Generation** tab and specify the **Storage Class** parameter as `ImportedExternPointer`.
- 5 Click **OK**.
- 6 Press **Ctrl+B** to generate code.

Results

The generated code includes the `extern` declaration for the pointer in the `ex_pointer_signal_private.h` file.

```
extern real_T *u1;
```

Pointers Using Simulink Data Objects

You can control the data type of a signal or parameter by using a Simulink data object to generate a pointer.

C Construct

```
extern double *u1;
```

Procedure

You can use this procedure for either a signal or parameter. To create a pointer for a parameter, use an `mpt.Parameter` instead of an `mpt.Signal` data object described in step 3.

- 1 Create the `ex_pointer_signal_data_object` model using the blocks shown and follow the steps to configure the signals and model.



- 2 Label the signal to be imported as a pointer, in this example, `u1`.
- 3 At the MATLAB command line, create a data object for signal `u1`.


```
u1 = mpt.Signal;
```
- 4 In the base workspace, double-click `u1` to open the `mpt.Signal` dialog box.
- 5 Specify the **Storage class** parameter as `ImportedExternPointer`.
- 6 Click **Apply** and **OK**.
- 7 Press **Ctrl+B** to generate code.

Results

The generated code includes the `extern` declaration for the pointer in the `ex_pointer_signal_data_object_private.h` file.

```
extern real_T *u1;
```

The `ex_pointer_signal_data_object_private.h` file imports the pointer into the generated code. To compile the code, you must declare and define the pointer in the main program.

Variant Systems

- “About Variant Systems” on page 5-2
- “Why Generate Code for Variant Systems?” on page 5-3
- “Generate Preprocessor Conditionals for Variant Systems” on page 5-4
- “Review Code Variants in Code Generation Report” on page 5-7
- “Generate Code for Model Variants” on page 5-8
- “Generate Code for Variant Subsystems” on page 5-10
- “Restrictions on Variant Subsystem Code Generation” on page 5-15
- “Special Considerations for Generating Preprocessor Conditionals” on page 5-17
- “Limitations on Generating Code for Variants” on page 5-18
- “Generated Code Components Not Compiled Conditionally” on page 5-19

About Variant Systems

Embedded Coder generates code from a Simulink model containing one or more Variant Subsystem blocks. To learn how to create a model containing variant blocks, see “Workflow for Implementing Variants”.

Code is generated for different variant choices, the active variant, and the default variant. To generate code for variants, set the following conditions in the Variant Subsystem block:

- Deselect **Override variant conditions and use the following variant**.
- Select **Generate preprocessor conditionals**.

Code generated for variants is surrounded by C preprocessor conditionals `#if`, `#else`, `#elif`, and `#endif`. Therefore, the active variant is selected at compile time and the preprocessor conditionals determine which sections of the code to execute.

To construct model reference variants and generate preprocessor directives in the generated code, see the example `rtwdemo_preprocessor_script`.

To construct variant subsystems and generate preprocessor directives in the generated code, see the example `rtwdemo_preprocessor_subsys_script`.

Why Generate Code for Variant Systems?

When you implement variants in the generated code, you can:

- Reuse generated code from a set of application models that share functionality with minor variations.
- Share generated code with a third party that activates one of the variants in the code.
- Validate the supported variants for a model and then choose to activate one variant for a particular application, without regenerating and re-validate the code.
- Generate code for the default variant that is selected when an active variant does not exist.

Generate Preprocessor Conditionals for Variant Systems

Define Variant Controls

To learn about variant controls, see “Create, Export, and Reuse Variant Controls” in the Simulink documentation. Perform the following steps to define variant controls for generating code.

- 1 Open the Model Explorer and select the **Base Workspace** node.
- 2 A variant control can be a condition expression, a “Simulink.Variant class” object specifying a condition expression or a `Simulink.Parameter` object. In the Model Explorer, select **Add > Simulink Parameter**. Specify a name for the new parameter.
- 3 On the `Simulink.Parameter` property dialog box, specify the **Value** and **Data type**.
- 4 Select one of these **Storage class** values.
 - `ImportedDefine (Custom)`
 - `CompilerFlag (Custom)`
 - A storage class created using the Custom Storage Class Designer. Your storage class must have the **Data initialization** parameter set to `Macro` and the **Data scope** parameter set to `Imported`. See “Use Custom Storage Class Designer” for more information.
- 5 Specify the value of the variant control. If the storage class is either `ImportedDefine (Custom)` or a custom storage class, do the following:
 - a Specify the **Header File** parameter as an external header file in the Custom Attributes section of the `Simulink.Parameter` property dialog box.
 - b Enter the values of the variant controls in the external header file.

Note: The generated code refers to a variant control as a user-defined macro. The generated code does not contain the value of the macro. The value of the variant control determines the active variant in the compiled code.

If the variant control is a `CompilerFlag` custom storage class the value of the variant control is set at compile time. On the **Code Generation > General** pane of the **Model Configuration Parameters** dialog box, add a makefile option to

the “Make command” parameter. For example, for variant control, VSSMODE, enter `make_rtw OPTS="-DVSSMODE=1"` in the **Make command** field.

Note: If you want to modify the value of the variant control after generating the makefile, use a makefile option when compiling your code. For example, at a command line outside of MATLAB, enter:

```
makecommand -f model.mk OPTS="-DVSSMODE=1"
```

- 6 Follow the instructions in “Configure Model for Generating Preprocessor Conditional Directives” on page 5-5 to implement variant objects for code generation. Check that only one variant object is active in the generated code by implementing the condition expressions of the variant objects such that only one evaluates to `true`. The generated code includes a test of the variant objects to determine that there is only one active variant. If this test fails, your code will not compile.

Note: You can define the variant controls using `Simulink.Parameter` object of enumerated type. This approach provides meaningful names and improves the readability of the conditions. The generated code includes preprocessor conditionals to check that the variant condition contains valid values of the enumerated type.

Configure Model for Generating Preprocessor Conditional Directives

- 1 Open the Configuration Parameter dialog box.
- 2 Select the **Code Generation** pane, and set **System target file** as `ert.tlc`.
- 3 In the **Report** pane, select **Create code generation report**.
- 4 Select the **Code Generation** pane, and clear “Ignore custom storage classes”. In order to generate preprocessor conditionals, you must use custom storage classes.
- 5 Select the **Interface** pane, and select the **Use Local Settings** option of the **Generate preprocessor conditionals** parameter. This parameter is a global setting for the parent model. This setting enables the **Generate preprocessor conditionals** parameter located in the Model Variants block parameters dialog box or Variant Subsystem parameters dialog box. See “Generate preprocessor conditionals” for more information.
- 6 Open the Model Variants block parameters dialog box or the Variant Subsystem block parameter dialog box, depending on your application. Select the **Generate preprocessor conditionals** parameter. If the block parameters dialog box

was already open, close and reopen the dialog box to see the enabled **Generate preprocessor conditionals** parameter.

- 7 Clear the parameter, **Override variant conditions and use following variant**.

Build Your Model

After configuring your model to generate code, build your model.

Review Code Variants in Code Generation Report

The Code Variants Report displays a list of the variant objects in alphabetical order and their condition. The report also lists the model blocks that have Variants, and the referenced models that use them. In the **Contents** section of the code generation report, click the link to the Code Variants Report:

Code Variants Report for rtwdemo_preprocessor

Table of Contents

- [Variant Control](#)
- [Model Reference Blocks that have Variants](#)
- [Subsystem Blocks that have Variants](#)

Variant Control [\[hide\]](#)

Variant	Condition	Used in Blocks
LINEAR	VSSMODE == 0	<Root>/Left Controller <Root>/Right Controller
NONLINEAR	VSSMODE == 1	<Root>/Left Controller <Root>/Right Controller

Model Reference Blocks that have Variants [\[hide\]](#)

Model Block	Variant	Model
<Root>/Left Controller	LINEAR	rtwdemo_linl
	NONLINEAR	rtwdemo_nlinl
<Root>/Right Controller	LINEAR	rtwdemo_linr
	NONLINEAR	rtwdemo_nlinr

Subsystem Blocks that have Variants [\[hide\]](#)

(No SubSystem blocks that have Variants)

Generate Code for Model Variants

To open a model for generating preprocessor conditionals, enter `rtwdemo_preprocessor`.

After building the model, look at the variants in the generated code.

`rtwdemo_preprocessor_types.h` includes the following:

- Call to external header file, `rtwdemo_preprocessor_macros.h`, which contains the macro definition for the variant control variable, `VSSMODE`.

```
/* Includes for objects with custom storage classes. */
#include "rtwdemo_importedmacros.h"
```

- Preprocessor directives defining the variant objects, `LINEAR` and `NONLINEAR`. The values of these macros depend on the value of the variant control variable, `VSSMODE`. The condition expression associated with each macro, `LINEAR` and `NONLINEAR`, determine the active variant.

```
/* Model Code Variants */
#ifndef LINEAR
#define LINEAR (VSSMODE == 0)
#endif

#ifndef NONLINEAR
#define NONLINEAR (VSSMODE == 1)
#endif
```

- Check that exactly one variant is active at a time:

```
/* Exactly one variant for '<Root>/Left Controller' should be active */
#if (LINEAR) + (NONLINEAR) != 1
#error Exactly one variant for '<Root>/Left Controller' should be active
#endif
```

Calls to the step and initialization functions are conditionally compiled as shown in a portion of the step function, `rtwdemo_preprocessor_step`:

```
#if LINEAR

/* ModelReference: '<Root>/Left Controller' */
rtwdemo_linl(&rtb_Add, &rtb_LeftController_vmerge_1,
            &(rtwdemo_preprocessor_DWork.LeftController_1_DWORK1.rtdw));

#elif NONLINEAR

/* ModelReference: '<Root>/Left Controller' */
rtwdemo_nlinl(&rtb_Add, &rtb_LeftController_vmerge_1,
            &(rtwdemo_preprocessor_DWork.LeftController_2_DWORK1.rtdw));

#endif
and
#if LINEAR

/* ModelReference: '<Root>/Right Controller' */
```



```
    rtwdemo_linr(&rtb_Add, &rtb_RightController_vmerge,
                &(rtwdemo_preprocessor_DWork.RightController_1_DWORK1.rtdw));
#elif NONLINEAR
    /* ModelReference: '<Root>/Right Controller' */
    rtwdemo_nlinr(&rtb_Add, &rtb_RightController_vmerge_1,
                &(rtwdemo_preprocessor_DWork.RightController_2_DWORK1.rtdw));
#endif                                     /* LINEAR */
```

Generate Code for Variant Subsystems

In this section...
“Open Example Model” on page 5-10
“Define Variant Controls” on page 5-10
“Configure Model for Generating Preprocessor Conditional Directives” on page 5-12
“View Generated Code” on page 5-13

Open Example Model

Open model `rtwdemo_preprocessor_subsys`, which contains a variant subsystem.

Define Variant Controls

Variant controls can be a condition expression or `Simulink.Variant` object specifying a condition expression or a default variant. Condition expressions specified directly or used in `Simulink.Variant` objects should reference `Simulink.Parameter` objects.

- 1 Open the Model Explorer and click the **Base Workspace**.
- 2 Select **Add > Simulink Parameter** to create the variant control variable, `VSSMODE`.
- 3 In the `Simulink.Parameter` property dialog box for `VSSMODE`, specify the **Value** as 1 and the **Data type** as `int32`.
- 4 Select one of these **Storage class** values.
 - `ImportedDefine (Custom)`
 - Set the external **Header File** as `rtwdemo_importedmacros.h`. An external header file is required for the `ImportedDefine (Custom)` storage class.

Note: The generated code refers to a variant control variable as a user-defined macro. The generated code does not contain the value of the macro. The value of the variant control variable determines the active variant in the compiled code.

Simulink.Parameter: VSSMODE

Value:

Data type:

Dimensions: Complexity:

Minimum: Maximum:

Units:

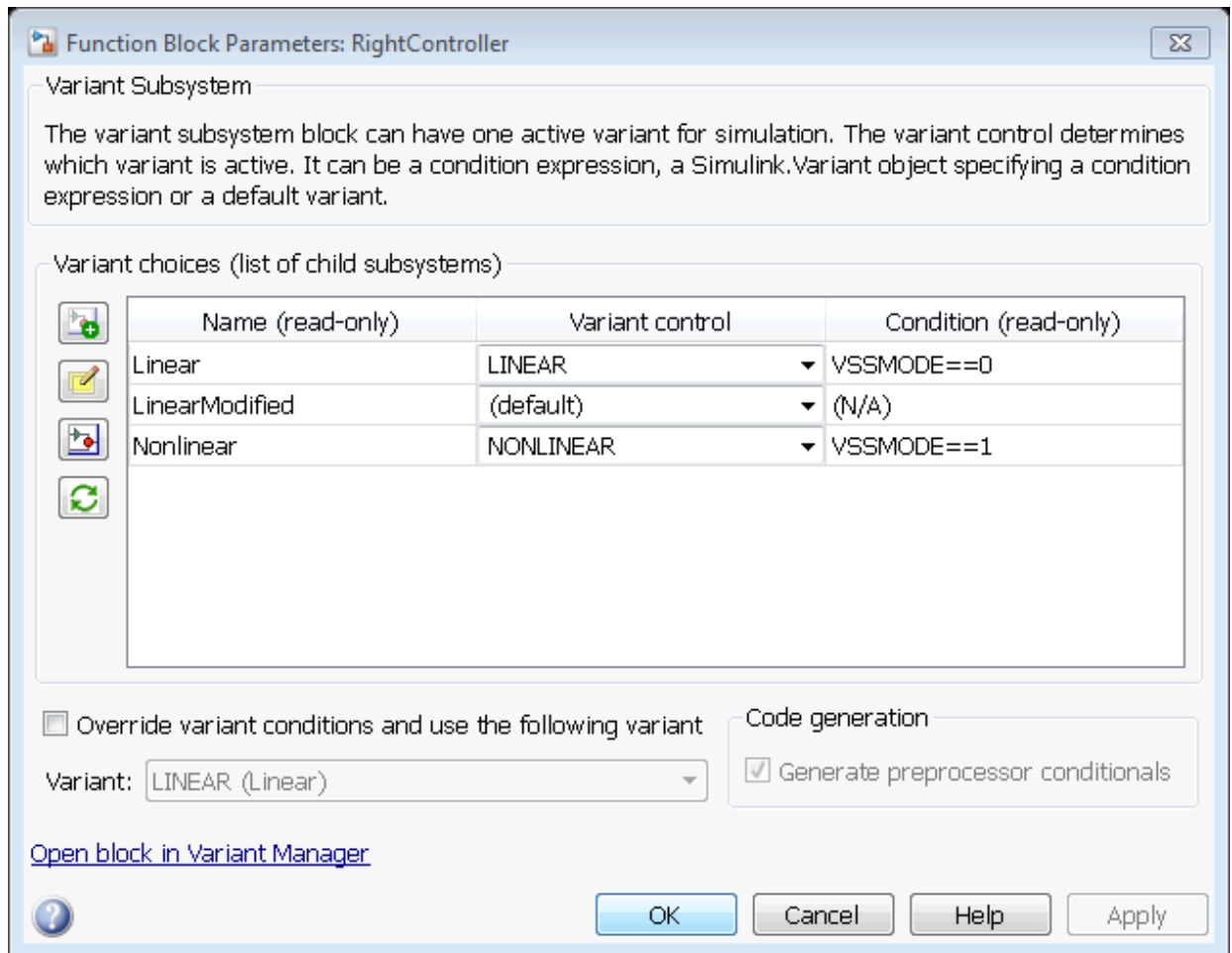
Code generation options

Storage class:

Custom attributes

HeaderFile:

- **CompilerFlag (Custom)**
 - Set the makefile option to enable a variant. In the **Configuration Parameters** dialog box, select the **Code Generation > General** pane. Then set **Build configuration** to **Specify**.
 - In the **C Compiler** field, add a **-D** option. For example, for variant control **VSSMODE**, enter **-D"VSSMODE=1"**.
- 5** Open the **RightController** variant subsystem and create third variant **LinearModified**. Then, open the parameter dialog box for the **RightController** subsystem variant.
 - 6** Set the **Variant control** values for the three variants, setting the **LinearModified** variant as default.



- For each variant subsystem, open the parameter dialog box and select **Treat as atomic unit**.

Configure Model for Generating Preprocessor Conditional Directives

In order to generate preprocessor conditional directives configure your model as follows:

- 1 On the **Code Generation** pane of the **Configuration Parameter** dialog box, specify the **System target file** parameter as `ert.tlc` and clear “Ignore custom storage classes”. In order to generate preprocessor conditionals, you must use custom storage classes.
- 2 On the **Code Generation > Interface** pane of the **Configuration Parameters** dialog box, set **Generate preprocessor conditionals** to **Enable all** or **Use local settings**. This parameter is a global setting that is applicable to all variant blocks in the model.

If you specify **Enable all**, the **Generate preprocessor conditionals** option is disabled on the variant subsystem block and the preprocessor conditions are always generated.

If you specify **Use local settings**, preprocessor conditions are generated only when you set the block parameter `GeneratePreprocessorConditions` to `on`.

For more information, see “Generate preprocessor conditionals”.

View Generated Code

The generated code contains child subsystems of the Variant Subsystem block protected by C preprocessor conditionals. In this case, the selection of the active variant (subsystem) is deferred until the generated code is compiled. Only one variant object, which is encoded in C macros, must evaluate to true.

After building the model, look at the variants in the generated code. `AutoSSVar_types.h` includes the following:

- Call to external header file, `rtwdemo_preprocessor_subsys_types.h`, which contains the macro definitions for the variant control variable `VSSMODE`.

```
/* Includes for objects with custom storage classes. */
#include "rtwdemo_importedmacros.h"
```

- Preprocessor directives defining the variant objects. The values of these macros depend on the value of `VSSMODE`. The condition expression associated with each macro determine the active variant.

```
/* Model Code Variants */
#ifndef LINEAR
#define LINEAR (VSSMODE == 0)
#endif

#ifndef NONLINEAR
#define NONLINEAR (VSSMODE == 1)
```

```
#endif
```

- Check for exactly one variant being active at a time:

```
/* Exactly one variant for '<Root>/LeftController' should be active */  
#if (LINEAR) + (NONLINEAR) != 1  
#error Exactly one variant for '<Root>/LeftController' should be active  
#endif
```

```
/* Exactly one variant for '<Root>/RightController' should be active */  
#if (LINEAR) + (NONLINEAR) > 1  
#error Exactly one variant for '<Root>/LController' should be active  
#endif
```

Calls to the step and initialization functions are conditionally compiled in `rtwdemo_preprocessor_subsys.c`. The conditional for the default variant is also included.

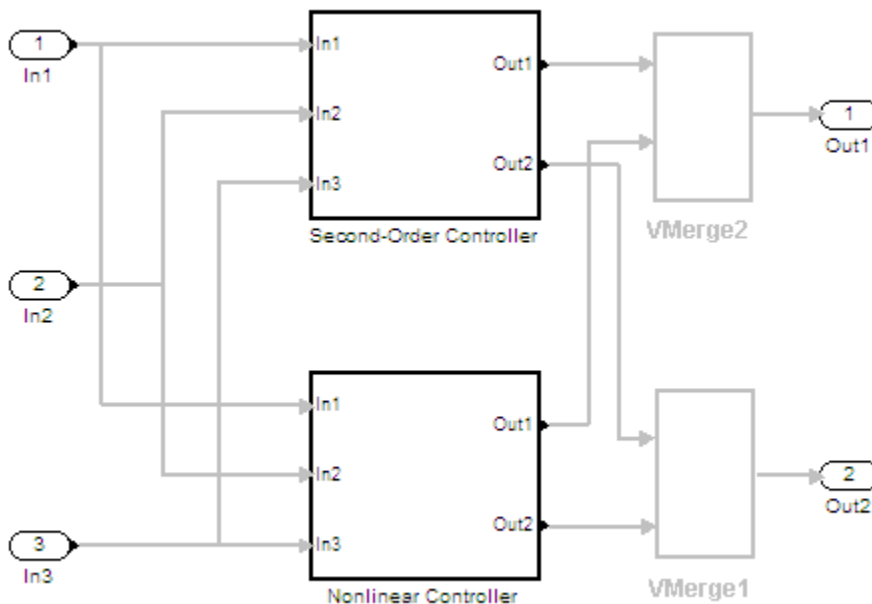
```
#if LINEAR  
  
Linear(rtb_Add1, &rtb_VariantMergeForOutportOut1, &rtDWork.Linear_c);  
  
#elif NONLINEAR  
  
Nonlinear(rtb_Add1, &rtb_VariantMergeForOutportOut1, &rtDWork.Nonlinear_a);  
  
#else  
/* Output and update for atomic system: '<S2>/LinearModified' */  
rtb_VariantMergeForOutportOut1 = look1_binlx(rtb_Add1,...  
rtCP_LookupTable_bp01Dat_j, rtCP_LookupTable_tableDa_j, 4U);  
  
#endif
```

Restrictions on Variant Subsystem Code Generation

To generate preprocessor conditionals, the types of blocks that you can place within the child subsystems of a Variant Subsystem block are limited. Connections are not allowed in the Variant Subsystem block diagram. However, during the code generation process, one `VariantMerge` block is placed at the input of each Output block within the Variant Subsystem block diagram. All of the child subsystems connect to each of the `VariantMerge` blocks.

In the figure below, the code generation process makes the following connections and adds `VariantMerge` blocks to the `sldemo_variant_subsystems` model.

When compared to a generic Merge block the `VariantMerge` block can have only one parameter which is the number of Inputs. The `VariantMerge` block is used for code generation in variant subsystems internally, and is not available externally to be used in models. The number of inputs for `VariantMerge` is determined and wired as shown in the figure below.



The child subsystems of the Variant Subsystem block must be atomic subsystems. Select **Treat as atomic unit** parameter in the Subsystem block parameters dialog, to make the subsystems atomic. The **VariantMerge** blocks are inserted at the output of the subsystems if more than one child subsystems are present. If the source block of a **VariantMerge** block input is nonvirtual, an error message will be displayed during code generation. You must make the source block contiguous, by inserting Signal Conversion blocks inside the variant choices. The **VariantMerge** block does not support variable dimensions through it, so you cannot have child subsystems with different output signal dimensions.

Special Considerations for Generating Preprocessor Conditionals

- The code generation process checks that the inports and outports of a Model Variants block are identical (same port numbers and names) to the corresponding inports and outports of its variants. The build process for simulation does not make this check. Therefore, if your variant block contains mismatched inports or outports, the code generation process issues an error.
- The port numbers and names for each child variant subsystem must belong to a subset of the port numbers and names of the parent Variant Subsystem block.
- The code generation process checks that there is at least one active variant by using the variant control values stored in the base workspace. The variant control that evaluates to `true` becomes the active variant. If none of the variant controls evaluates to `true`, the default variant, if specified, becomes the active variant. The code generation process issues an error if an active variant does not exist.
- If you comment out child subsystems listed in the **Variant Choices** table in the Variant Subsystem block parameter dialog box, the code generator does not generate code for the commented out subsystems.
- If the sample time for a default variant differs from that of the other variant choices, the `#else` preprocessor conditional is not generated for the default variant. Instead, an `#if !(<variant conditions>)` is generated.

Limitations on Generating Code for Variants

- When you are generating code for Model Variants blocks and Variant Subsystem blocks, the blocks cannot have:
 - Continuous states or mass matrices
 - Function call ports
 - Outports with constant sample time
 - Simscape™ blocks
- The Model Variants block and its referenced models must have the same number of inports and outports.
- The port numbers and names for each active child subsystem must belong to a subset of the port numbers and names of the parent Variant Subsystem block.

Generated Code Components Not Compiled Conditionally

The following components in the generated code are not compiled conditionally. This is true even if they are referenced only by code for variant subsystems or models that are conditionally compiled.

- `rtModel` data structure fields
- `#include`'s of utility files
- Global non-constant parameter structure fields; when the configuration parameter **Optimization > Signals and Parameters > Parameter structure** is set to **NonHierarchical**
- Global constant parameter structure fields that are referenced by multiple subsystems activated by different variants
- Parameters that are configured to use an imported, exported, or custom code generation storage class, and are referenced by multiple subsystems that are activated by different variants
- Parameters that are configured to use an imported, exported, or custom code generation storage class, and are used by variant model blocks

Scheduling Considerations

- “Use Discrete and Continuous Time” on page 6-2
- “Optimize Multirate Multitasking Operation on RTOS Targets” on page 6-4

Use Discrete and Continuous Time

In this section...
“Support for Discrete and Continuous Time Blocks” on page 6-2
“Support for Continuous Solvers” on page 6-2
“Support for Stop Time” on page 6-2

Support for Discrete and Continuous Time Blocks

The ERT target supports code generation for discrete and continuous time blocks. If the **Support continuous time** option is selected, you can use these blocks in your models, without restriction.

Note that use of certain blocks is not recommended for production code generation for embedded systems. The Simulink Block Data Type Support table summarizes characteristics of blocks in the Simulink and Fixed-Point Designer block libraries, including whether or not they are recommended for use in production code generation. To view this table, execute the following command and see the “Code Generation Support” column of the table that appears:

```
showblockdatatypetable
```

Support for Continuous Solvers

The ERT target supports continuous solvers. In the **Solver** options dialog, you can select an available solver in the **Solver** menu. (Note that the solver **Type** must be **fixed-step** for use with the ERT target.)

Note Custom targets must be modified to support continuous time. The required modifications are described in “Custom Targets” in the Simulink Coder documentation.

Support for Stop Time

The ERT target supports the stop time for a model. When generating host-based executables, the stop time value is honored if one of the following is true:

- **Classic call interface** is selected on the **Interface** pane

- **External mode** is selected in the **Data exchange** subpane of the **Interface** pane
- **MAT-file logging** is selected on the **Interface** pane

Otherwise, the executable runs indefinitely.

Note: The ERT target provides both generated and static examples of the `ert_main.c` file. The `ert_main.c` file controls the overall model code execution by calling the `model_step` function and optionally checking the `ErrorStatus/StopRequested` flags to terminate execution. For a custom target, if you provide your own custom static `main.c`, you should consider including support for checking these flags.

Optimize Multirate Multitasking Operation on RTOS Targets

In this section...

“Overview” on page 6-4

“Use `rtmStepTask`” on page 6-5

“Scheduling Code for Multirate Multitasking on VxWorks” on page 6-5

“Suppress Redundant Scheduling Calls” on page 6-6

Overview

Using the `rtmStepTask` macro, targets that employ the task management mechanisms of an RTOS can eliminate certain redundant scheduling calls during the execution of tasks in a multirate, multitasking model, thereby improving performance of the generated code.

To understand the optimization that is available for an RTOS target, consider how the ERT target schedules tasks for bareboard targets (where RTOS is not present). The ERT target maintains *scheduling counters* and *event flags* for each subrate task. The scheduling counters are implemented within the real-time model (rtM) data structure as arrays, indexed on task identifier (`tid`).

The scheduling counters are updated by the base-rate task. The counters are clock rate dividers that count up the sample period associated with each subrate task. When a given subrate counter reaches a value that indicates it has a hit, the sample period for that rate has elapsed and the counter is reset to zero. When this occurs, the subrate task must be scheduled for execution.

The event flags indicate whether or not a given task is scheduled for execution. For a multirate, multitasking model, the event flags are maintained by code in the main program for the model. For each task, the code maintains a task counter. When the counter reaches 0, indicating that the task's sample period has elapsed, the event flag for that task is set.

On each time step, the counters and event flags are updated and the base-rate task executes. Then, the scheduling flags are checked in `tid` order, and tasks whose event flag is set is executed. Therefore, tasks are executed in order of priority.

For bareboard targets that cannot rely on an external RTOS, the event flags are mandatory to allow overlapping task preemption. However, an RTOS target uses the

operating system itself to manage overlapping task preemption, making the maintenance of the event flags redundant.

Use `rtmStepTask`

The `rtmStepTask` macro is defined in `model.h` and its syntax is as follows:

```
boolean task_ready = rtmStepTask(rtm, idx);
```

The arguments are:

- `rtm`: pointer to the real-time model structure (`rtm`)
- `idx`: task identifier (`tid`) of the task whose scheduling counter is to be tested

`rtmStepTask` returns `TRUE` if the task's scheduling counter equals zero, indicating that the task should be scheduled for execution on the current time step. Otherwise, it returns `FALSE`.

If your target supports the **Generate an example main program** parameter, you can generate calls to `rtmStepTask` using the TLC function `RTMTaskRunsThisBaseStep`.

Scheduling Code for Multirate Multitasking on VxWorks

The following task scheduling code, from `ertmainlib.tlc`, is designed for multirate multitasking operation on a Wind River® Systems VxWorks® target. The example uses the TLC function `RTMTaskRunsThisBaseStep` to generate calls to the `rtmStepTask` macro. A loop iterates over each subrate task, and `rtmStepTask` is called for each task. If `rtmStepTask` returns `TRUE`, the VxWorks `semGive` function is called, and the VxWorks RTOS schedules the task to run.

```
%assign ifarg = RTMTaskRunsThisBaseStep("i")
for (i = 1; i < %<FcnNumST>; i++) {
    if (%<ifarg>) {
        semGive(taskSemList[i]);
        if (semTake(taskSemList[i],NO_WAIT) != ERROR) {
            logMsg("Rate for SubRate task %d is too fast.\n",i,0,0,0,0);
            semGive(taskSemList[i]);
        }
    }
}
```

Suppress Redundant Scheduling Calls

Redundant scheduling calls are still generated by default for backward compatibility. To change this setting and suppress them, add the following TLC variable definition to your system target file before the `%include "codegenentry.tlc"` statement:

```
%assign SuppressSetEventsForThisBaseRateFcn = 1
```

Data, Function, and File Definition

Data Types

- “What are User-Defined Data Types?” on page 7-2
- “Location of User-Defined Type Definitions” on page 7-3
- “Create and Apply User-Defined Data Types” on page 7-5
- “Specify Persistence Level for Signals and Parameters” on page 7-8
- “Buses” on page 7-11
- “Rename Built-In Data Types” on page 7-26
- “Register mpt User Object Types” on page 7-28
- “Data Type Replacement” on page 7-32
- “Data Definition and Declaration Management” on page 7-43

What are User-Defined Data Types?

User-defined data types are objects of the following data type classes.

- `Simulink.AliasType`
- `Simulink.Bus`
- `Simulink.NumericType`

You can apply user-defined data types to achieve the following objectives in generated code.

- Map your own data type definitions to built-in data types, and specify that your data types are to be used in generated code.
- Optionally, generate `#include` directives specifying own header files that contain your data type definitions. This technique allows you to use legacy data types in generated code.

In general, code generated from user-defined data objects conforms to the properties and attributes of the objects as defined for use in simulation. When generating code from user-defined data objects, the name of the object is the name of the data type that is used in the generated code. For `Simulink.NumericType` objects whose `ISAlias` property is false, the name of the functionally equivalent built-in or fixed-point Simulink data type is used instead.

See Also

“Data Objects”

Location of User-Defined Type Definitions

In this section...

“Omit a HeaderFile Value” on page 7-3

“Specify a HeaderFile Value” on page 7-3

When a model uses `Simulink.DataType` and `Simulink.Bus` objects, the generated code includes corresponding `typedef` definitions. Both `Simulink.DataType` and `Simulink.Bus` objects have a `HeaderFile` property that controls the location of the object's `typedef`. Setting a `HeaderFile` is optional and specific to code generation only.

Omit a HeaderFile Value

If the `HeaderFile` property for a `Simulink.DataType` or `Simulink.Bus` object is left empty, a generated `typedef` for the object appears in the generated file `model_types.h`.

Example

For a `Simulink.NumericType` object named `myfloat` with a `Category` of `double` and without `HeaderFile` property specified, `model_types.h` in the generated code contains:

```
typedef real_T myfloat;
```

Specify a HeaderFile Value

If the `HeaderFile` property for a `Simulink.DataType` or `Simulink.Bus` object is set to a string value,

- The string must be the name of a header file that contains a `typedef` for the object.
- The generated file `model_types.h` contains a `#include` that gives the header file name.

You can use this technique to include legacy or other externally created `typedefs` in generated code. When the generated code compiles, the specified header file must be accessible on the build process include path.

HeaderFile Property Syntax

The `HeaderFile` property should include the desired preprocessor delimiter (`" "` or `'<>'`), as in the following examples.

This example:

```
myfloat.HeaderFile = '<legacy_types.h>'
```

generates the directive:

```
#include <legacy_types.h>
```

This example:

```
myfloat.HeaderFile = '"legacy_types.h"'
```

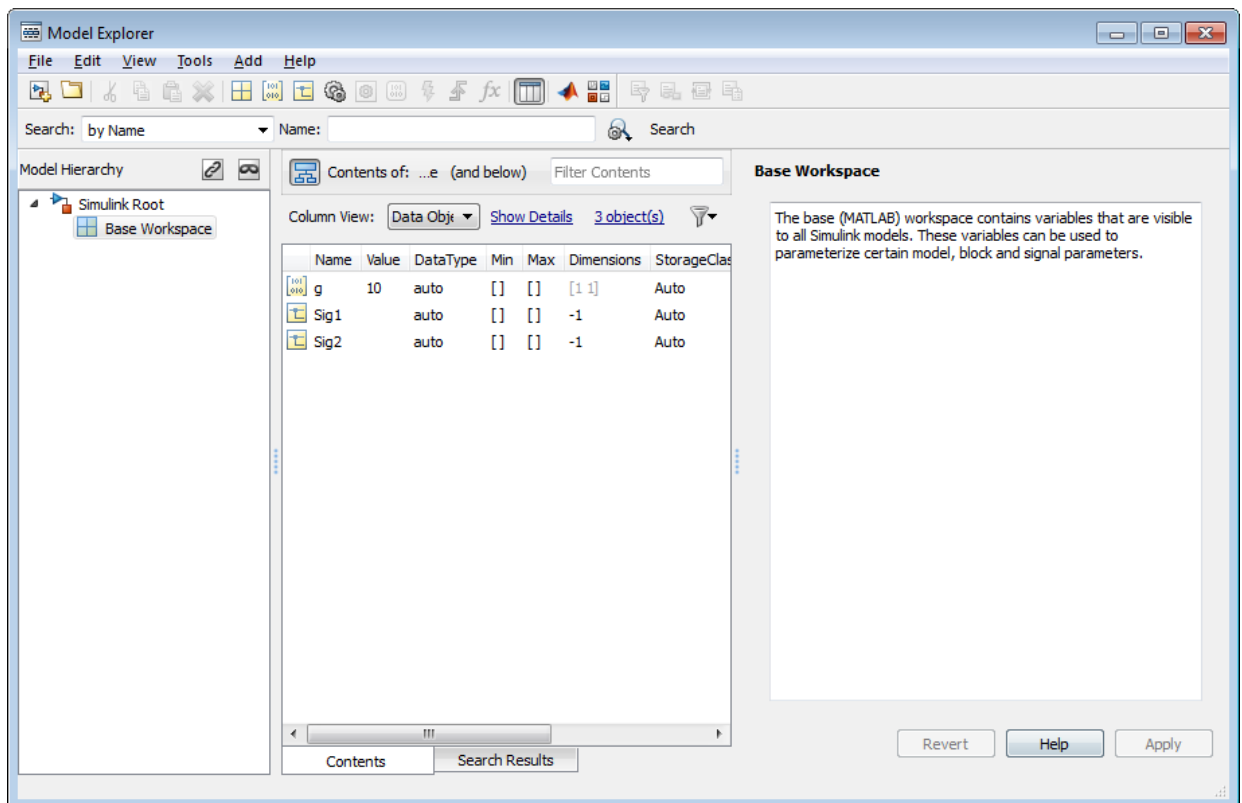
generates the directive:

```
#include "legacy_types.h"
```

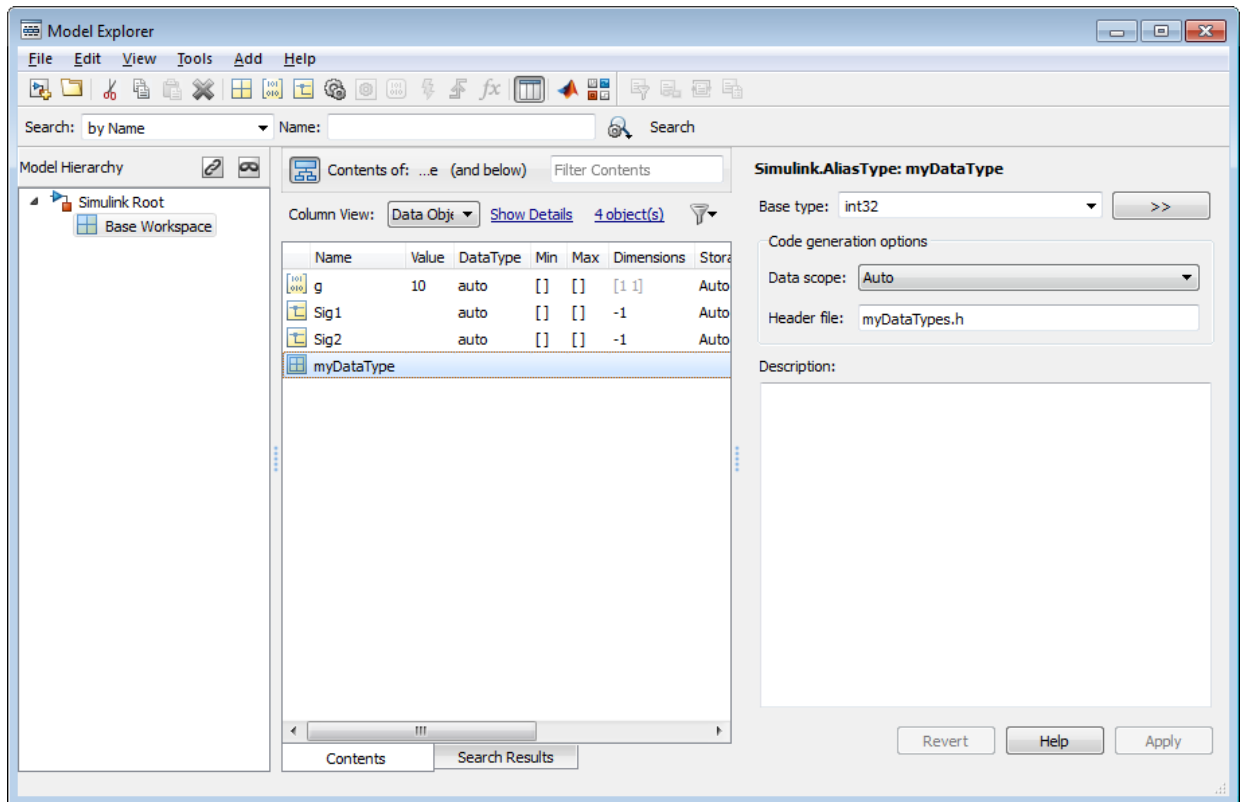

Create and Apply User-Defined Data Types

This example shows how to create user-defined data types and specify them for data objects.

- 1 Open the Model Explorer and create `Simulink.Signal` and `Simulink.Parameter` objects in the base workspace.



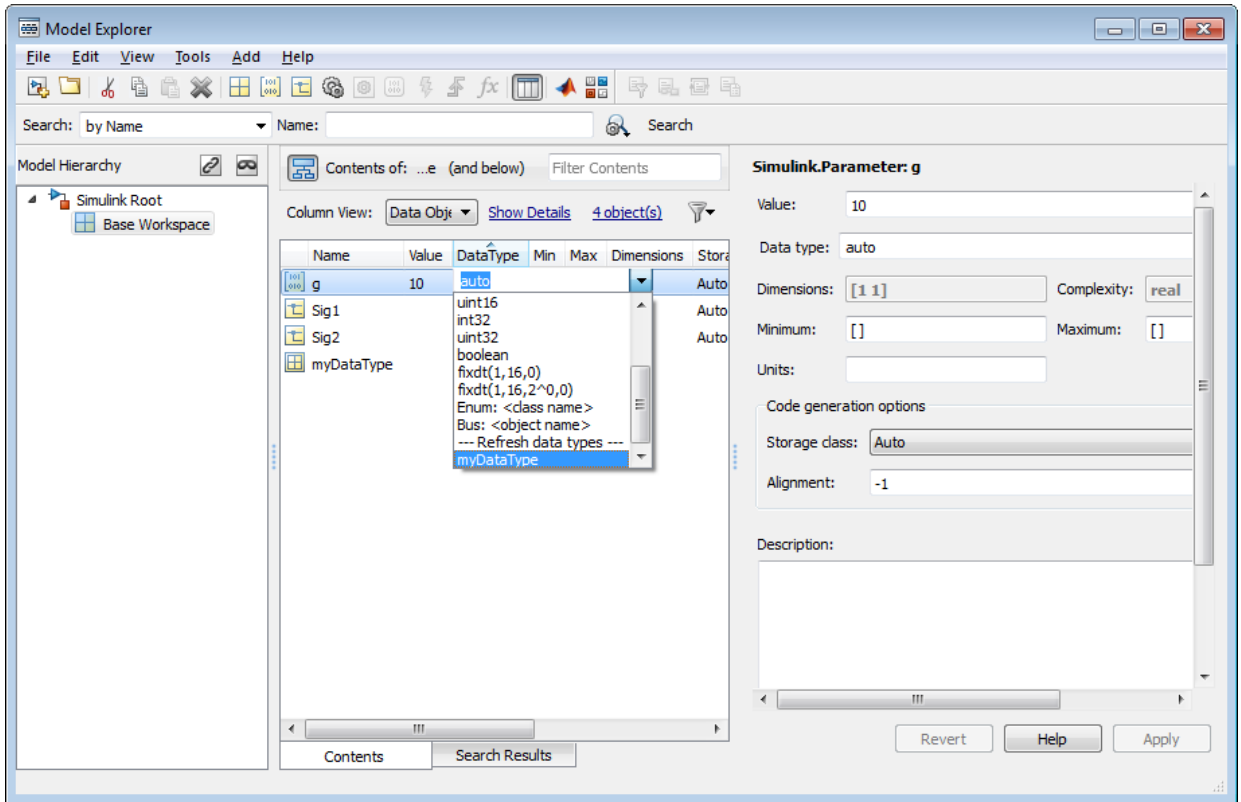
- 2 Click **Add > Simulink.AliasType** to create a data type object.
- 3 Name the object and set its **Base type** to `int32` and **Header file** to `myDataTypes.h`.



- 4 Select the data object for which you want to specify the user-defined data type. Click its **Data Type** field and from the drop down select **Refresh data types**.

This action updates the data type list with the user-defined data type you created.

- 5 Select the user-defined data type.

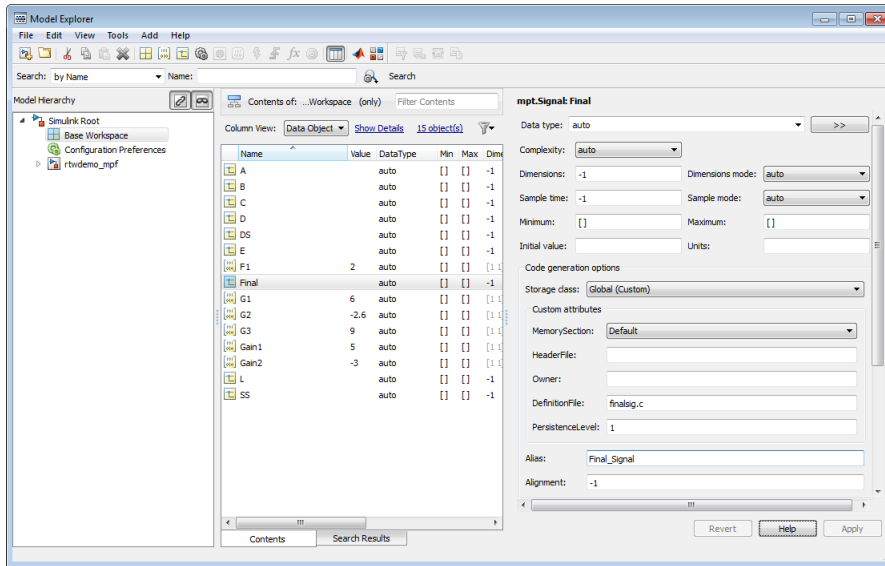


See Also

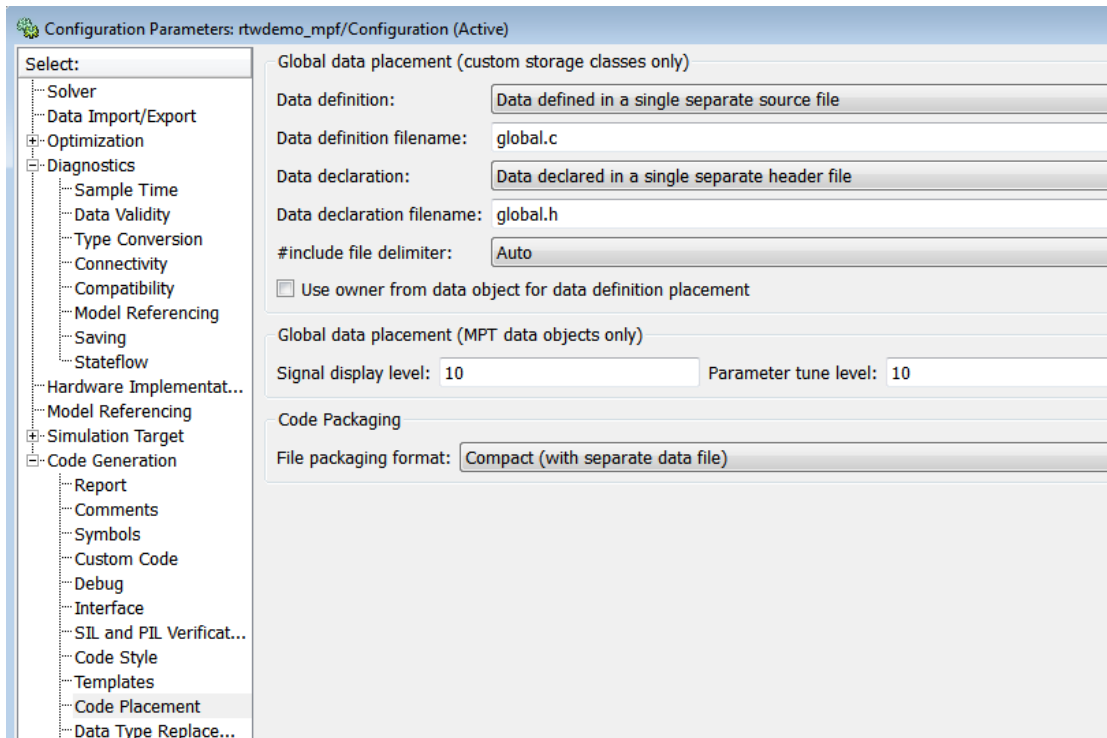
“Create Simulink and mpt Data Objects” on page 7-44 | “Data Objects”

Specify Persistence Level for Signals and Parameters

With this procedure, you can control the persistence level of signal and parameter objects associated with a model. Persistence level allows you to make intermediate variables or parameters global during initial development. At the later stages of development, you can use this procedure to remove these signals and parameters for efficiency. Notice the **Persistence Level** field on the Model Explorer, as illustrated in the figure below. For descriptions of the properties on the Model Explorer, see Parameter and Signal Property Values.



Notice also the **Signal display level** and **Parameter tune level** fields on the **Code Placement** pane of the Configuration Parameters dialog box, as illustrated in the next figure.



The **Signal display level** field allows you to specify whether or not the code generator defines a signal data object as global data in the generated code. The number you specify in this field is relative to the number you specify in the **Persistence level** field. The **Signal display level** number is for *mpt* (module packaging tool) signal data objects in the model. The **Persistence level** number is for a *particular* *mpt* signal data object. If the data object's **Persistence level** is equal to or less than the **Signal display level**, the signal appears in the generated code as global data with the properties (custom attributes) specified in “Create *mpt* Data Objects with Data Object Wizard” on page 7-51. For example, this would occur if **Persistence level** is 2 and **Signal display level** is 5.

Otherwise, the code generator automatically determines how the particular signal data object appears in the generated code. Depending on the settings on the **Optimization** pane of the Configuration Parameters dialog box, the signal data object could appear in the code as local data without the custom attributes you specified for that data object. Or,

based on expression folding, the code generator could remove the data object so that it does not appear in the code. For more information, see “Code Optimization Basics”.

The **Parameter tune level** field allows you to specify whether or not the code generator declares a parameter data object as tunable global data in the generated code.

The number you specify in this field is relative to the number you specify in the **Persistence level** field. The **Parameter tune level** number is for `mpt` parameter data objects in the model. The **Persistence level** number is for a *particular* `mpt` parameter data object. If the data object's **Persistence level** is equal to or less than the **Parameter tune level**, the parameter appears in the generated code with the properties (custom attributes) specified in “Create `mpt` Data Objects with Data Object Wizard” on page 7-51, and thus is tunable. For example, this would occur if **Persistence level** is 2 and **Parameter tune level** is 5.

Otherwise, the parameter is inlined in the generated code, and the code generation settings determine its exact form.

Note that, in the initial stages of development, you might be more concerned about debugging than code size. Or, you might want one or more particular data objects to appear in the code so that you can analyze intermediate calculations of an equation. In this case, you might want to specify the **Parameter tune level (Signal display level for signals)** to be higher than **Persistence level** for some `mpt` parameter (or signal) data objects. This results in larger code size, because the code generator defines the parameter (or signal) data objects as global data, which have the custom properties you specified. As you approach production code generation, however, you might have more concern about reducing the size of the code and less need for debugging or intermediate analyses. In this stage of the tradeoff, you could make the **Parameter tune level (Signal display level for signals)** greater than **Persistence level** for one or more data objects, generate code and observe the results. Repeat until satisfied.

- 1 With the model open, in the Configuration Parameters dialog box, click **Code Generation > Code Placement**.
- 2 Type the desired number in the **Signal display level** or **Parameter tune level** field, and click **Apply**.
- 3 In the Model Explorer, type the desired number in the **Persistence** field for the selected signal or parameter, and click **Apply**.
- 4 Save the model and generate code.

Buses

In this section...

“About Buses and Code Generation” on page 7-11

“Set Bus Diagnostics” on page 7-12

“Optimize Virtual and Nonvirtual Buses” on page 7-12

“Use Single-Rate and Multi-Rate Buses” on page 7-15

“Set Bus Signal Initial Values” on page 7-18

“Use Buses with Atomic Subsystems” on page 7-22

About Buses and Code Generation

When you use buses in a model for which you intend to generate code:

- Setting diagnostic configuration parameters can add to the ease of development.
- The bus implementation techniques used can influence the speed, size, and clarity of that code.
- Some bus implementation techniques that can be useful are not immediately obvious.

This chapter contains guidelines that you can use to improve the results when you work with buses. The guidelines describe techniques for:

- Simplifying the layout of the model
- Increasing the efficiency of generated code
- Defining data structures for function/subsystem interfaces
- Defining data structures that match existing data structures in external C code

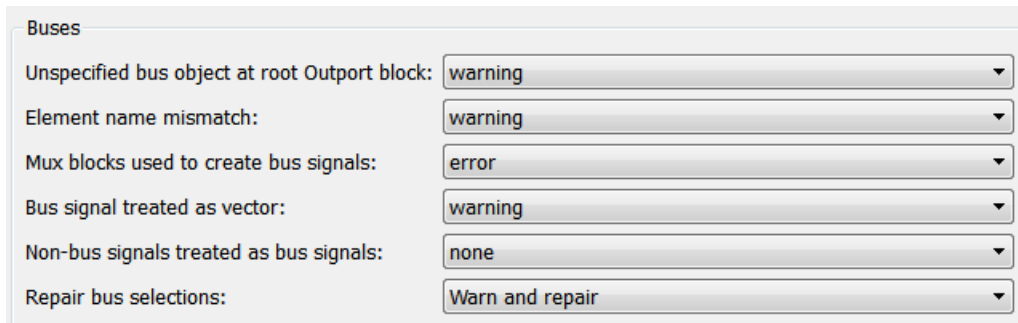
Some trade-offs inevitably exist among speed, size, and clarity. For example, the code for nonvirtual buses is easier to read because the buses appear in the code as structures, but the code for virtual buses is faster because virtual buses do not require copying signal data. The applicability of some guidelines can therefore depend on where you are in the application development process.

This chapter focuses on optimizations that are useful for final production code. Before you read this chapter, read “Composite Signals”. This topic assumes that you understand

the concepts and procedures described in that one, including the blocks used for creating and manipulating buses.

Set Bus Diagnostics

Simulink provides diagnostics that you can use to optimize bus usage. Set the following values on the **Configuration Parameters > Diagnostics > Connectivity** pane:



Buses	
Unspecified bus object at root Output block:	warning
Element name mismatch:	warning
Mux blocks used to create bus signals:	error
Bus signal treated as vector:	warning
Non-bus signals treated as bus signals:	none
Repair bus selections:	Warn and repair

Bus signal treated as vector is enabled only when **Mux blocks used to create bus signals** is set to **error**. See “Prevent Bus and Mux Mixtures” for more information.

Optimize Virtual and Nonvirtual Buses

- “Use Virtual Buses Wherever Possible” on page 7-12
- “Avoid Nonlocal Nested Buses in Nonvirtual Buses” on page 7-13

Use Virtual Buses Wherever Possible

Virtual buses are graphical conveniences that do not affect generated code. As a result, the code generation engine is able to fully optimize the signals in the bus. You should therefore use virtual rather than nonvirtual buses wherever possible. You can convert between virtual and nonvirtual buses using Signal Conversion blocks. In many cases, Simulink automatically converts a virtual bus to a nonvirtual bus when required. For example, a virtual bus input to a Model block becomes a nonvirtual bus without the need for explicit conversion. See for more information.

When are Virtual and Nonvirtual Buses Required?

In some cases, Simulink requires the use of nonvirtual buses:

- For non-auto storage classes
- Inports and Outports of Model blocks
- To generate a specific structure from the bus
- Root level Inport or Outport blocks when the bus has mixed data types

In one case, Simulink requires the use of virtual buses:

- Only virtual buses can be used for bundling function call signals.

Avoid Nonlocal Nested Buses in Nonvirtual Buses

Buses can contain subordinate buses. The storage class of a subordinate bus should be `auto`, which results in a local signal. Setting a subordinate bus to a non-`auto` storage class has two undesirable results:

- Allocation of redundant memory (memory for the subordinate bus object and memory for the final bus object)
- Additional copy operations (first copying to the subordinate bus and then copying from the subordinate bus to the final bus)

In the following example, the final bus is created from local scoped subordinate elements. The resulting assignment operations are relatively efficient:

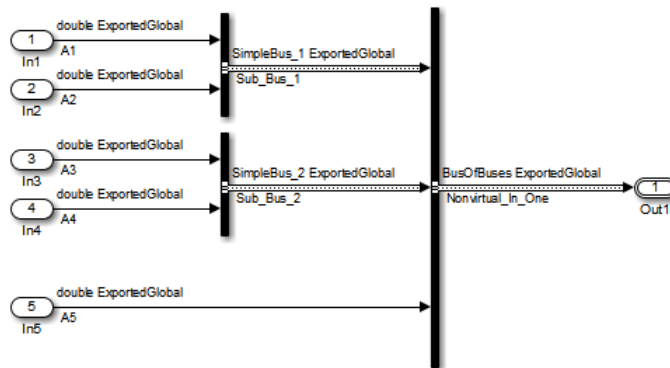


```

34 void bus_in_steps_a_step(void)
35 {
36     Nonvirtual_In_One.Simp_1.enableFlag = A1;
37     Nonvirtual_In_One.Simp_1.calValues[0] = A2[0];
38     Nonvirtual_In_One.Simp_1.calValues[1] = A2[1];
39     Nonvirtual_In_One.Simp_2.Entry_1 = A3;
40     Nonvirtual_In_One.Simp_2.Entry_2_Array[0] = A4[0];
41     Nonvirtual_In_One.Simp_2.Entry_2_Array[1] = A4[1];
42     Nonvirtual_In_One.A_Vector[0] = A5[0];
43     Nonvirtual_In_One.A_Vector[1] = A5[1];
44     Nonvirtual_In_One.A_Vector[2] = A5[2];
45 }

```

By contrast in the next example the subordinate elements `Sub_Bus_1` and `Sub_Bus_2` are global in scope. First the assignment to the subordinate bus occurs (lines 54 – 59) then the copy of the subordinate bus to the main bus (lines 60 – 61). In most cases, this is not an efficient implementation:



```
52 void bus_in_steps_b_step(void)
53 {
54     Sub_bus_1.enableFlag = A1;
55     Sub_bus_2.Entry_1 = A3;
56     Sub_bus_1.calValues[0] = A2[0];
57     Sub_bus_2.Entry_2_Array[0] = A4[0];
58     Sub_bus_1.calValues[1] = A2[1];
59     Sub_bus_2.Entry_2_Array[1] = A4[1];
60     Nonvirtual_In_Steps.Simp_1 = Sub_bus_1;
61     Nonvirtual_In_Steps.Simp_2 = Sub_bus_2;
62     Nonvirtual_In_Steps.A_Vector[0] = A5[0];
63     Nonvirtual_In_Steps.A_Vector[1] = A5[1];
64     Nonvirtual_In_Steps.A_Vector[2] = A5[2];
65 }
```

Use Single-Rate and Multi-Rate Buses

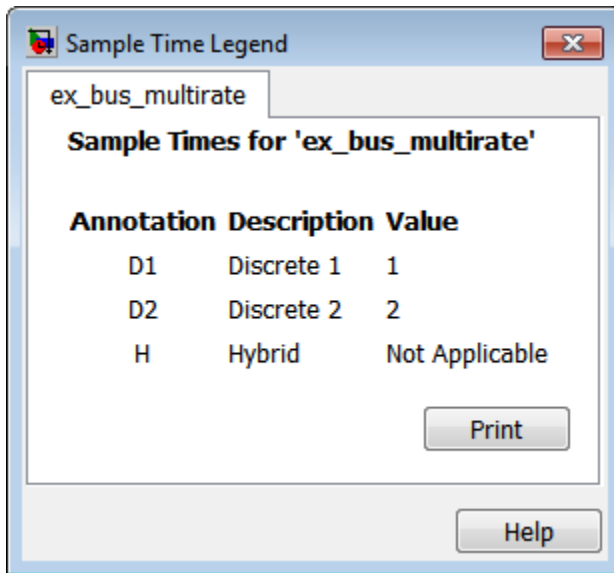
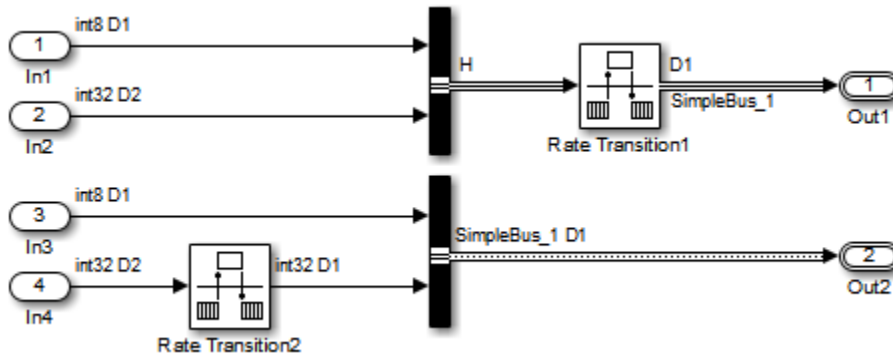
- “Introduction” on page 7-15
- “Techniques for Combining Multiple Rates” on page 7-15
- “Larger Buses and Multiple Rates” on page 7-17
- “Specify Sample Time Rates” on page 7-18

Introduction

Nonvirtual buses do not support multiple rates. Virtual buses support multiple rates as long as the bus does not cross a root level inport or outport. The best techniques for optimizing a bus that contains signals that initially have different rates can depend on the type of the bus and the number of signals.

Techniques for Combining Multiple Rates

The simplest bus contains only two signals. The next figure shows two examples of two-element buses. The first example shows a virtual bus created from two signals that have different rates. The second example shows a nonvirtual bus created from the same two signals. The Sample Time Legend shows the different signal rates:

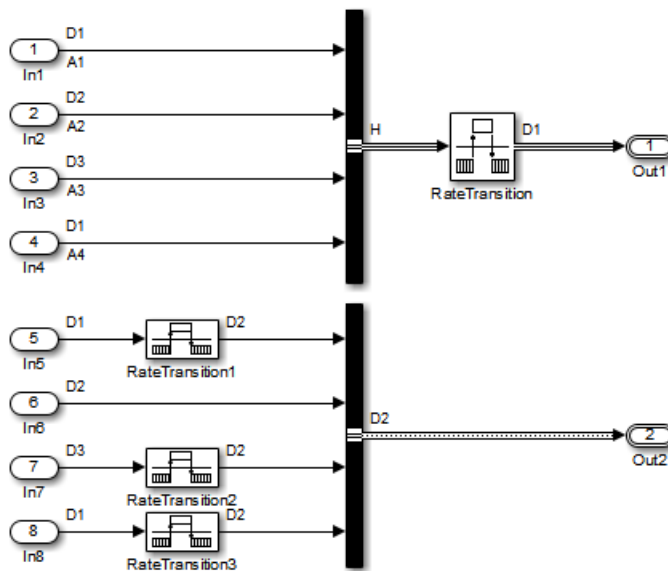


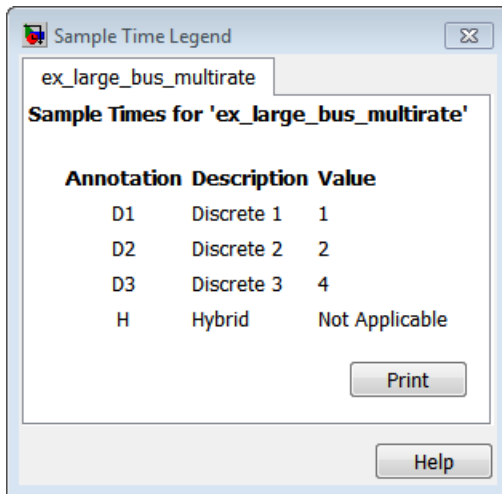
The signals with different rates in the first example can be combined into a virtual bus, because virtual buses support multiple rates. However, a multirate virtual bus cannot connect to a root-level output port. The bus therefore passes through a Rate Transition block that converts it to a single-rate bus, then connects to the Output. This technique is preferable only for virtual buses that contain one or two signals. See “Larger Buses and Multiple Rates” on page 7-17.

The signals with different rates in the second example cannot initially be combined into a nonvirtual bus, because nonvirtual buses do not support multiple rates. One of the signals therefore passes through a Rate Transition block, which converts it to have the same rate as the other signal, then connects to the Bus Creator block. The signals can then combine into a single-rate nonvirtual bus, which can connect to the root-level outport without further conversion.

Larger Buses and Multiple Rates

When you create a multirate virtual bus that contains more than two signals, you can convert the bus to single-rate by applying a Rate Transition block to the output of the Bus Creator block. Use a Rate Transition block on each input signal to give full control over the output rate. As the next figure shows, when a single Rate Transition block is used, the block sets the signals to the fastest rate (D1):





Note that the preferred techniques for a virtual bus with more than two signals, and the required technique for a nonvirtual bus with one or more signals, are the same. Note also that, in the preceding figure, the blocks that perform rate transition are not actual Rate Transition blocks, but other blocks that can change the signal rate as part of some other operation. The identity of the blocks that perform rate transition is not as significant; what matters is that the signal rates match when required.

Specify Sample Time Rates

The sample time for buses should be specified through the signals that define the bus. If the sample times do not match, use Rate Transition (or equivalent) blocks to create a uniform rate, as shown in the previous figures. The signal rates should *not* be set by specifying **Sample Time** values in a Bus Creator block's bus object. Instead, set the sample time for each signal before inputting it to the Bus Creator, and set each **Sample Time** in the corresponding bus object to -1, which indicates the value is inherited.

Set Bus Signal Initial Values

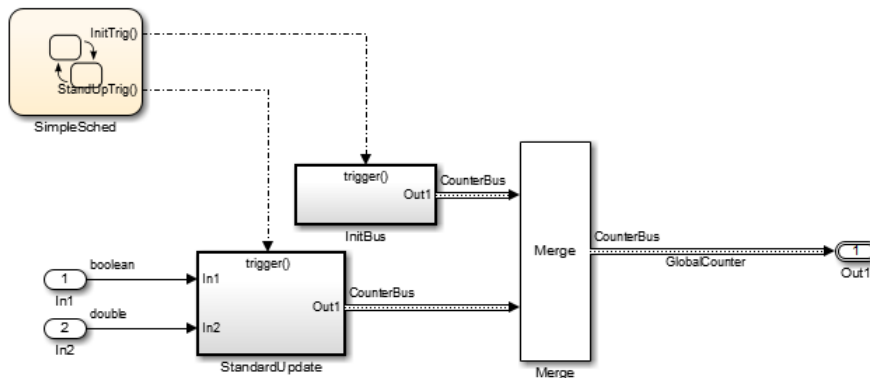
- “Introduction” on page 7-19
- “Initialize Bus Signals in Simulink” on page 7-19
- “Bus Initialization in Stateflow” on page 7-19
- “Create a Bus of Constants” on page 7-21

Introduction

Unlike scalar and vector signals, buses do not provide a direct way to initialize signals. This section describes techniques for initializing bus signals using Simulink, Stateflow, and MATLAB functions.

Initialize Bus Signals in Simulink

In Simulink, you can set initial values on a bus by using a set of conditionally executed subsystems, such as Function-Call subsystems, and a Merge block, as shown in this example:

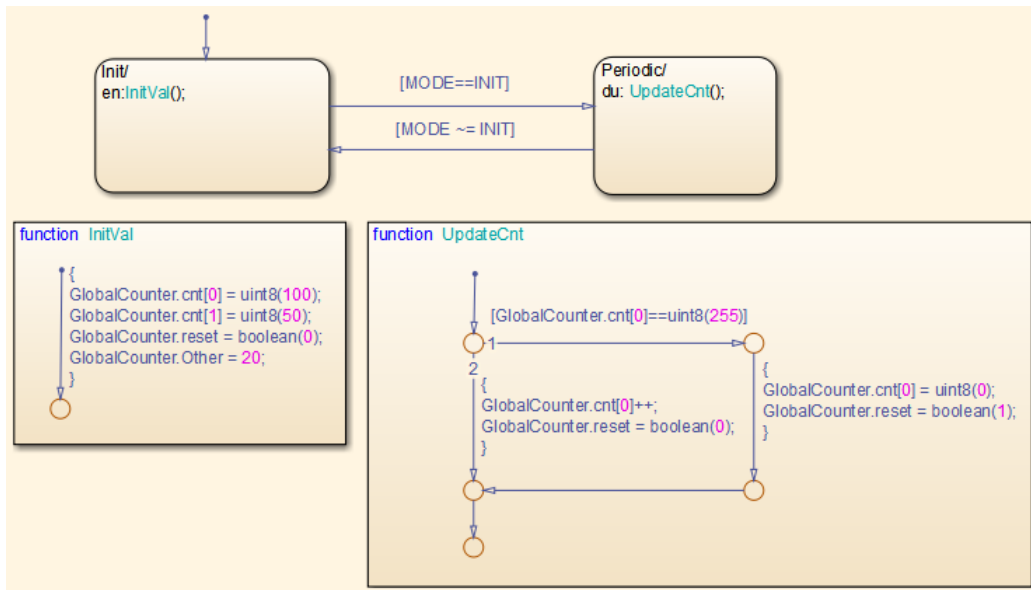


Both subsystems (`InitBus` and `StandardUpdate`) create a bus signal of type `CounterBus`. However, the assignment to the variable `GlobalCounter` is controlled by the `Merge` block. See “Create a Function-Call Subsystem” for more information.

This technique is limited because the `StandardUpdate` subsystem does not use the initial values from the `InitBus` subsystem. If the calculations depend on past information from the bus, consider using Stateflow or MATLAB functions to initialize bus signals.

Bus Initialization in Stateflow

Stateflow and MATLAB functions allow for conditional execution internally. In the following example, the `init` and `update` code are Functions in the Stateflow diagram. This technique simplifies the presentation in the generated code:

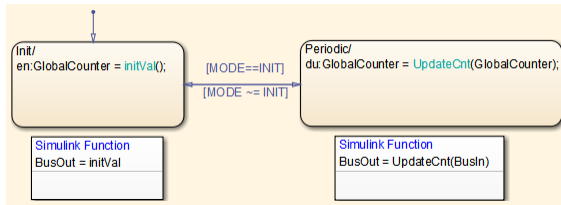


In the generated code, you can see that the UpdateCnt function uses the past value of GlobalCounter.cnt:

```
static void initBus_4_Stateflow_Arr_initVal(void)
{
    GlobalCounter.cnt[0] = 100U;
    GlobalCounter.cnt[1] = 50U;
    GlobalCounter.reset = false;
    GlobalCounter.Other = 20.0;
}

static void initBus_4_Stateflow_A_UpdateCnt(void)
{
    if (GlobalCounter.cnt[0] == 255) {
        GlobalCounter.cnt[0] = 0U;
        GlobalCounter.reset = true;
    } else {
        GlobalCounter.cnt[0] = (uint8_T)(GlobalCounter.cnt[0] + 1);
        GlobalCounter.reset = false;
    }
}
```

The previous example used Stateflow Graphical functions to initialize and update the buses. Alternatively, you can use MATLAB functions or Simulink subsystems embedded in a Stateflow diagram. The next figure illustrates this technique:



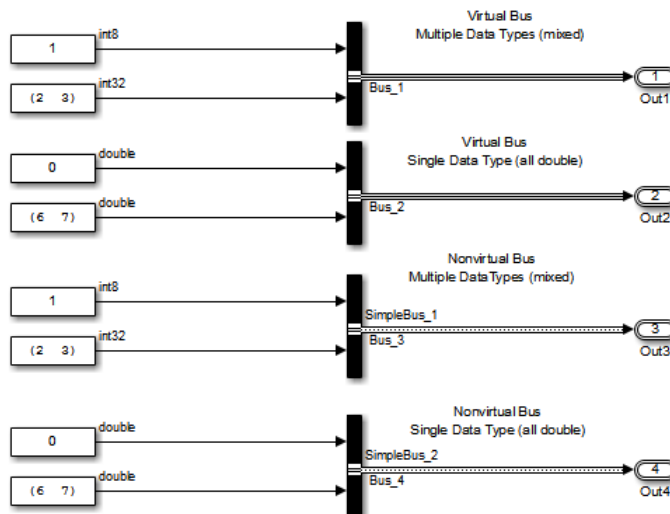
The Simulink subsystems are the same as those used in the earlier Simulink-only example.

Create a Bus of Constants

The code for specifying a bus of constant values will appear in either the **Init** or the **Step** function of the model. The code location depends on the configuration of the bus. In most cases the code appears in the **Step** function. However if the following conditions hold the code will be placed in the **Init** function:

- The bus is a virtual bus
- The signals in the bus have the same data type
- The signals in the bus are constants

In the next figure, only the bus named **Bus_2** meets the requirements:



The code for Bus_2 therefore appears in the Init function. The code for the other buses appears in the Step function:

```
SimpleBus_2 Bus_4;
SimpleBus_1 Bus_3;
ExternalOutputs_busOfConstants_busOfConstants_A_Y;
RT_MODEL_busOfConstants_A_busOfConstants_A_M;
RT_MODEL_busOfConstants_A *busOfConstants_A_M = &busOfConstants_A_M;
void busOfConstants_A_step(void)
{
    busOfConstants_A_Y.Out_1.enableFlag = 1;
    Bus_3.enableFlag = 1;
    Bus_4.Entry_1 = 0.0;
    busOfConstants_A_Y.Out_1.calValues[0] = 2;
    Bus_3.calValues[0] = 2;
    Bus_4.Entry_2_Array[0] = 6.0;
    busOfConstants_A_Y.Out_1.calValues[1] = 3;
    Bus_3.calValues[1] = 3;
    Bus_4.Entry_2_Array[1] = 7.0;
}

void busOfConstants_A_initialize(void)
{
    busOfConstants_A_Y.Out_2[0] = 0.0;
    busOfConstants_A_Y.Out_2[1] = 6.0;
    busOfConstants_A_Y.Out_2[2] = 7.0;
}
```

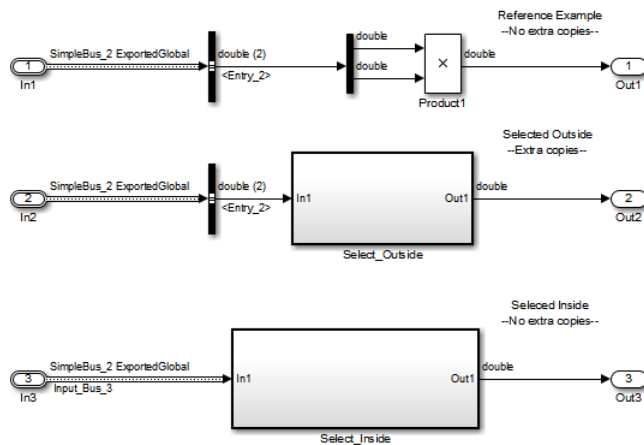
To avoid repeatedly updating a bus of constants, place the bus code into a function-call subsystem, as described in “Initialize Bus Signals in Simulink” on page 7-19. When you use this technique, make sure the function-call subsystem is called at the start of execution. See “Create a Function-Call Subsystem” for more information.

Use Buses with Atomic Subsystems

- “Extract Nonvirtual Bus Signals from Atomic Subsystems” on page 7-22
- “Virtual Bus Signals Crossing Atomic Boundaries” on page 7-23
- “Atomic Subsystems and Buses of Constants” on page 7-24

Extract Nonvirtual Bus Signals from Atomic Subsystems

Selecting signals from a nonvirtual bus can result in unnecessary data copies when those signals cross an atomic boundary. In the following example the same code, a simple multiplication of two elements in a vector, is executed three times:

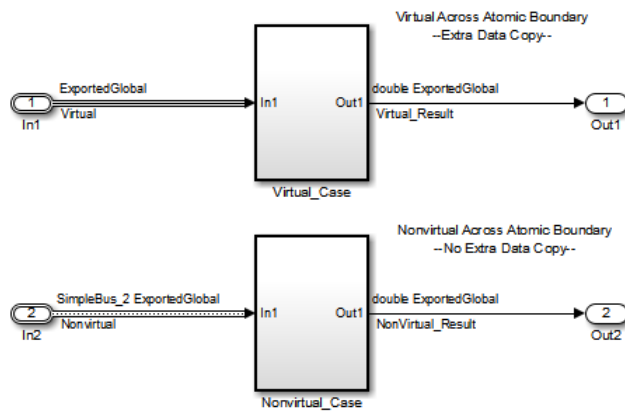


In the second instance when the bus signals are selected outside of the atomic subsystem an unnecessary copy of the bus data is created.

Although this example shows only signals with global scope, both global and local signals show the same behavior: the selection of the signals outside of the model results in an unnecessary copy, while the internal selection does not.

Virtual Bus Signals Crossing Atomic Boundaries

Virtual buses that cross atomic boundaries can result in the creation of unnecessary data copies. The following example shows the data copy that occurs when a virtual bus crosses an atomic boundary:



```

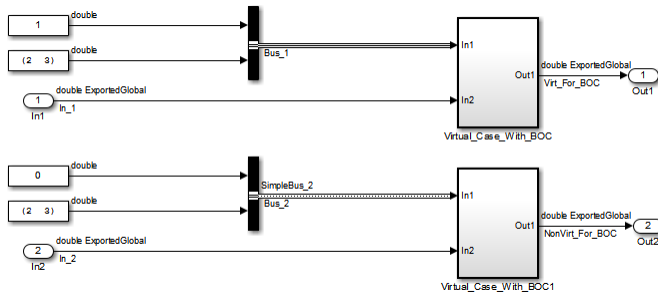
12 void virtualAcrossBo_Nonvirtual_Case(void)
13 {
14     Nonvirtual_Result = Nonvirtual.Entry_2_Array[0] * Nonvirtual.Entry_2_Array[1];
15 }
16
17 void virtualAcrossBound_Virtual_Case(void)
18 {
19     Virtual_Result = virtualAcrossBoundary_B.Entry_2_Array[0] *
20     virtualAcrossBoundary_B.Entry_2_Array[1];
21 }
22
23 void virtualAcrossBoundary_step(void)
24 {
25     virtualAcrossBoundary_B.Entry_2_Array[0] = Virtual.Entry_2_Array[0];
26     virtualAcrossBoundary_B.Entry_2_Array[1] = Virtual.Entry_2_Array[1];
27     virtualAcrossBound_Virtual_Case();
28     virtualAcrossBo_Nonvirtual_Case();
29 }

```

Lines 25–26 show the signals being selected out of the bus before they are used in the function on lines 19–20. By comparison the nonvirtual bus does not require the use of temporary variables.

Atomic Subsystems and Buses of Constants

If the bus passed into an atomic subsystem consists exclusively of constants, using a virtual bus is more efficient, because Simulink is able to inline the constant values into the code:



```

void virtualAc_Virtual_Case_With_BOC(void)
{
    Virt_For_BOC = 6.0 * In_1;
}

void virtualA_Virtual_Case_With_BOC1(void)
{
    NonVirt_For_BOC = virtualAcrossBoundaryBOC_B.Bus_2.Entry_2_Array[0] *
        virtualAcrossBoundaryBOC_B.Bus_2.Entry_2_Array[1] * In_2;
}

void virtualAcrossBoundaryBOC_step(void)
{
    virtualAc_Virtual_Case_With_BOC();
    virtualAcrossBoundaryBOC_B.Bus_2.Entry_1 = 1.0;
    virtualAcrossBoundaryBOC_B.Bus_2.Entry_2_Array[0] = 2.0;
    virtualAcrossBoundaryBOC_B.Bus_2.Entry_2_Array[1] = 3.0;
    virtualA_Virtual_Case_With_BOC1();
}

```

Rename Built-In Data Types

You can replace built-in data type names with user-defined replacement data type names in the generated code for a model.

To configure replacement data types,

- 1 In the Model Editor, set the simulation mode of the open model. From the **Simulation** menu, select **Normal**, **Software-in-the-loop (SIL)**, or **Processor-in-the-loop (PIL)**.
- 2 In the Configuration Parameters dialog box, click **Code Generation > Data Type Replacement > Replace data type names in the generated code**. A **Data type names** table appears. The table lists each Simulink built-in data type name with its corresponding code generation data type name.

Replace data type names in the generated code

Data type names

Simulink Name	Code Generation Name	Replacement Name
double	real_T	<input type="text"/>
single	real32_T	<input type="text"/>
int32	int32_T	<input type="text"/>
int16	int16_T	<input type="text"/>
int8	int8_T	<input type="text"/>
uint32	uint32_T	<input type="text"/>
uint16	uint16_T	<input type="text"/>
uint8	uint8_T	<input type="text"/>
boolean	boolean_T	<input type="text"/>
int	int_T	<input type="text"/>
uint	uint_T	<input type="text"/>
char	char_T	<input type="text"/>

- 3 Fill in fields in the third column with your replacement data types. Each replacement data type should be the name of a `Simulink.AliasType` object that exists in the base workspace. Replacements may be specified or not for each individual built-in type.

For each replacement data type you enter, the `BaseType` property of the associated `Simulink.AliasType` object must be consistent with the built-in data type it replaces.

- For `double`, `single`, `int32`, `int16`, `int8`, `uint32`, `uint16`, and `uint8`, the replacement data type's **BaseType** must match the data type.
- For `boolean`, the replacement data type's **BaseType** must be either an 8-bit integer or an integer of the size displayed for **Number of bits: int** on the **Hardware Implementation** pane of the Configuration Parameters dialog box.
- For `int`, `uint`, and `char`, the replacement data type's size must match the size displayed for **Number of bits: int** or **Number of bits: char** on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

An error occurs, if

- A replacement data type specification is inconsistent.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Register mpt User Object Types

In this section...
“Introduction” on page 7-28
“Register mpt User Object Types Using <code>sl_customization.m</code> ” on page 7-28
“mpt User Object Type Customization Using <code>sl_customization.m</code> ” on page 7-30

Introduction

Embedded Coder software allows you to create custom `mpt` object types and specify properties and property values to be associated with them (see “Create mpt Data Objects with Data Object Wizard” on page 7-51). Once created, a user object type can be applied to data objects displayed in Model Explorer. When you apply a user object type to a data object, by selecting a type name in the **User object type** pull-down list in Model Explorer, the data object is automatically populated with the properties and property values that you specified for the user object type.

To register `mpt` user object type customizations, use the Simulink customization file `sl_customization.m`. This file is a mechanism that allows you to use MATLAB code to perform customizations of the standard Simulink user interface. The Simulink software reads the `sl_customization.m` file, if present on the MATLAB path, when it starts and the customizations specified in the file are applied to the Simulink session. For more information on the `sl_customization.m` customization file, see “Registering Customizations”.

Register mpt User Object Types Using `sl_customization.m`

To register `mpt` user object type customizations, you create an instance of `sl_customization.m` and include it on the MATLAB path of the Simulink installation that you want to customize. The `sl_customization` function accepts one argument: a handle to a customization manager object. For example,

```
function sl_customization(cm)
```

As a starting point for your customizations, the `sl_customization` function must first get the default (factory) customizations, using the following assignment statement:


```
hObj = cm.slDataObjectCustomizer;
```

You then invoke methods to register your customizations. The customization manager object includes the following methods for registering mpt user object type customizations:

- `addMPTObjectType(hObj, objectTypeName, classtype, propName1, propValue1, propName2, propValue2, ...)`

```
addMPTObjectType(hObj, objectTypeName, classtype, {propName1,
propName2, ...}, {propValue1, propValue2, ...})
```

Registers the specified user object type, along with specified values for object properties, and adds the object type to the top of the user object type list, as displayed in the **User object type** pull-down list in the Model Explorer.

- `objectTypeName` — Name of the user object type
- `clasType` — Class to which the user object type applies: 'Signal', 'Parameter', or 'Both'
- `propName` — Name of a property of an mpt or mpt-derived data object to be populated with a corresponding `propValue` when the registered user object type is selected
- `propValue` — Specifies the value for a corresponding `propName`
- `moveMPTObjectTypeToTop(hObj, objectTypeName)`

Moves the specified user object type to the top of the user object type list, as displayed in the **User object type** pull-down list in the Model Explorer.

- `moveMPTObjectTypeToEnd(hObj, objectTypeName)`

Moves the specified user object type to the end of the user object type list, as displayed in the **User object type** pull-down list in the Model Explorer.

- `removeMPTObjectType(hObj, objectTypeName)`

Removes the specified user object type from the user object type list.

Your instance of the `sl_customization` function should use these methods to register mpt object type customizations for your Simulink installation.

The Simulink software reads the `sl_customization.m` file when it starts. If you subsequently change the file, to use the changes, you must restart your MATLAB session.

mpt User Object Type Customization Using `sl_customization.m`

The `sl_customization.m` file shown in `sl_customization.m` for mpt Object Type Customizations uses the `addMPTObjectType` method to register the user signal types `EngineType` and `FuelType` for mpt objects.

`sl_customization.m` for mpt Object Type Customizations

```
function sl_customization(cm)
% Register user customizations

% Get default (factory) customizations
hObj = cm.slDataObjectCustomizer;

% Add commonly used signal types
hObj.addMPTObjectType(...
    'EngineType', 'Signal', ...
    'DataType', 'uint8', ...
    'Min', 0, ...
    'Max', 255, ...
    'DocUnits', 'm/sec');

hObj.addMPTObjectType(...
    'FuelType', 'Signal', ...
    'DataType', 'int16', ...
    'Min', -12, ...
    'Max', 3000, ...
    'DocUnits', 'mg/hr');

end
```

If you include the above file on the MATLAB path of the Simulink installation that you want to customize, the specified customizations will appear in Model Explorer. For example, you could view the customizations as follows:

- 1 Start a MATLAB session.
- 2 Open Model Explorer, for example, by entering the MATLAB command `daexplr`.
- 3 Select **Base Workspace**.
- 4 Add an mpt signal, for example, by selecting **Add > Add Custom**.
- 5 In the right-hand pane display for the added mpt signal, examine the **User object type** drop-down list, noting the impact of the changes specified in `sl_customization.m` for mpt Object Type Customizations.

- 6 From the **User object type** drop-down list, select one of the registered user signal types, for example, `FuelType`, and verify that the displayed settings are consistent with the arguments specified to the `addMPTObjectType` method in `sl_customization.m`.

Data Type Replacement

In this section...
“Replace Built-In Data Type Names” on page 7-32
“Replace Built-In Data Type Names with Replacement Data Type Names” on page 7-35
“Programmatically Replace Built-In Data Type Names” on page 7-40
“Replace <code>boolean</code> with an Integer Data Type” on page 7-40
“Data Type Replacement Limitations” on page 7-42

Replace Built-In Data Type Names

In the generated code for a model, you can replace built-in data type names with user-defined names.

To configure replacement data type names:

- 1 In the Model Editor, set the simulation mode of the open model. From the **Simulation > Mode** menu, select **Normal**, **Software-in-the-loop (SIL)**, or **Processor-in-the-loop (PIL)**.
- 2 In the Configuration Parameters dialog box, click **Code Generation > Data Type Replacement**. Select **Replace data type names in the generated code**. The **Data type names** table lists each Simulink built-in data type name with its corresponding code generation data type name.

Replace data type names in the generated code

Data type names

Simulink Name	Code Generation Name	Replacement Name
double	real_T	<input type="text"/>
single	real32_T	<input type="text"/>
int32	int32_T	<input type="text"/>
int16	int16_T	<input type="text"/>
int8	int8_T	<input type="text"/>
uint32	uint32_T	<input type="text"/>
uint16	uint16_T	<input type="text"/>
uint8	uint8_T	<input type="text"/>
boolean	boolean_T	<input type="text"/>
int	int_T	<input type="text"/>
uint	uint_T	<input type="text"/>
char	char_T	<input type="text"/>

- 3** Fill in the **Replacement Name** column with values to replace the default names in the **Code Generation Name** column. Specify one of the following:
- The name of a `Simulink.AliasType` object that exists in the base workspace. Set the `BaseType` object property value to the corresponding **Simulink Name** data type. When you use a `Simulink.AliasType` object, you can replace a data type name with a valid `Simulink.AliasType` name.
 - The **Simulink Name** data type. This name replaces the name in the generated code. Using the **Simulink Name**, you can replace all values except `real_T` and `real32_T`. For information on how you specify replacement names for `boolean_T`, `int_T`, `uint_T`, and `char_T`, see the following table.
 - The name of a `Simulink.NumericType` object that exists in the base workspace. Set the `DataTypeMode` object property value to the corresponding **Simulink Name** data type. Using `Simulink.NumericType`, you can define replacement names for `real_T`, `real32_T`, and `boolean_T`.

Specify the Data Type for the Code Generation Name

Code Generation Name	Simulink.AliasType BaseType	Simulink Name	Simulink.NumericType DataTypeMode
real_T	double	—	Double

Code Generation Name	Simulink.AliasType BaseType	Simulink Name	Simulink.NumericType DataTypeMode
real32_T	single	–	Single
int32_T	int32	int32	–
int16_T	int16	int16	–
int8_T	int8	int8	–
uint32_T	uint32	uint32	–
uint16_T	uint16	uint16	–
uint8_T	uint8	uint8	–
boolean_T	uint8 or int8 or intn*	uint8 or int8 or intn*	Boolean
int_T	intn*	intn*	–
uint_T	uintn*	uintn*	–
char_T	intn*	intn*	–

* n is the number of bits displayed in the Configuration Parameters box, **Hardware Implementation** pane, either **Number of bits: int** or **Number of bits: char**, depending on the data type.

To specify the data type for **Replacement Name** in column one of the table, use the value listed in the column representing the replacement representation.

For example, to use a `Simulink.AliasType` object to change `uint32_T` to the name `unsigned_integer` in the generated code:

- In the Command Window, create the `unsigned_integer` object:

```
>> unsigned_integer=Simulink.AliasType;
```

- Look in the table under Code Generation Name for `uint32_T`.

Assign the value in the column labeled `Simulink.AliasType BaseType`, to `unsigned_integer.BaseType`:

```
>> unsigned_integer.BaseType='uint32';
```

- In the Data Type Replacement table, in the **Replacement Name** column, in the `uint32_T` row, enter `unsigned_integer`.

An error occurs, if:

- A replacement data type specification does not correspond to the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

For more information, see “`Simulink.AliasType`” and “`Simulink.NumericType`”.

Replace Built-In Data Type Names with Replacement Data Type Names

If your application requires you to replace built-in data type names with user-defined replacement data type names in the generated code, you can do so from the **Code Generation > Data Type Replacement** pane of the Configuration Parameters dialog box.

This pane is available for ERT target-based Simulink models. In addition to providing a mechanism for mapping built-in data types to user-defined replacement data types, this feature:

- Performs consistency checks so that your specified data type replacements are consistent with your model's data types.
- Allows many-to-one data type replacement, the ability to map multiple built-in data types to one replacement data type in generated code. For example, built-in data types `uint8` and `boolean` can be replaced in your generated code by a data type `U8` that you have previously defined.

Note: For limitations that apply, see “Data Type Replacement Limitations” on page 7-42.

If you select **Replace data type names in the generated code**, the **Data type names** table is displayed.

The table **Data type names** lists each Simulink built-in data type name along with its code generation data type name. Selectively fill in fields in the third column with your replacement data types. You can specify replacements or not for each individual built-in type. Each replacement data type must be one of the following:

- The name of a `Simulink.AliasType` object that exists in the base workspace. Set the `BaseType` object property value to the corresponding **Simulink Name** data type.

When you use a `Simulink.AliasType` object, you can replace a data type name with a valid `Simulink.AliasType` name.

- The **Simulink Name** data type. This name replaces the name in the generated code. Using the **Simulink Name**, you can replace all values except `real_T` and `real32_T`. For information on how you specify replacement names for `boolean_T`, `int_T`, `uint_T`, and `char_T`, see the following table.
- The name of a `Simulink.NumericType` object that exists in the base workspace. Set the `DataTypeMode` object property value to the corresponding **Simulink Name** data type. Using `Simulink.NumericType`, you can define replacement names for `real_T`, `real32_T`, and `boolean_T`.

An error occurs if a replacement data type specification is inconsistent.

Specify the Data Type for the Code Generation Name

Code Generation Name	<code>Simulink.AliasType</code> <code>BaseType</code>	<code>Simulink Name</code>	<code>Simulink.NumericType</code> <code>DataTypeMode</code>
<code>real_T</code>	<code>double</code>	–	<code>Double</code>
<code>real32_T</code>	<code>single</code>	–	<code>Single</code>
<code>int32_T</code>	<code>int32</code>	<code>int32</code>	–
<code>int16_T</code>	<code>int16</code>	<code>int16</code>	–
<code>int8_T</code>	<code>int8</code>	<code>int8</code>	–
<code>uint32_T</code>	<code>uint32</code>	<code>uint32</code>	–
<code>uint16_T</code>	<code>uint16</code>	<code>uint16</code>	–
<code>uint8_T</code>	<code>uint8</code>	<code>uint8</code>	–
<code>boolean_T</code>	<code>uint8</code> or <code>int8</code> or <code>intn*</code>	<code>uint8</code> or <code>int8</code> or <code>intn*</code>	<code>Boolean</code>
<code>int_T</code>	<code>intn*</code>	<code>intn*</code>	–
<code>uint_T</code>	<code>uintn*</code>	<code>uintn*</code>	–
<code>char_T</code>	<code>intn*</code>	<code>intn*</code>	–

* *n* is the number of bits displayed in the Configuration Parameters box, **Hardware Implementation** pane, either **Number of bits: int** or **Number of bits: char**, depending on the data type.

It is not recommended to use data type replacement to work around data type checking. Do not intentionally replace a data type with a type that has inconsistent characteristics, such as replacing `real_T` with a type name that maps to a single-precision data type (inconsistent size). Remapping to inconsistent types can introduce significant numerical differences, relative to simulation results, and erroneous behavior. Examples of inconsistencies include:

- Floating-point precision differences, due to the floating-point number representation and arithmetic unit on the target processor using a smaller number of bits.
- Literal values and tolerances, calculated and used during simulation and incorporated into generated code, which are inconsistent with the actual data types used on the target system.
- Algorithm convergence and run-time library behavior observed during simulation not being representative of the target system.

For example, suppose that you have previously defined the following replacement data types, which exist as `Simulink.AliasType` objects in the base workspace.

User-Defined Name	Description
FLOAT64	64-bit floating point
FLOAT32	32-bit floating point
S32	32-bit integer
S16	16-bit integer
S8	8-bit integer
U32	Unsigned 32-bit integer
U16	Unsigned 16-bit integer
U8	Unsigned 8-bit integer
CHAR	Character data

You can fill in the **Data Type Replacement** pane with a one-to-one replacement mapping, as follows:

Replace data type names in the generated code

Data type names

Simulink Name	Code Generation Name	Replacement Name
double	real_T	FLOAT64
single	real32_T	FLOAT32
int32	int32_T	S32
int16	int16_T	S16
int8	int8_T	S8
uint32	uint32_T	U32
uint16	uint16_T	U16
uint8	uint8_T	U8
boolean	boolean_T	
int	int_T	
uint	uint_T	
char	char_T	CHAR

You can also apply a many-to-one data type replacement mapping. For example, in the following display:

- `int32` and `int` are replaced with user type `S32`.
- `uint32` and `uint` are replaced with user type `U32`.
- `uint8` and `boolean` are replaced with user type `U8`.

Note: Many-to-one data type replacement does not support the `char` (`char_T`) built-in data type. Use `char` only in one-to-one data type replacements.

Data type names		
Simulink Name	Code Generation Name	Replacement Name
double	real_T	
single	real32_T	
int32	int32_T	S32
int16	int16_T	
int8	int8_T	
uint32	uint32_T	U32
uint16	uint16_T	
uint8	uint8_T	U8
boolean	boolean_T	U8
int	int_T	S32
uint	uint_T	U32
char	char_T	

The user-defined replacement types you specify appear in your model's generated code in place of the corresponding built-in data types. For example, if you specify user-defined data type `FLOAT64` to replace built-in data type `real_T` (`double`), the original generated code shown in `Generated Code with real_T Built-In Data Type` becomes the modified generated code in `Generated Code with FLOAT64 Replacement Data Type`.

Generated Code with `real_T` Built-In Data Type

```
...
/* Model initialize function */
void sinwave_initialize(void)
{
...
  {real_T *dwork_ptr = (real_T *) &sinwave_DWork.lastSin;
...
}
...
```

Generated Code with `FLOAT64` Replacement Data Type

```
...
/* Model initialize function */
void sinwave_initialize(void)
```

```
{
...
  {FLOAT64 *dwork_ptr = (FLOAT64 *) &sinwave_DWork.lastSin;
...
}
```

Programmatically Replace Built-In Data Type Names

To replace the built-in data type names for your model, modify the `ReplacementTypes` property, which is managed as a structure. The following lines of example code show how you can modify the `ReplacementTypes` property to replace the built-in data type names `int8`, `uint8`, and `boolean` with user-defined data type names `my_T_S8`, `my_T_U8`, and `my_T_BOOL`, respectively.

```
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);
```

Replace `boolean` with an Integer Data Type

Using data type replacement, you can replace the `boolean` built-in data type with an integer type in the generated code. The integer type can improve the performance of the generated code on your production hardware:

- `int8`
- `uint8`
- `int n`

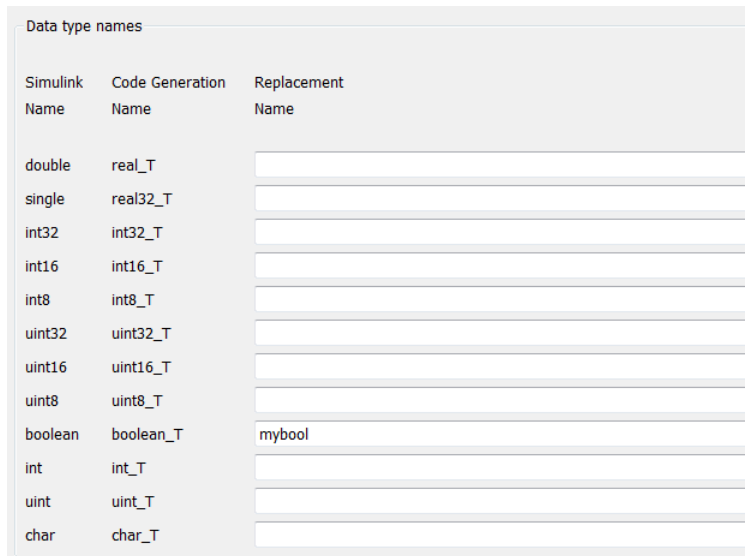
n is 8, 16, or 32, matching the integer word size for the production hardware (for example, `int32` for 32-bit hardware).

For example, to map `boolean` to the `int32` data type, perform the following steps.

- 1 Define a `Simulink.AliasType` object with a base type of `int32`:

```
mybool = Simulink.AliasType;
mybool.BaseType = 'int32';
```

- 2 Open an ERT-based model. In the **Data Type Replacement** pane of the Configuration Parameters dialog box, map `boolean` (`boolean_T`) to the replacement data type `mybool`.



In the resulting generated code, `boolean` is replaced with `mybool`. For example, `rtwtypes.h` contains:

```
/* Generic type definitions ... */
...
typedef int boolean_T;
...
/* Define Simulink Coder replacement data types. */
typedef boolean_T mybool; /* User defined replacement datatype for boolean_T */
```

Boolean data in the generated code is declared with `mybool`. For example, given a model with a Boolean output `Out1`, an `Out1` declaration such as:

```
boolean_T Out1; /* '<Root>/Out1' */
```

instead is generated in `model.h` as:

```
mybool Out1; /* '<Root>/Out1' */
```

Data Type Replacement Limitations

- Data type replacements are ignored during code generation for a model unless the simulation mode of the model is set to **Normal**, **Software-in-the-Loop (SIL)**, or **Processor-in-the-Loop (PIL)**. Set the simulation mode from the **Simulation** menu in the Model Editor.
- Data type replacement does not support multiple levels of mapping. Each replacement data type name maps directly to one or more built-in data types.
- Data type replacement is not supported for simulation target code generation for referenced models.
- If the **Classic call interface** option is selected for your model, data type replacement is not supported.
- Data type replacement occurs during code generation for `.c`, `.cpp`, and `.h` files generated in build folders (including top and referenced model build folders) and in the `_sharedutils` folder. *Exceptions* are as follows:
 - `rtwtypes.h`
 - `multiword_types.h`
 - `model_reference_types.h`
 - `builtin_typeid_types.h`
 - `model_sf.c` or `.cpp` (ERT S-function wrapper)
 - `model_dt.h` (C header file supporting external mode)
 - `model_capi.c` or `.cpp`
 - `model_capi.h`
- Data type replacement is not supported for complex data types.
- Many-to-one data type replacement is not supported for the `char` data type. Attempting to use `char` as part of a many-to-one mapping to a user-defined data type introduces a violation of the MISRA C specification. Specifically, if `char` (`char_T`) and either `int8` (`int8_T`) or `uint8` (`uint8_T`) are mapped to the same user replacement type, the result is a MISRA C violation. If you try to generate C++ code, invalid implicit type casts are made and compile-time errors can result. Use `char` only in one-to-one data type replacements.
- For ERT S-functions, replace the `boolean` data type with only an 8-bit integer, `int8` or `uint8`.

Data Definition and Declaration Management

In this section...

“Overview of Data Objects” on page 7-43

“Create Simulink and mpt Data Objects” on page 7-44

“Create Data Objects for a Model” on page 7-57

“Define Global Data Objects in Separate File” on page 7-63

“Define Global Data Objects in Separate Files” on page 7-64

“Save and Load Data Objects” on page 7-65

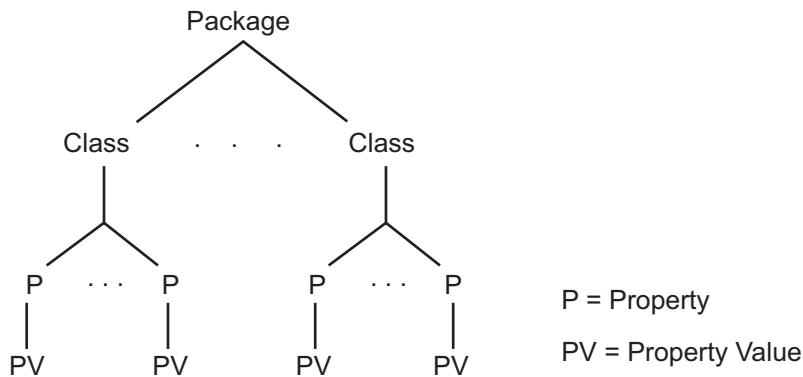
“Apply Naming Rules to Identifiers Globally” on page 7-65

Overview of Data Objects

Data objects include the parameters and signals that the source code uses, and a description of their properties. Data objects appear in the middle pane of the Model Explorer. They also appear in the MATLAB workspace. You can control the property values for each data object, thereby determining how each parameter and signal is defined and declared in generated code.

The values of data object properties are used to determine where the code generator places a parameter or signal in the generated file. This is because some property values are associated with different template symbols. The location of a symbol in a template determines where the associated parameter or signal is located in the generated file. For details about templates and symbols, see “Customize Code Organization and Format”.

Simulink uses a hierarchy of terms that are drawn from object-oriented programming. For details, see “Data Objects” in the Simulink documentation. The sketch below summarizes this hierarchy.



Simulink or `mpt` is the package. Parameter and Signal are two classes in each of these packages. Each class has a number of properties associated with it. Sometimes properties are called *attributes*. Data objects (the parameters and signals) are the instances of a `package.class`. Parameter data objects have a set of properties. Signal data objects have a different set of properties than that for parameters. For each data object, each property in the set has its own property *value*.

Create Simulink and `mpt` Data Objects

- “Ways to Create Data Objects” on page 7-44
- “Create Data Objects with Data Object Wizard” on page 7-45
- “Create `mpt` Data Objects with Data Object Wizard” on page 7-51
- “Simulink and `mpt` Data Object Comparison” on page 7-51
- “Create Data Objects from External Data Source” on page 7-55

Ways to Create Data Objects

The Embedded Coder software provides the `mpt` (module packaging tool) data object, which contains the properties of Simulink data objects plus properties that provide additional control over module packaging. For a comparison of the properties of Simulink and `mpt` data objects, see “Simulink and `mpt` Data Object Comparison” on page 7-51.

There are different ways of creating Simulink and `mpt` data objects.

- Using the MATLAB command line.

- Using the Model Explorer.

For Simulink data objects, select **Add > Simulink Parameter** or **Add > Simulink Signal**.

For mpt data objects, select **Add > Add Custom**.

For more information, see “Data Objects” in the Simulink documentation.

- By invoking the Data Object Wizard for an existing model. For more information and examples, see “Data Object Wizard” in the Simulink documentation and “Create mpt Data Objects with Data Object Wizard” on page 7-51.
- Creating data objects based on an external source. You can do this manually item by item, or together using a script. For more information, see “Create Data Objects from External Data Source” on page 7-55.

The following sections illustrate how to create Simulink and mpt data objects and compares their properties as data types.

Create Data Objects with Data Object Wizard

You can use Data Object Wizard to create data objects for your model (see “Data Object Wizard” in the Simulink documentation).

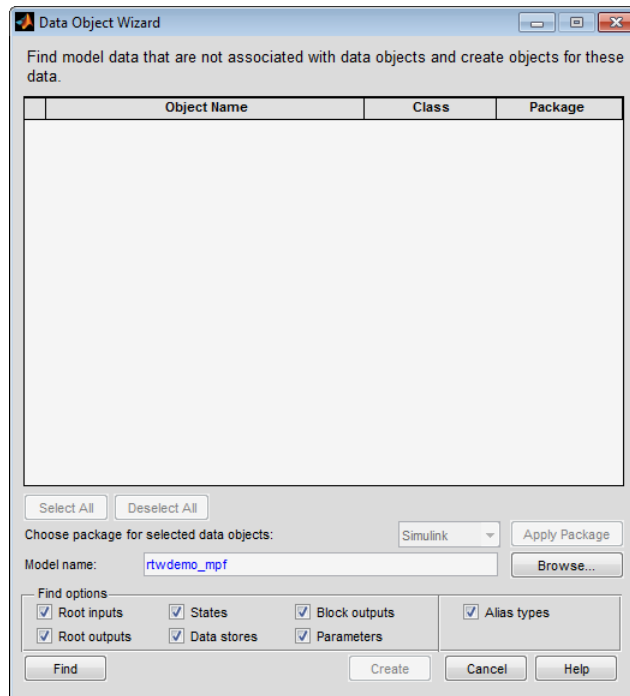
Data Object Wizard is especially useful for creating multiple data objects for

- Existing models that do not currently use data objects.
- Existing models to which you have added signals or parameters and therefore you need to create more data objects.

Create Data Objects

This procedure creates Simulink data objects using Data Object Wizard.

- 1 Open the model for which you want to create data objects. For example, open `rtwdemo_mpf` (which is located in `toolbox/rtw/rtwdemos`).
- 2 Open Data Object Wizard by entering `dataobjectwizard` at the MATLAB command line or by selecting **Data Object Wizard** from the **Code > Data Objects** menu in the Simulink editor. The Data Object Wizard dialog box appears:

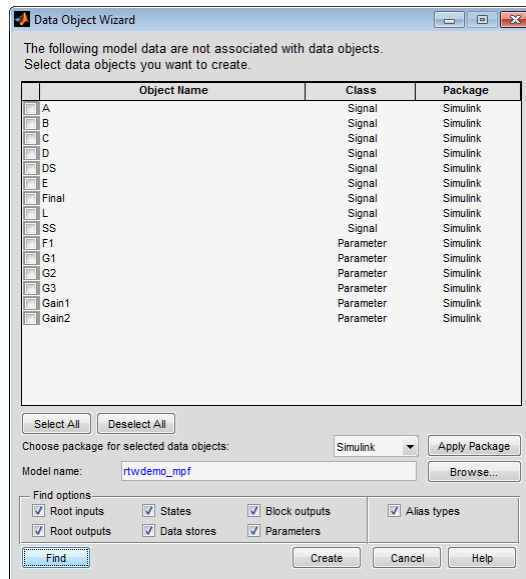


The **Model name** field displays the name of the model. You could specify a different model by editing the field or by selecting the model using the adjacent **Browse** button. When the **Model name** field is nonempty, the **Find** button is enabled.

- 3 In the **Find options** pane, select the desired check boxes. For descriptions of each check box, see “Data Object Wizard” in the Simulink documentation. Be sure to check the **Alias types** option. This finds user-registered data types in the `sl_customization.m` file plus data type replacements specified for the model in the **Data Type Replacement** pane of the Configuration Parameters dialog box. The Data Object Wizard can create `Simulink.AliasType` objects from these.
- 4 Click the **Find** button. The model's potential data objects appear. This includes the model's signals (root inputs, root outputs, and block outputs), discrete states, data stores, and parameters, depending on:
 - The check boxes you selected in the previous step
 - The constraint mentioned in the note above

Data Object Wizard finds only those signals, parameters, data stores, and states whose storage class is set to **Auto**. The Wizard lists each data store and discrete state that it finds as a signal class.

- 5 Click **Select All**. Notice in the **Choose package for selected data objects** field that **Simulink**, the default, is selected. Therefore, the data objects are associated with the **Simulink** package, as shown below.



- 6 Click **Create**. The data objects are added to the MATLAB workspace, and they disappear from Data Object Wizard.
- 7 Click **Cancel**. The Data Object Wizard dialog box disappears.

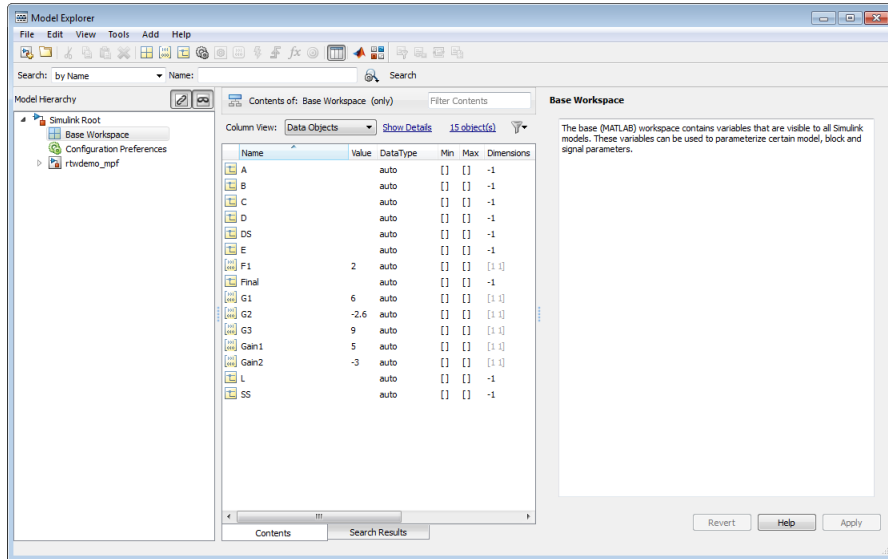
Now you can set property values for the data objects.

Set Property Values for Data Objects

Most of the property values of data objects are supplied by defaults. A few are from the model. Note that for Simulink data objects, the default storage class is **Auto**.

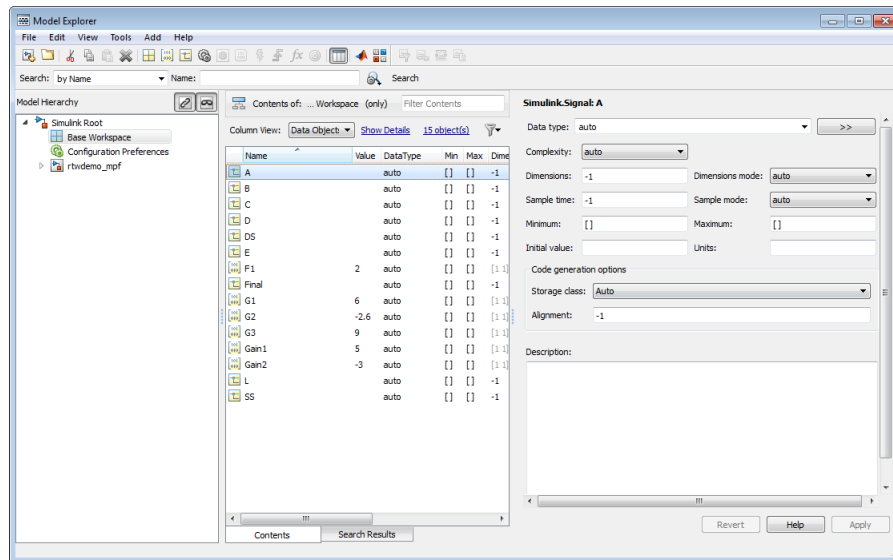
- 1 Type `daexplr` on the MATLAB command line, and press **Enter**. The Model Explorer appears.

- 2 In the Model Hierarchy (left) pane, select **Base Workspace**. The Simulink data objects appear in the **Contents** (middle) pane, as shown below.

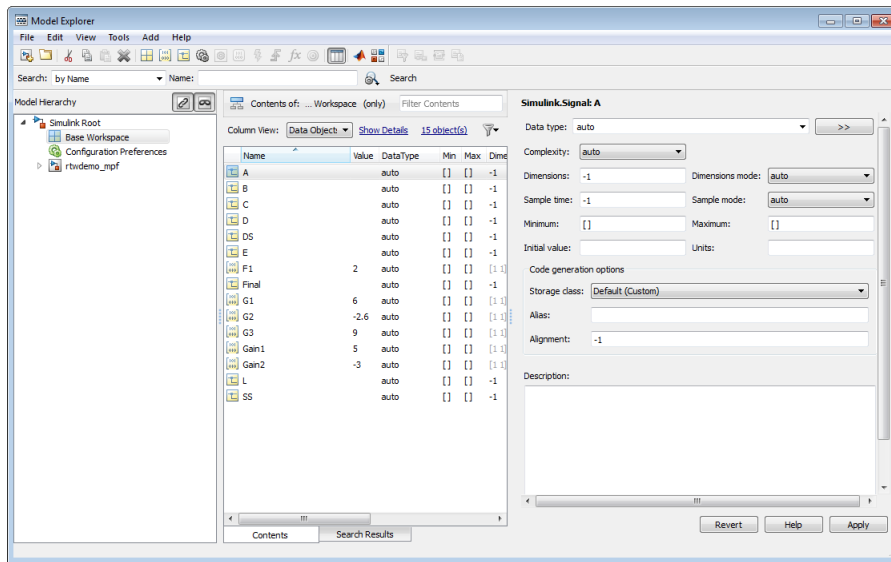


If the objects that you see do not appear in the order shown, click the **Name** column header in the middle pane to sort the objects in ascending order by name.

- 3 To see the properties of a Simulink data object, select a data object in the middle pane. The right pane displays the property names for that object. These property names also appear as column headings in the middle pane. For example, if you select signal data object A in the middle pane, the Model Explorer looks like this:



- 4 You can change the values specified for the properties of the selected object. For example, with A selected, change its **StorageClass** property from **Auto** to **Default (Custom)**, then click **Apply**. The property changes as shown below:



You can use Control-Right-Click to select multiple objects in the center pane, then edit a property value. The wizard applies the new value to the selected objects. For descriptions of object properties and their values, see “Parameter and Signal Property Values”.

Generate and Inspect Code

Data objects for the model have been created. You have specified property values for each data object's properties. Now you generate and inspect the source code, to see if it needs correction or modification. If it does, you can change property values and regenerate the code until it is what you want.

- 1 In the Configuration Parameters dialog box, click **Code Generation** in the left pane.
- 2 In the **Report** pane, select the **Create code generation report** check box.

Note When you select the **Create code generation report** check box, the code generation software automatically selects two check boxes on the pane: **Launch report automatically** and **Code-to-model**. For large models, you may find that HTML report generation (step 4 below) takes longer than you want. In this case, consider clearing the **Code-to-model** check box (and the **Model-to-code** check box if selected). The report will be generated faster.

- 3 In the **Code Generation** pane, select the **Generate code only** check box. The **Build** button changes to **Generate Code**.

Note The generate code process generates the `.c` / `.cpp` and `.h` files. The build process adds compiling and linking to generate the executable. For details on build, see “How Executable Programs Are Built From Models” in the Simulink Coder documentation.

- 4 Click the **Generate Code** button. After a moment, the HTML code generation report appears, listing the generated files on the left pane.
- 5 Select and review files in the HTML code generation report. See “Traceability in Code Generation Report” for more information.

Create `mpt` Data Objects with Data Object Wizard

Create `mpt` data objects using Data Object Wizard the same way you did for Simulink data objects, as explained in “Create Data Objects with Data Object Wizard” on page 7-45, except select `mpt` as the package instead of `Simulink`.

Set the property values for the `mpt` data objects the same way you set them for Simulink data objects, as explained in “Set Property Values for Data Objects” on page 7-47, with the following exceptions:

- Accept the default custom storage class for `mpt` data objects, `Global (Custom)`
- For data objects `A` and `F1`, type `mydefinitionfile` in the **Definition file** field on the Model Explorer.

Then generate and inspect the code.

Note The **Alias** field is related to “Apply Naming Rules to Identifiers Globally” on page 7-65.

Simulink and `mpt` Data Object Comparison

The `mpt` data object contains the properties of Simulink data objects plus properties that provide additional control over module packaging. The differences between Simulink and `mpt` data objects can be illustrated by comparing

- “Signal and Parameter Properties” on page 7-52

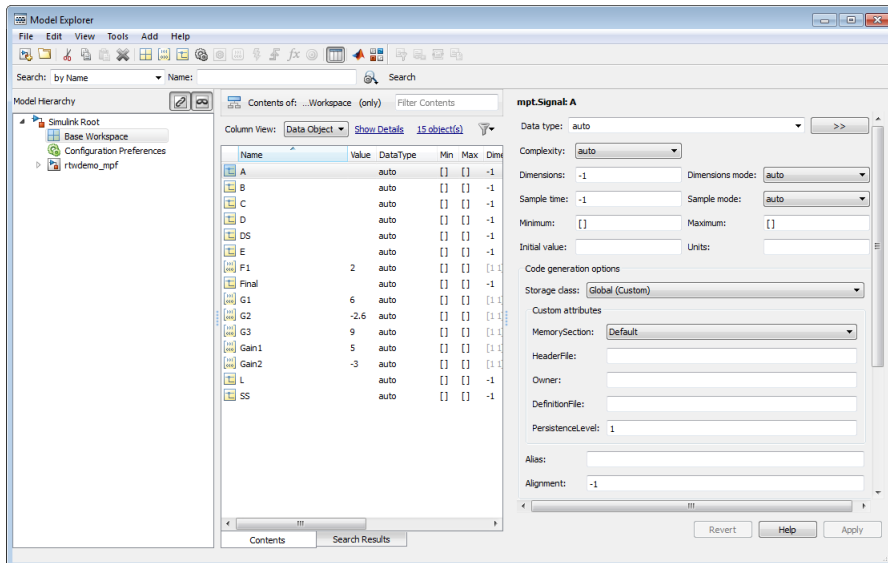
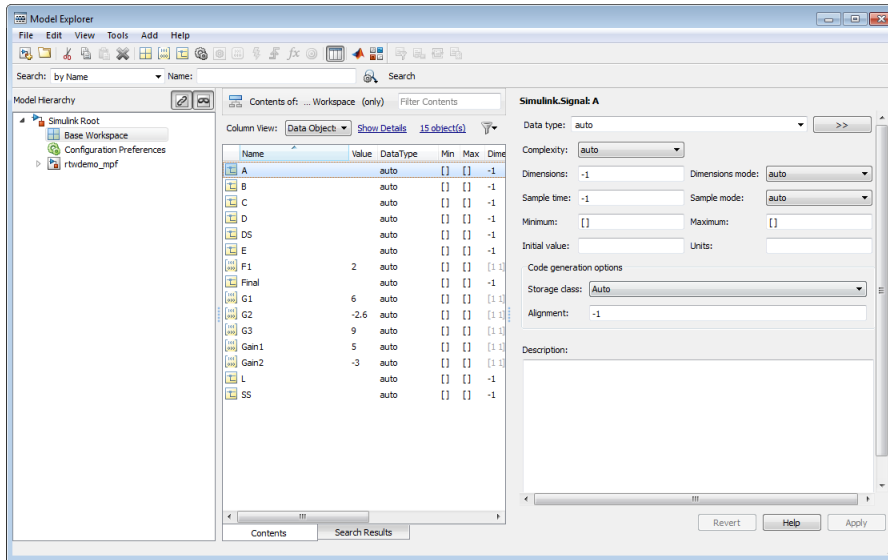
- “Configuration Parameters” on page 7-53
- “Generated Code” on page 7-54

Key differences include the following:

- Different custom storage classes displayed in the Model Explorer for `mpt` data objects provide more control over the appearance of the generated code.
- Additional custom attributes (owner, definition file, persistence level, memory section) for `mpt` data objects provide more control over data packaging in the generated code.
- On the **Comments** pane of the Configuration Parameters dialog box, the **Custom comments (MPT objects only)** option allows you to add a comment just above a signal or parameter's identifier in the generated code.
- On the **Code Placement** pane of the Configuration Parameters dialog box, in the **Global data placement (MPT data objects only)** subpane:
 - The **Signal display level** parameter allows you to specify whether or not the code generator declares a signal data object as global data
 - The **Parameter tune level** parameter allows you to specify whether or not the code generator declares a parameter data object as tunable global data

Signal and Parameter Properties

The properties that appear in Model Explorer when `mpt` is the package include the properties that appear when `Simulink` is the package plus additional properties. Notice this by comparing the next two figures. (For descriptions of properties in Model Explorer, see “Parameter and Signal Property Values”.)



Configuration Parameters

The following configuration parameters relate to module packaging features. These parameters are available in the Configuration Parameters dialog box and Model Explorer

when the system target file selected for a Simulink model is `ert.tlc` (or a system target file derived from an `ert.tlc`):

- **Custom comments (MPT objects only)** option on the **Code Generation > Comments** pane
- In the **Global data placement (MPT data objects only)** subpane on the **Code Generation > Code Placement** pane:
 - **Signal display level** parameter
 - **Parameter tune level** parameter

Generated Code

In the example used in “Set Property Values for Data Objects” on page 7-47, you selected **Default (Custom)** in the **Storage class** field for signal A and parameter F1. You selected the default **Auto** in the **Storage class** field for the remaining data objects. But for the **mpt** data objects you used the default **Global (Custom)** in the **Storage class** field. When you generated code, these selections resulted in the definitions and declarations shown in the table below.

Simulink Data Object with Auto Storage Class	Simulink Data Object with Default (Custom) Storage Class	mpt Data Object with Global (Custom) Storage Class and Definition File Named mydefinitionfile
<pre>In rtwdemo_mpf.c: /* For signal A */ ExternalInputs rtU; /* For parameter F1 */ if(rtU.A * 2.0 > 10.0) {... In rtwdemo_mpf.h: /* For signal A */ typedef struct { real_T A; } ExternalInputs; extern ExternalInputs rtU;</pre>	<pre>In global.c: real_T A; real_T F1 = 2.0; In global.h: extern real_T A; extern real_T F1;</pre>	<pre>In mydefinitionfile.c: real_T A; real_T F1 = 2.0; In global.h: extern real_T A; extern real_T F1;</pre>

The results shown in the second and third columns of the preceding table require the following configuration parameter adjustments on the **Code Generation > Code Placement** pane:

- Set **Data definition** to Data defined in single separate source file.
- Set **Data definition filename** to `global.c`
- Set **Data declaration** to Data declared in single separate source file.
- Set **Data definition filename** to `global.h`

See the left column of the table, which shows generated code for Simulink signal and parameter data objects, whose **Storage class** field is `Auto`. The input `A` is defined as part of the structure `rtU` as shown above. In the case of the Simulink parameter data object `F1`, since the **StorageClass** was set to `auto`, the code generator chose to include the literal value of `F1` in the generated code. `F1` is a constant in the Stateflow diagram whose value is initialized as `2.0`:

```
if(rtU.A * 2.0 > 10.0) { ...
```

For more details, see “Introduction to Custom Storage Classes” and “Summary of Signal Storage Class Options” in the Simulink Coder documentation.

See the middle column of the table. The Simulink data objects whose **Storage class** is not `Auto` are defined in a definition statement in the global source file (`global.c`) and declared in a declaration statement in the global header file (`global.h`).

In the right column, Simulink data objects whose **Storage class** is not `Auto` are defined in `mydefinitionfile`, as you specified. The declarations for those objects are in the global header file.

Create Data Objects from External Data Source

This procedure creates data objects based on an external data source (such as a Microsoft® Excel® file). You can do this manually or automatically.

Create Data Objects Manually

You can create data objects (and their properties) one-by-one, based on an external data source, as follows:

- 1 Open the external file that contains the data (such as a spreadsheet or database file).
- 2 Determine the data in this file that correspond to the parameters and signals in the model. Parameters in the external source belong to the Simulink parameter class and signals belong to the Simulink signal class.

- 3 On the MATLAB command line, type `daexplr` and press **Enter**. The Model Explorer appears.
- 4 On the **Model Hierarchy** (left) pane, expand **Simulink Root**, and select **Base Workspace**.
- 5 On the **Add** menu, select **Add Custom** for an `mpt` data object or **Simulink Parameter** for a Simulink data object. The default name `Param` appears in the **Contents of** (middle) pane.
- 6 Double-click `Param` and rename this data object as desired.
- 7 Repeat steps 5 and 6 for each additional data item in the external file that belongs to the `mpt.Parameter` class or `Simulink.Parameter` class.

Now you will add data items in the external file that belong to the `mpt.Signal` class or `Simulink.Signal` class.

- 8 On the **Add** menu, select **Add Custom** to add an `mpt` data object or **Simulink Signal** to add a Simulink data object. The default name `Sig` appears in the **Contents of** pane.
- 9 Double-click `Sig` and rename the data object as desired.
- 10 Repeat steps 8 and 9 for each additional data item in the external file that belongs to the `mpt.Signal` class or `Simulink.Signal` class.

External data items for the `mpt.Parameter` or `Simulink.Parameter` class, and the `mpt.Signal` or `Simulink.Signal` class now appear in the **Contents of** pane and in the MATLAB workspace.

Note The property *values* for these data objects are supplied by default.

Create Data Objects Automatically

You can create data objects (and their properties) based on an external data source by creating and running a `.m` file. This file contains the same MATLAB commands you could use for creating data objects one-by-one on the command line, as explained in “Data Objects” in the Simulink documentation. But instead of using the command line, you place the MATLAB commands in the `.m` file for the desired data in the external file:

- 1 Create a new `.m` file.
- 2 Place information in the file that describes the data in the external file that you want to be data objects. For example, the following information creates two `mpt` data

objects with the indicated properties. The first is for a parameter and the second is for a signal:

```
% Parameters
mptParCon = mpt.Parameter;
mptParCon.CoderInfo.CustomStorageClass = 'Const';
mptParCon.value = 3;
% Signals
mptSigGlb = mpt.Signal;
mptSigGlb.DataType = 'int8';
```

- 3 Run the `.m` file. The data objects appear in the MATLAB workspace.

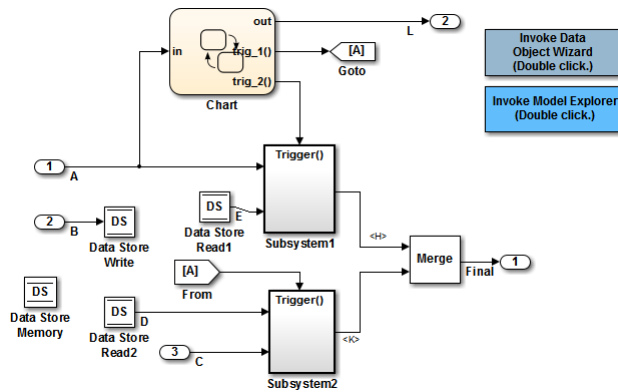
Note: If you want to import data from an external data source, you can write functions that read the information, convert these to data objects, and load them into the MATLAB workspace. Among available MATLAB functions that you can use for this process are `xmlread`, `xmlwrite`, `xlsread`, `xlswrite`, `csvread`, `csvwrite`, `dlmread`, and `dlmwrite`.

Create Data Objects for a Model

In this procedure, you create data objects for a model using the Data Object Wizard, inspect the data objects, and generate code. Definitions for the data objects are generated in the model source file (`model.c`).

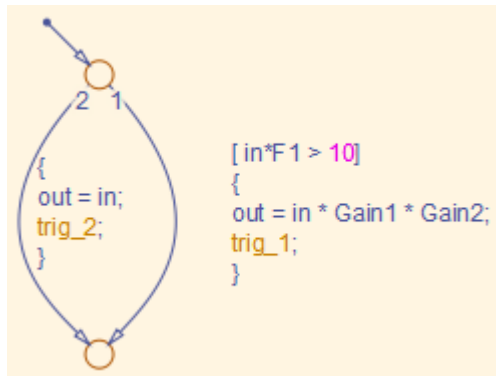
Use Data Object Wizard

- 1 Open the model `rtwdemo_mpf` by clicking the link or by typing `rtwdemo_mpf` in the MATLAB Command Window.

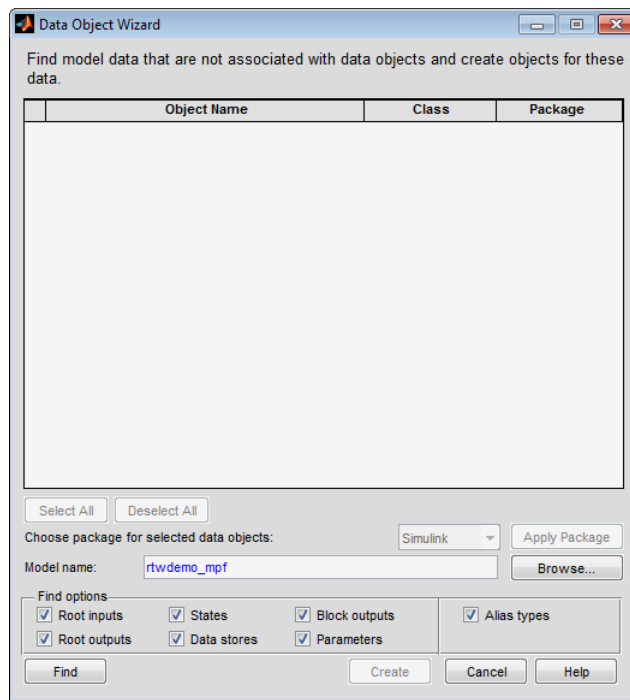


In this model,

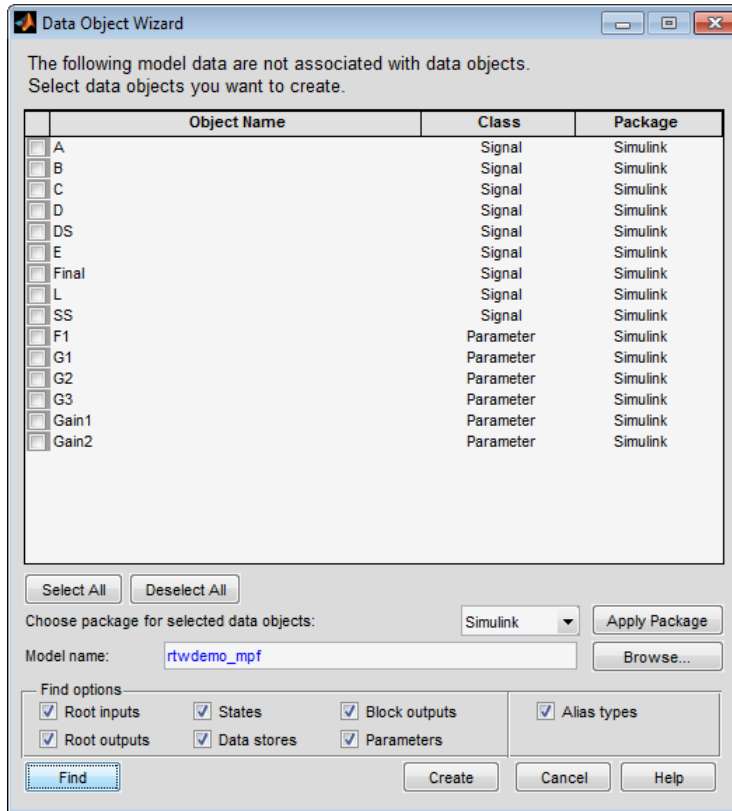
- A, B, and C are input signals, and L and Final are output signals.
 - Subsystem1 receives inputs A and E and contains constants G1 and G2. Signal E is an output from Data Store Read1.
 - Subsystem2 receives inputs C and D. Signal D is an output from Data Store Read2. There is a constant in Subsystem2 named G3. Also, this subsystem has a Unit Delay block whose state name is SS.
- 2 Double-click the Stateflow chart and notice it has constants F1, Gain1, and Gain2, as shown below:



- 3 Change to a work folder that is not on an installation path and save the model in that work folder. The code generation software does not allow you to generate code from an installation folder.
- 4 Double-click the **Invoke Data Object Wizard** button on the model. Or, type `dataobjectwizard('rtwdemo_mpf')` in the MATLAB Command Window. Data Object Wizard opens and `rtwdemo_mpf` appears in the **Model name** field, as shown below.



- 5 Click **Find** on Data Object Wizard. After a moment, the model's parameters and signals appear in Data Object Wizard.
- 6 Click **Select All**.
- 7 In the **Choose package for selected objects** field, select `mpt`. For an explanation of “package,” see “Overview of Data Objects” on page 7-43.
- 8 Click **Apply Package**. Data Object Wizard associates the selected data objects with the `mpt` package, as shown below.



- 9 Click **Create**. Data Object Wizard creates data objects for the selected parameters and signals. Data Object Wizard removes the objects from its object view. Also, the objects are added to the MATLAB workspace, as shown below.

The screenshot shows a 'Workspace' window with a table of data objects. The table has four columns: Name, Value, Min, and Max. The 'Name' column lists various objects like A, B, C, D, DS, E, F1, Final, G1, G2, G3, Gain1, Gain2, L, SS, and ans. The 'Value' column shows their respective values, such as '<1x1 mpt.Signal>' for signals and '<1x1 mpt.Parameter>' for parameters. The 'Min' and 'Max' columns are currently empty.

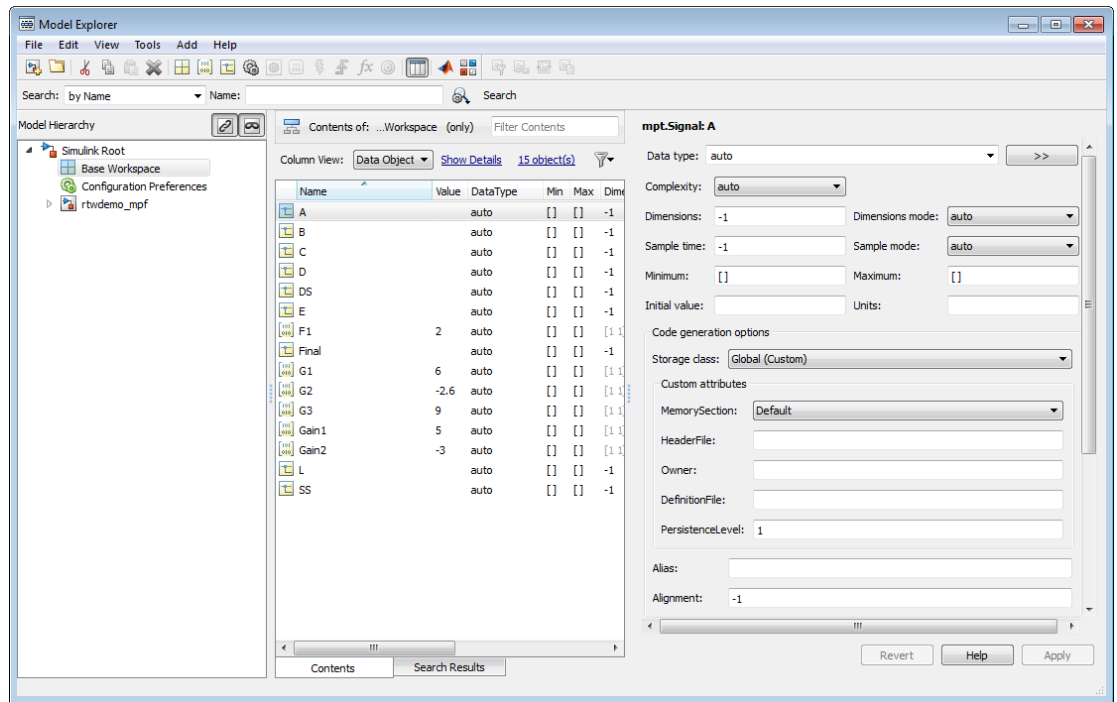
Name	Value	Min	Max
A	<1x1 mpt.Signal>		
B	<1x1 mpt.Signal>		
C	<1x1 mpt.Signal>		
D	<1x1 mpt.Signal>		
DS	<1x1 mpt.Signal>		
E	<1x1 mpt.Signal>		
F1	<1x1 mpt.Parameter>		
Final	<1x1 mpt.Signal>		
G1	<1x1 mpt.Parameter>		
G2	<1x1 mpt.Parameter>		
G3	<1x1 mpt.Parameter>		
Gain1	<1x1 mpt.Parameter>		
Gain2	<1x1 mpt.Parameter>		
L	<1x1 mpt.Signal>		
SS	<1x1 mpt.Signal>		
ans	'c:\work6'		

10 Close Data Object Wizard.

Inspect Data Objects

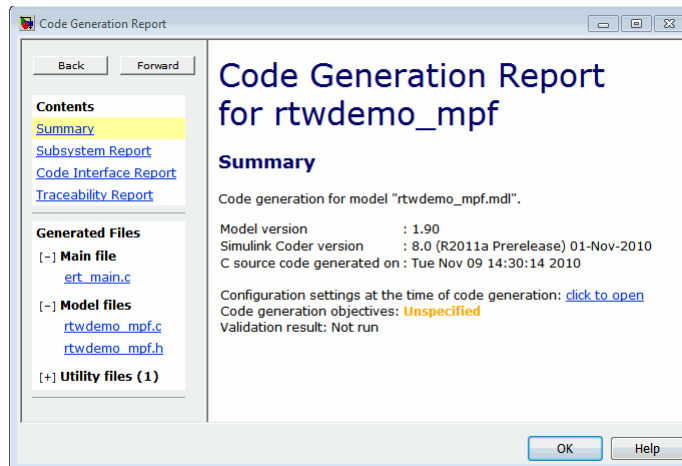
You can inspect each data object you selected in the Data Object Wizard using the Model Explorer:

- 1 If you have not already done so, complete the steps in “Use Data Object Wizard” on page 7-57 .
- 2 Open the Model Explorer.
- 3 In the left pane, select **Base Workspace**. Notice that data objects appear in the middle pane.
- 4 In the middle pane, select data objects one at a time, and notice their property values in the right pane. The figure below shows this for signal A. The data objects have default property values. Note that for an **mpt** data object, the default in the **Storage class** field is **Global (Custom)**. For descriptions of the properties on the Model Explorer, see “Set Property Values for Data Objects” on page 7-47.



Generate and Inspect Code

- 1 If you have not already done so, complete the steps in “Use Data Object Wizard” on page 7-57 and “Inspect Data Objects” on page 7-61.
- 2 In the left pane of the Model Explorer, expand the **rtwdemo_mpf** node.
- 3 In the left pane, click **Configuration (Active)**.
- 4 In the center pane, click **Code Generation**. The active configuration parameters appear in the right pane.
- 5 In the **Report** tab, select **Create code generation report**
- 6 Select the **General** tab. Select **Generate code only**, and then click **Generate Code**. After a few moments, the names of the generated files are listed on the Code Generation Report on the left pane.



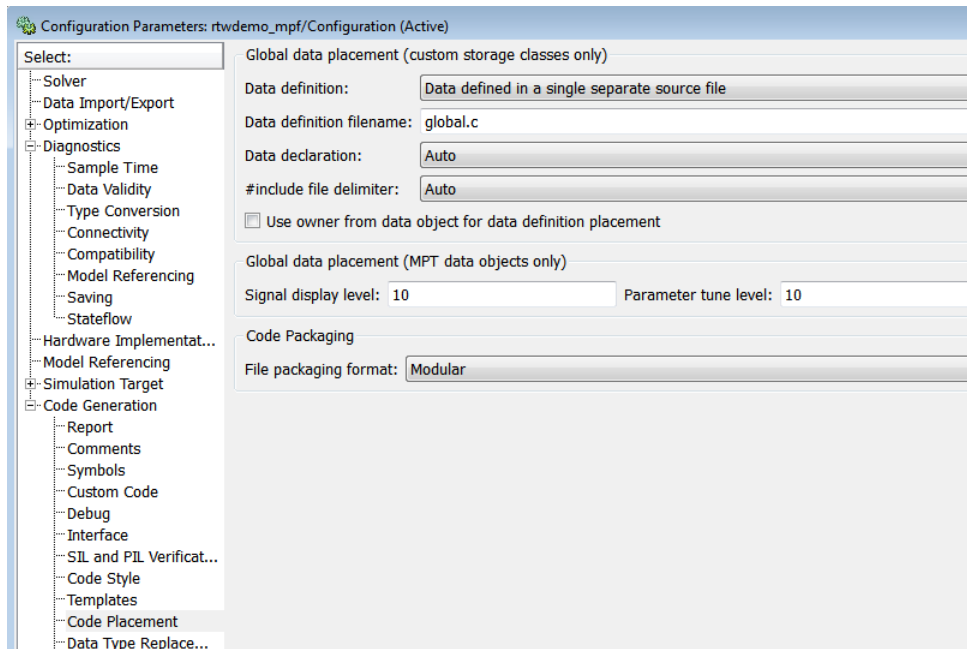
- 7 Open and inspect the content of the model source file `rtwdemo_mpf.c`. The following data objects are initialized in this file.

```
real_T F1 = 2.0;
real_T G1 = 6.0;
real_T G2 = -2.6;
real_T G3 = 9.0;
real_T Gain1 = 5.0;
real_T Gain2 = -3.0;
```

Define Global Data Objects in Separate File

The previous procedure placed the model's data objects in the model source file. Now you place global data objects in a file separate from the model source file:

- 1 Configure the model's generated code to include Simulink data objects (signal and parameter) in a separate definition file. Set **Diagnostics > Data Validity > Signal resolution** to `Explicit` and `implicit`.
- 2 Specify that data be defined in a separate file. Set **Code Generation > Code Placement > Data definition** to `Data defined in single separate source file`. Accept the default for **Data definition filename**, `global.c`



- 3 Specify that data be declared in a separate file. Set **Data declaration** to **Data declared in a single separate header file** and accept the default for **Data declaration filename**, `global.h`. Then, click **Apply**.
- 4 Click **Generate Code**. Notice that the code generation report lists `global.c` and `global.h` files.
- 5 Inspect the code generation report. Notice that
 - The data objects formerly initialized in `rtwdemo_mpf.c` now are initialized in `global.c`.
 - The file `rtwdemo_mpf.c` includes `rtwdemo_mpf.h`.
 - The file `rtwdemo_mpf.h` includes `global.h`.

Define Global Data Objects in Separate Files

The previous procedure placed global data objects in a separate definition file. You named that file `global.c`. (You named the corresponding declaration file `global.h`.) MPF allows you to override this and place a specific data object in its own definition file.

In the following procedure, you move the `Final` signal to a file called `finalsig.c`, and keep the other data objects defined in `global.c`:

- 1 In the Model Explorer, display the base workspace and select the `Final` signal object. The **mpt.Signal** properties appear in the right pane.
- 2 In the **Code generation options** section, type `finalsig.h` in the **Header file** text box, type `finalsig.c` in the **Definition file** text box, and click **Apply**.
- 3 On the **Code Generation > General** pane, click **Generate Code**. The code generation report still lists `global.c` and `global.h`, but adds `finalsig.c` and `finalsig.h`.
- 4 Open the files to inspect them. Notice that the `Final` signal is defined in `finalsig.c`. Other data objects are defined in `global.c`.

Save and Load Data Objects

In a `.mat` file, you can save the set of data objects (and their properties) that you have created and load this information for later use or exchange it with another user. You can save data objects in the workspace.

Apply Naming Rules to Identifiers Globally

- “Overview” on page 7-65
- “Change Names of Identifiers” on page 7-66
- “Specify Data Object Naming Rules” on page 7-68
- “Define Rules That Change Signal Names” on page 7-69
- “Define Rules That Change Parameter Names” on page 7-69
- “Define Rules That Change #defines” on page 7-70

Note The capabilities described in this section apply only to mpt data objects.

Overview

Signal and parameter names appear on a Simulink model. The same names appear as data objects on the Model Explorer. By default, these names are replicated exactly in the generated code. For example, “`Speed`” on the model (and workspace) appears as the identifier “`Speed`” in the code, by default. But you can change how they appear in

the code. For example, if desired, you can change "Speed" to SPEED or speed. Or, you can choose to use a different name altogether in the generated code, like MPH. The only restriction is that you follow ANSI® C³/C++ rules for naming identifiers.

There are two ways of changing how a signal name or parameter name is represented in the generated code. You can do this *globally*, by following the procedure in this section. This procedure makes selections on the Configuration Parameters dialog box to change the names when code generation occurs, according to the same rule. Or, you can change the names *individually* by following the steps described in "Set Property Values for Data Objects" on page 7-47. The relevant field in that procedure is **Alias** on the Model Explorer.

If the **Alias** field is empty, the naming rule that you select on the Configuration Parameters dialog box applies across data objects. But if you do specify a name in the **Alias** field, this overrides the naming rule for that data object. The table below illustrates these cases. The table assumes that you selected **Force lower case** as the naming rule. But with the information provided, you can determine how the naming rules work for an **mpt** data object or a Simulink data object (**Force upper case**, **Force lower case**, or **Custom M-function**).

Naming Rules and Alias Override (Global Change of Force Lower Case Rule)

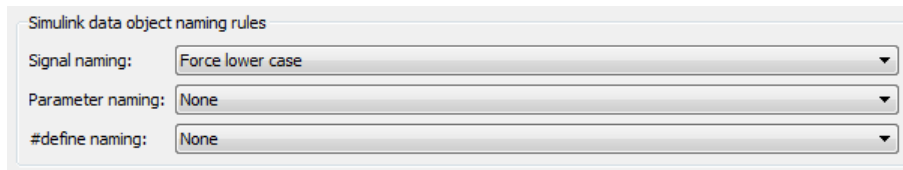
Name of Data Object in Model	Name in Alias Field	Package	Result in Generated Code
A		Simulink or mpt	a
A	D	Simulink or mpt	D

Change Names of Identifiers

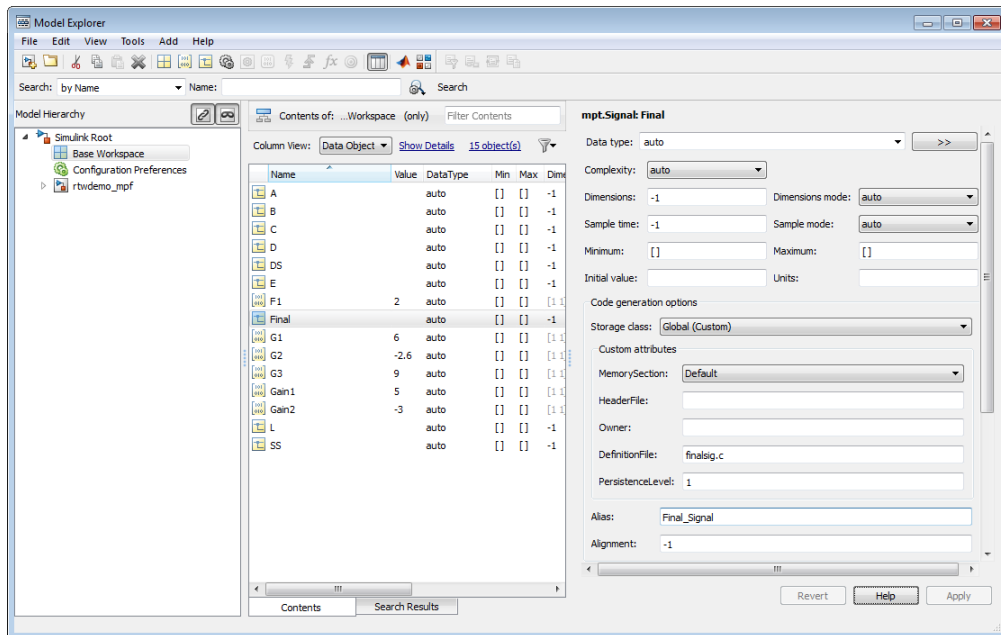
This procedure changes the names of signal identifiers so that they are spelled with lowercase letters. A in the definition statement located in `global.c` is changed to `a`. The one exception is the `Final` signal in the `finalsig.c` file. You change this identifier name to `Final_Signal`. The names of the rest of the identifiers in the generated files remain the same:

- 1 Open the **Code Generation > Symbols** pane of the Configuration Parameters dialog.
- 2 In the **Simulink data object naming rules** section, set **Signal naming** to **Force lower case**, and click **Apply**.

3. ANSI is a registered trademark of the American National Standards Institute, Inc.



- 3 Display the base workspace and select **Final**.
- 4 In the right pane, type **Final_Signal** in the **Alias** text box, then click **Apply**.



- 5 On the **Code Generation > General** pane, click **Generate Code**. Now, the signal identifiers in `global.c` and `global.h` appear with lowercase letters.

```
real_T F1 = 0.0;
real_T G1 = 1.0;
real_T G2 = 1.0;
real_T G3 = 1.0;
real_T Gain1 = 0.0;
real_T Gain2 = 0.0;
real_T a;
real_T b;
```

```
real_T c;  
real_T d;  
real_T ds;  
real_T e;  
real_T l;  
real_T ss;
```

The statement defining the `Final` signal in `finalsig.c` looks like this:

```
real T Final_Signal;
```

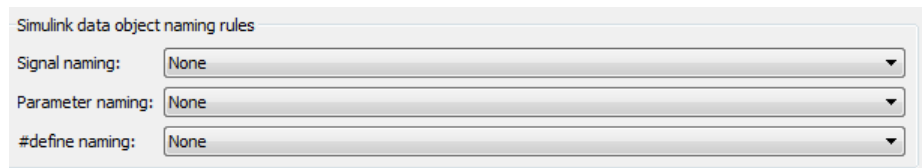
The statement declaring this identifier in `finalsig.h` looks like this:

```
extern real_T Final_Signal;
```

Specify Data Object Naming Rules

You specify Simulink data object naming rules on the **Code Generation > Symbols** pane of the Configuration Parameters dialog box. To access that pane,

- 1 Open an ERT-based model.
- 2 Open the Configuration Parameters dialog box from the **Simulation** menu or Model Explorer.
- 3 Open the **Code Generation > Symbols** pane. See the subpane **Simulink data object naming rules**.



Notice the preconfigured settings on this pane. “Code Generation Pane: Symbols” in the Simulink Coder documentation describes the fields on this pane and their possible settings for these procedures.

- “Define Rules That Change Signal Names” on page 7-69
- “Define Rules That Change Parameter Names” on page 7-69
- “Define Rules That Change #defines” on page 7-70

Define Rules That Change Signal Names

This procedure allows you to change a model's signal names, using the same rule. The new names will appear as identifiers in the generated code:

- 1 In the **Signal naming** menu, click the desired selection. (“Signal naming” explains the possible selections.) The default is **None**. If you selected **Custom M-function**, go to the next step. Otherwise, click **Apply** and then generate and inspect code.
- 2 Write a MATLAB function that changes occurrences of signal names in the model to appear the way you want as identifiers in the generated code. (An example is shown in “Define Rules That Change Parameter Names” on page 7-69.)
- 3 Save the function as a `.m` file in a folder on the MATLAB path.
- 4 In the **M-function** field under **Signal naming**, type the name of the file you saved in the previous step.
- 5 Click **Apply** and then generate and inspect code.

Define Rules That Change Parameter Names

This procedure allows you to change a model's parameter names, using the same rule. The new names will appear as identifiers in the generated code:

- 1 In the **Parameter naming** field, click the desired selection. (“Parameter naming” explains the possible selections.) The default is **None**. If you selected **Custom M-function**, go to the next step. Otherwise, click **Apply**, and proceed to “Define Rules That Change Signal Names” on page 7-69.
- 2 Write a MATLAB function that changes occurrences of parameter names in the model to appear the way you want as identifiers in the generated code. For example, the code below changes parameter names to make their first letter uppercase, and their remaining letters lowercase.

```
function
revisedName = initial_caps_only(name, object)
% INITIAL_CAPS_ONLY: User-defined naming rule causing each
% identifier in the generated code to have initial cap(s).
%
% name: name as spelled in model.
% object: the object of name; includes name's properties.
%
% revisedName: manipulated name returned to MPT for the
code.
%
```

```
%  
:  
revisedName = [upper(name(1)),lower(name(2:end))];  
:
```

- 3** Save the function as a `.m` file in a folder on the MATLAB path.
- 4** In the **M-function** field under **Parameter naming**, type the name of the file you saved in the previous step.
- 5** Click **Apply** and then define rules applicable to signal names.

Define Rules That Change #defines

This procedure allows you to change a model's parameter names whose storage class you selected as **Define** in “Create mpt Data Objects with Data Object Wizard” on page 7-51, using the same rule. The new names will appear as identifiers in the generated code:

- 1** In **#define naming**, click the desired selection. (“#define naming” explains the possible selections.) The default is **None**. If you select **Custom M-function**, go to the next step. Otherwise, click **Apply** and proceed to “Define Rules That Change Parameter Names” on page 7-69.
- 2** Write a MATLAB function that changes occurrences of the parameter name whose storage class you specified as **Define** in “Create mpt Data Objects with Data Object Wizard” on page 7-51 so that it appears the way you want as an identifier in the generated code. (An example is shown below.)
- 3** Save the function as a `.m` file in a folder on the MATLAB path.
- 4** In the **M-function** field under **#define naming**, type the name of the file you saved in the previous step.
- 5** Click **Apply** and then define rules that change parameter names.

Module Packaging Tool (MPT) Data Objects

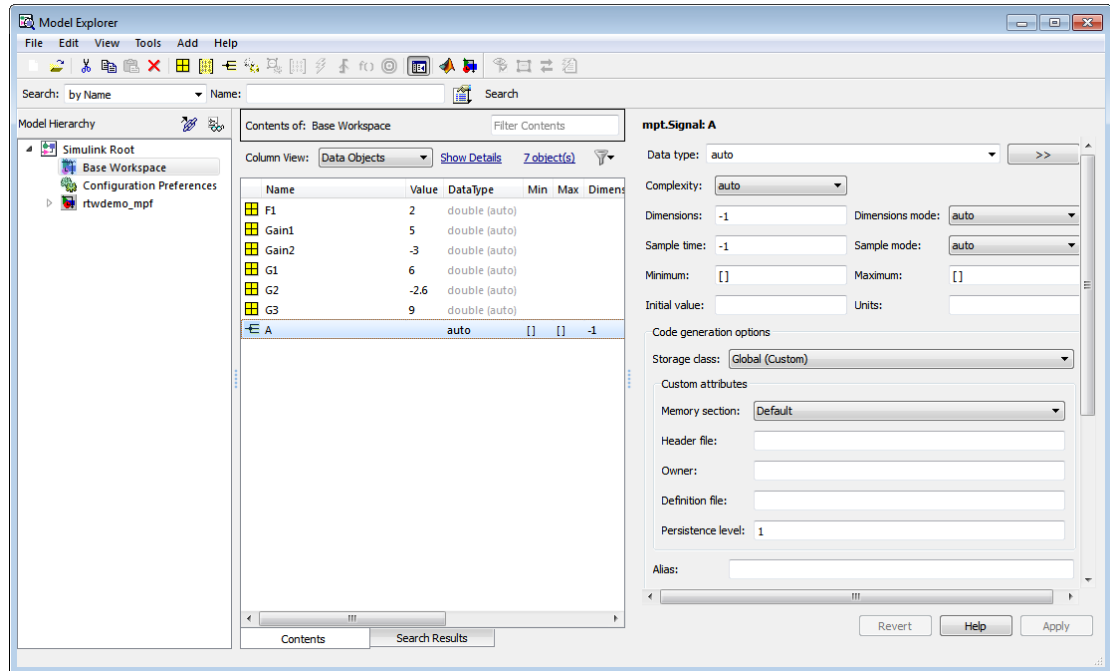
MPT Data Object Properties

The following table describes the properties and property values for `mpt.Parameter` and `mpt.Signal` data objects that appear in the Model Explorer.

Note: You can create `mpt.Signal` and `mpt.Parameter` objects in the base MATLAB or model workspace. However, if you create the object in a model workspace, the object's storage class must be set to `auto`.

The figure below shows an example of the Model Explorer. When you select an `mpt.Parameter` or `mpt.Signal` data object in the middle pane, its properties and property values display in the rightmost pane.

In the Properties column, the table lists the properties in the order in which they appear on the Model Explorer.



Parameter and Signal Property Values

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
Both	User object type	*auto	<p>Prenamed and predefined property sets that are registered in the <code>sl_customization.m</code> file. (See “Register mpt User Object Types”.) This field is active when a user object type is registered.</p> <p>Select <code>auto</code> if this field is available but you do not want to apply the properties of a user object type to a selected data object. The fields on the Model Explorer are populated with default values.</p>
		Listed user object type name	Select a user object type name to apply the properties and values that you associated with this name in the <code>sl_customization.m</code> file. The fields on the Model Explorer are automatically populated with those values.
Parameter	Value	*0	The data type and numeric value of the data object. For example, <code>int8(5)</code> . The numeric value is used as an initial parameter value in the generated code.
Both	Data type		Used to specify the data type for an <code>mpt.Signal</code> data object, but not for an <code>mpt.Parameter</code> data object. The data type for an <code>mpt.Parameter</code> data object is specified in the Value field above. See “Data Types” in the Simulink documentation.
Both	Units	*null	Units of measurement of the signal or parameter. (Enter text in this field.)
Both	Dimensions	* - 1	The dimension of the signal or parameter. For a parameter, the dimension is derived from its value.

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
Both	Complexity	*auto real complex	Complexity specifies whether the signal or parameter is a real or complex number. Select auto for the code generator to decide. For a parameter, the complexity is derived from its value.
Signal	Sample time	* - 1	Model or block execution rate.
Signal	Sample mode	*auto	Determines how the signal propagates through the model. Select auto for the code generator to decide.
		Sample based	The signal propagates through the model one sample at a time.
		Frame based	The signal propagates through the model in batches of samples.
Both	Minimum	*0.0	The minimum value to which the parameter or signal is expected to be bound.
		Number within the minimum range of the parameter or signal. (Based on the data type and resolution of the parameter or signal.)	
Both	Maximum	*0.0	Maximum value to which the parameter or signal is expected to be bound. (Enter information using a dialog box.)
	Code generation options		
	Storage class		Note that an auto selection for a storage class tells the build process to decide how to declare and store the selected parameter or signal.

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
Both	Default (Custom)		Code generation decides how to declare the data object.
Both	Global (Custom)	Global (Custom) is the default storage class for mpt data objects.	Specifies that a code generator not place a qualifier in the data object's declaration.
Both	Memory section	*Default	Memory section allows you to specify storage directives for the data object. Default specifies that the code generator not place a type qualifier and pragma statement with the data object's declaration.
Parameter		MemConst	Places the const type qualifier in the declaration.
Both		MemVolatile	Places the volatile type qualifier in the declaration.
Parameter		MemConstVolatile	Places the const volatile type qualifier in the declaration.
Both	Header file		Name of the file used to import or export the data object. This file contains the declaration (extern) to the data object. Also, you can specify this header filename between the double-quotation or angle-bracket delimiter. You can specify the delimiter with or without the .h extension. For example, specify "object.h" or "object" . For the selected data object, this overrides the general delimiter selection in the #include file delimiter field on the Configuration Parameters dialog box.
Both	Owner	*Blank	The name of the module that owns this signal or parameter. This is used to help

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
			determine the ownership of a definition. For details, see “Ownership Settings” and the table “Ownership Settings”.
Both	Definition file	*Blank	Name of the file that defines the data object.
		Valid string	
Both	Persistence level		The number you specify is relative to Signal display level or Parameter tune level on the Code Placement pane of the Configuration Parameters dialog box. For a signal, allows you to specify whether or not the code generator declares the data object as global data. For a parameter, allows you to specify whether or not the code generator declares the data object as tunable global data. See Signal display level and Parameter tune level in “Code Generation Pane: Code Placement”.
Both	Bitfield (Custom)		Embeds Boolean data in a named bit field.
	Struct name		Name of the struct into which the object's data will be packed.
Parameter	Const (Custom)		Places the const type qualifier in the declaration.
Parameter	Header file		See above.
Parameter	Owner		See above.
Parameter	Definition file		See above.
Parameter	Persistence level		See above.
Both	Volatile (Custom)		Places the volatile type qualifier in the declaration.
Both	Header file		See above.

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
Both	Owner		See above.
Both	Definition file		See above.
Both	Persistence level		See above.
Parameter	ConstVolatile (Custom)		Places the <code>const volatile</code> type qualifier in declaration.
Parameter	Header file		See above.
Parameter	Owner		See above.
Parameter	Definition file		See above.
Parameter	Persistence level		See above.
Parameter	Define (Custom)		Represents parameters with a <code>#define</code> macro.
Parameter	Header file		See above.
Both	ExportToFile (Custom)		Generates global variable definition, and generates a user-specified header (<code>.h</code>) file that contains the declaration (<code>extern</code>) to that variable.
Both	Memory section		See above.
Both	Header file		See above.
Both	Definition file		See above.
Both	ImportFromFile (Custom)		Includes predefined header files containing global variable declarations, and places the <code>#include</code> in a corresponding file. Assumes external code defines (allocates memory) for the global variable.
Both	Data access	*Direct	Allows you to specify whether the identifier that corresponds to the selected data object stores data of a data type

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
			(Direct) or stores the address of the data (a pointer).
Both		Pointer	If you select Pointer , the code generator places * before the identifier in the generated code.
	Header file		See above.
Both	Struct (Custom)		Embeds data in a named struct to encapsulate sets of data.
Both	Struct name		See above.
Signal	GetSet (Custom)		Reads (gets) and writes (sets) data using functions.
Signal	Header file		See above.
Signal	Get function		Specify the Get function.
Signal	Set function		Specify the Set function.
Both	Alias	*null	As explained in detail in “Apply Naming Rules to Identifiers Globally”, for a Simulink or mpt data object (identifier), specifying a name in the Alias field overrides the global naming rule selection you make on the Configuration Parameters dialog box.
		Valid ANSI ^a C/C++ variable name	
Both	Description	*null	Text description of the parameter or signal. Appears as a comment beside the signal or parameter's identifier in the generated code.
		String	
Signal	Reusable (Custom)		Allows the code generator to reuse a pair of root I/O signals when you specify the same name and the same custom storage

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
			class for both. The custom storage class is either Reusable (Custom) or derived from Reusable (Custom) .
Signal	Data Scope	*Auto	You can specify the scope of symbols code generation generates for a data object of this class by selecting a value for DataScope . When you take the default of Auto , code generation determines the symbol scope internally. If possible, symbols have File scope. Otherwise, they have Exported scope.
		File	Code generation defines the scope of each symbol as the file that defines it. File scope requires each symbol to be used in a single file. If the same symbol is referenced in multiple files, code generation reports an error.
		Exported	Code generation exports symbols to external code in the header file specified by the HeaderFile field. If a HeaderFile is not specified, symbols are exported to external code in <i>model.h</i> .
		Imported	Code generation imports symbols from external code in the header file specified by the HeaderFile field. If you do not specify a header file, code generation generates an extern directive in <i>model_private.h</i> .
Signal	Header file		See above.
Signal	Owner		See above.
Signal	Definition file		See above.

a. ANSI is a registered trademark of the American National Standards Institute, Inc.

Examples of Property Value Changes on Generated Code

What I noticed when inspecting the <code>.c/.cpp</code> file	Change I made to property value settings	What I noticed after regenerating and reinspecting the file
<p>Example 1: Parameter data objects can be declared or defined as constants. I know that the data object <code>GAIN</code> is a parameter. I want this to be declared or defined in the <code>.c</code> file as a variable. But I notice that <code>GAIN</code> is declared as a constant by the statement <code>const real_T GAIN = 5.0;</code>. Also, this statement is in the constant section of the file.</p>	<p>In the Model Explorer, I clicked the data object <code>GAIN</code>. I noticed that the property value for its Memory section property is set at <code>MemConst</code>. I changed this to <code>Default</code>.</p>	<p>I notice two differences. One is that now <code>GAIN</code> is declared as a variable with the statement <code>real_T GAIN = 5.0;</code>. The second difference is that the declaration now is located in the <code>MemConst</code> memory section in the <code>.c</code> or <code>.cpp</code> file.</p>
<p>Example 2: I notice again the declaration of <code>GAIN</code> in the <code>.c</code> file mentioned in Example 1. It appears as <code>real_T GAIN = 5.0;</code>. But I have changed my mind. I want data object <code>GAIN</code> to be <code>#define</code>.</p>	<p>I changed the Storage class selection to <code>Define (Custom)</code>.</p>	<p><code>GAIN</code> is not declared in the <code>.c</code> file as a <code>MemConst</code> parameter. Rather, it is defined as a <code>#define</code> macro by the code <code>#define GAIN 5.0</code>, and this is located near the top of the <code>.c</code> file with the other preprocessor directives.</p>
<p>Example 3: I changed my mind again after doing Example 2. I do want <code>GAIN</code> defined using the <code>#define</code> preprocessor directive. But I do not want to include the <code>#define</code> in this file. I know it exists in another file and I want to reference that file.</p>	<p>On the Model Explorer, I notice that the property value for the Header file property is blank. I changed this to <code>filename.h</code>. (I chose the ANSI C/C++ double quote mechanism for the <code>#include</code>, but could have chosen the angle bracket mechanism.) Also, I must make the user-defined <code>filename.h</code> available to the compiler, placing it either in the system path or local directory.</p>	<p><code>#define GAIN 5.0</code> is not present in this <code>.c</code> file. Instead, the <code>#include filename.h</code> code appears as a preprocessor directive at the top of the file.</p>

What I noticed when inspecting the .c/.cpp file	Change I made to property value settings	What I noticed after regenerating and reinspectng the file
<p>Example 4: I have one more change I want to make. Let us say that we have declared the data object <code>data_in</code>, and that its declaration statement in the <code>.c</code> file reads <code>real_T data_in = 0.0;</code>. I want to replace this statement with an alias in the <code>.c</code> file.</p>	<p>In the Model Explorer, I selected the data object <code>data_in</code>. I noticed that the Alias field is blank. I changed this to <code>data_in_alias</code>, which I know is a valid ANSI C/C++ variable name.</p>	<p>The identifier <code>data_in_alias</code> now appears in the <code>.c</code> file everywhere <code>data_in</code> appeared.</p>

Custom Storage Classes

- “Introduction to Custom Storage Classes” on page 9-2
- “Resources for Defining Custom Storage Classes” on page 9-5
- “Simulink Package Custom Storage Classes” on page 9-6
- “Design Custom Storage Classes and Memory Sections” on page 9-8
- “Apply Custom Storage Classes” on page 9-29
- “Generate Code with Custom Storage Classes” on page 9-46
- “Define Advanced Custom Storage Classes Types” on page 9-50
- “GetSet Custom Storage Class” on page 9-53
- “Custom Storage Class Implementation” on page 9-56
- “Custom Storage Class Limitations” on page 9-57

Introduction to Custom Storage Classes

In this section...

“Custom Storage Class Memory Sections” on page 9-3

“Register Custom Storage Classes” on page 9-3

“Custom Storage Class Examples” on page 9-3

During the build process, the *storage class* specification of a signal, tunable parameter, block state, or data object specifies how that entity is declared, stored, and represented in generated code. Note that in the context of the build process, the term “storage class” is not synonymous with the term “storage class specifier”, as used in the C language.

The Simulink Coder software defines four built-in storage classes for use with targets: `Auto`, `ExportedGlobal`, `ImportedExtern`, and `ImportedExternPointer`. These storage classes provide limited control over the form of the code generated for references to the data. For example, data of storage class `Auto` is typically declared and accessed as an element of a structure, while data of storage class `ExportedGlobal` is declared and accessed as unstructured global variables. For information about built-in storage classes, see “Signals” in the Simulink Coder documentation.

If the built-in storage classes do not provide data representation required by your application, you can define *custom storage classes* (CSCs). Embedded Coder (CSCs) extend the built-in storage classes provided by the Simulink Coder software. CSCs can provide application-specific control over the constructs required to represent data in an embedded algorithm. For example, you can use CSCs to:

- Define structures for storage of parameter or signal data.
- Conserve memory by storing Boolean data in bit fields.
- Integrate generated code with legacy software whose interfaces cannot be modified.
- Generate data structures and definitions that comply with your organization's software engineering guidelines for safety-critical code.

Custom storage classes affect only code generated for ERT targets. When **Configuration Parameters > Code Generation > Target Selection > System target file** specifies a GRT target, the names of custom storage classes sometimes appear in dialog boxes, but selecting a CSC is functionally the same as selecting `Auto`. See “Targets and Code Formats” for information about ERT and GRT targets.

Custom Storage Class Memory Sections

Every custom storage class has an associated *memory section* definition. A memory section is a named collection of properties related to placement of an object in memory; for example, in RAM, ROM, or flash memory. Memory section properties let you specify storage directives for data objects. For example, you can specify `const` declarations, or compiler-specific `#pragma` statements for allocation of storage in ROM or flash memory sections.

See “Create and Edit Memory Section Definitions” on page 9-23 for details about using the Custom Storage Class designer to define memory sections. While memory sections are often used with data in custom storage classes, they can also be used with various other constructs. See “Memory Sections” for more information about using memory sections with custom storage classes, and complete information about using memory sections with other constructs.

Register Custom Storage Classes

CSCs are associated with Simulink data class packages (such as the `Simulink` package) and with classes within packages (such as the `Simulink.Parameter` and `Simulink.Signal` classes). The custom storage classes associated with a package are defined by a *CSC registration file*. For example, a CSC registration file exists for the `Simulink` package. This registration file provides predefined CSCs for use with the `Simulink.Signal` and `Simulink.Parameter` classes, and with subclasses derived from these classes. The predefined CSCs are used for a wide variety of applications.

If you use only predefined CSCs, you do not need to be concerned with CSC registration files. You cannot add or change CSCs associated with built-in packages and classes, but you can create your own packages and subclasses, then associate CSCs with those. See “Custom Storage Class Implementation” on page 9-56 for more information.

Custom Storage Class Examples

Three examples show Custom Storage Class capabilities:

`rtwdemo_cscpredef` — Shows predefined custom storage classes and embedded signal objects

`rtwdemo_importstruct` — Shows custom storage classes used to access imported data efficiently

`rtwdemo_advsc` — Shows how custom storage classes can support data-object-driven modeling

Click the links above, or type the name in the MATLAB Command Window.

Resources for Defining Custom Storage Classes

The resources for working with custom storage class definitions are:

- Use MATLAB class syntax to create a data class in a package. You can assign properties to the data class and add initialization code to enable custom storage class definition. For complete instructions, see “Define Data Classes” in Simulink documentation.
- A set of ready-to-use CSCs. These CSCs are designed to be useful in code generation for embedded systems development. CSC functionality is integrated into the `Simulink.Signal` and `Simulink.Parameter` classes; you do not need to use special object classes to generate code with CSCs.
- The Custom Storage Class Designer (`cscdesigner`) tool, which is described in this chapter. This tool lets you define CSCs that are tailored to your code generation requirements. The Custom Storage Class Designer provides a graphical user interface that you can use to implement CSCs. You can use your CSCs in code generation immediately, without a Target Language Compiler (TLC) or other programming. See “Design Custom Storage Classes and Memory Sections” on page 9-8 for details.

Simulink Package Custom Storage Classes

The `Simulink` package includes a set of built-in custom storage classes. These are categorized as custom storage classes, even though they are built-in, because they:

- Extend the storage classes provided by the Simulink Coder software
- Are functionally the same as if you had defined them yourself using the CSC Designer

You cannot change the CSCs built into the `Simulink` package, but you can subclass the package and add CSCs to the subclass, following the steps in “Resources for Defining Custom Storage Classes” on page 9-5.

Some CSCs in the `Simulink` package are valid for parameter objects but not signal objects and vice versa. For example, you can assign the storage class `Const` to a parameter but not to a signal, because signal data is not constant. The next table defines the CSCs built into the `Simulink` package and shows where each of the CSCs can be used.

CSC Name	Purpose	Signals?	Parameters?
<code>BitField</code>	Generate a <code>struct</code> declaration that embeds Boolean data in named bit fields.	Y	Y
<code>CompilerFlag</code>	Supports preprocessor conditionals defined via compiler flag. See “Generate Preprocessor Conditionals for Variant Systems”.	N	Y
<code>Const</code>	Generate a constant declaration with the <code>const</code> type qualifier.	N	Y
<code>ConstVolatile</code>	Generate declaration of volatile constant with the <code>const volatile</code> type qualifier.	N	Y
<code>Default</code>	<code>Default</code> is a placeholder CSC that the code generator assigns to the <code>CoderInfo.CustomStorageClass</code> property of signal and parameter objects when they are created. You cannot edit the <code>Default</code> CSC definition.	Y	Y

CSC Name	Purpose	Signals?	Parameters?
Define	Generate <code>#define</code> directive.	Y	Y
ExportToFile	Generate header (.h) file, with user-specified name, containing global variable declarations.	Y	Y
FileScope	Generate a static qualifier suffix for a variable declaration so that the scope of the variable is limited to the current file.	Y	Y
GetSet	Supports specialized function calls to read and write the memory associated with a Data Store Memory block. See “GetSet Custom Storage Class” on page 9-53.	Y	Y
ImportedDefine	Supports preprocessor conditionals defined via legacy header file. See “Generate Preprocessor Conditionals for Variant Systems”.	N	Y
ImportFromFile	Generate directives to include predefined header files containing global variable declarations.	Y	Y
Reusable	Allows the code generator to reuse a pair of root I/O signals when you specify the same name and the same custom storage class for both. The custom storage class is either <code>Reusable (Custom)</code> or derived from <code>Reusable (Custom)</code> .	Y	N
Struct	Generate a <code>struct</code> declaration encapsulating parameter or signal object data.	Y	Y
Volatile	Use <code>volatile</code> type qualifier in declaration.	Y	Y

Design Custom Storage Classes and Memory Sections

In this section...

“Create Packages for Custom Storage Class Definitions” on page 9-8

“Use Custom Storage Class Designer” on page 9-8

“Edit Custom Storage Class Properties” on page 9-14

“Use Custom Storage Class References” on page 9-19

“Create and Edit Memory Section Definitions” on page 9-23

“Use Memory Section References” on page 9-27

Create Packages for Custom Storage Class Definitions

Use MATLAB class syntax to create a data class in a package. You can assign properties to the data class and add initialization code to enable custom storage class definition. For complete instructions, see “Define Data Classes” in the Simulink documentation.

Use Custom Storage Class Designer

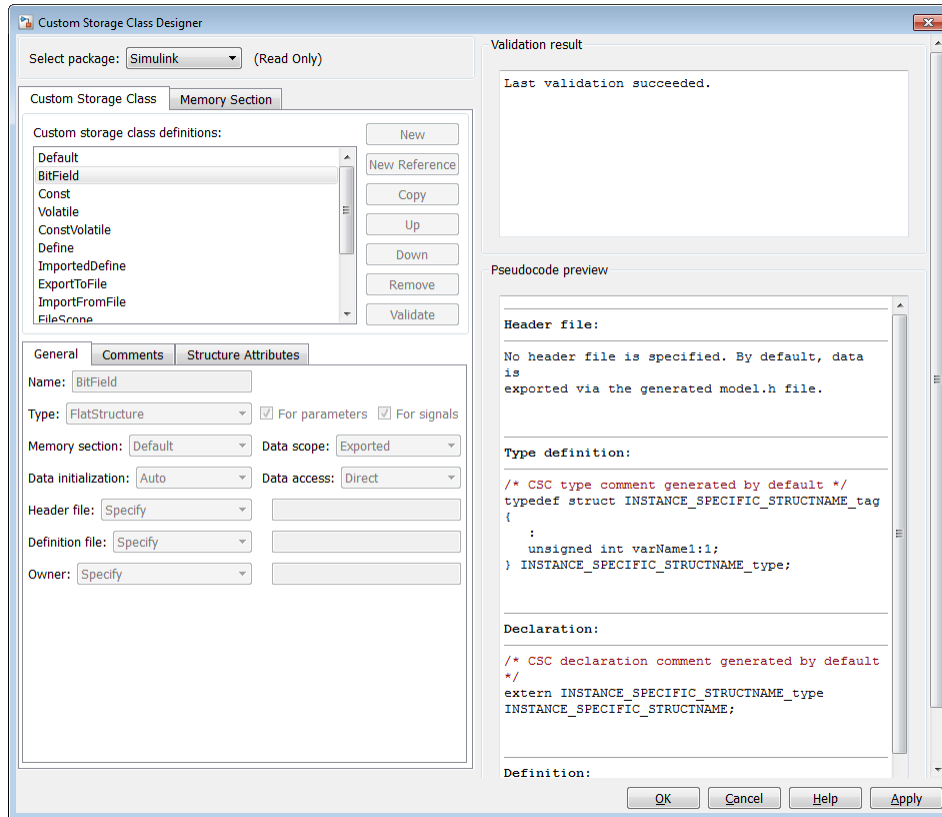
The Custom Storage Class Designer (`cscdesigner`) is a tool for creating and managing custom storage classes and memory sections. You can use the Custom Storage Class Designer to:

- Load existing custom storage classes and memory sections and view and edit their properties
- Create new custom storage classes and memory sections
- Create references to custom storage classes and memory sections defined in other packages
- Copy and modify existing custom storage class and memory section definitions
- Check a custom storage class and memory section definitions
- Preview pseudocode generated from custom storage class and memory section definitions
- Save custom storage class and memory section definitions

To open the Custom Storage Class Designer for a particular package, type the following command at the MATLAB prompt:

cscdesigner ('mypkg')

When first opened, the Custom Storage Class Designer scans data class packages on the MATLAB path to detect packages that have a CSC registration file. A message is displayed while scanning proceeds. When the scan is complete, the Custom Storage Class Designer window appears:



The Custom Storage Class Designer window is divided into several panels:

- **Select package:** Lets you select from a menu of data class packages that have CSC definitions associated with them. See “Select Data Class Package” on page 9-10 for details.
- **Custom Storage Class / Memory Section** properties: Lets you select, view, edit, copy, verify, and perform other operations on CSC definitions or memory section

definitions. The common controls in the **Custom Storage Class / Memory Section** properties panel are described in “Manipulate Custom Storage Classes and Memory Sections” on page 9-11.

- When the **Custom Storage Class** tab is selected, you can select a CSC definition or reference from a list and edit its properties. See “Edit Custom Storage Class Properties” on page 9-14 for details.
- When the **Memory Section** tab is selected, you can select a memory section definition or reference from a list and edit its properties. See “Create and Edit Memory Section Definitions” on page 9-23 for details.
- **Filename:** Displays the filename and location of the current CSC registration file, and lets you save your CSC definition to that file. See “Save Definitions” on page 9-13 for details.
- **Pseudocode preview:** Displays a preview of code that is generated from objects of the given class. The preview is pseudocode, since the actual symbolic representation of data objects is not available until code generation time. See “Preview Generated Code” on page 9-26 for details.
- **Validation result:** Displays errors encountered when the currently selected CSC definition is validated. See “Validate Definitions Category” on page 9-19 for details.

Select Data Class Package

A CSC or memory section definition or reference is uniquely associated with a Simulink data class package. The link between the definition/reference and the package is formed when a CSC registration file (`csc_registration.m`) is located in the package directory.

You need not search for or edit a CSC registration file directly: the Custom Storage Class Designer locates available CSC registration files. The **Select package** menu contains names of data class packages that have a CSC registration file on the MATLAB search path.

When you select a package, the CSCs and memory section definitions belonging to the package are loaded into memory and their names are displayed in the scrolling list in the **Custom storage class** panel. The name and location of the CSC registration file for the package is displayed in the **Filename** panel.

If you select a user-defined package, by default you can use the Custom Storage Class Designer to edit its custom storage classes and memory sections. If you select a built-

in package, you cannot edit its custom storage classes or memory sections. See “Custom Storage Class Implementation” on page 9-56 for more information.

Manipulate Custom Storage Classes and Memory Sections

The **Custom Storage Class / Memory Section** panel lets you select, view, and (if the CSC is writable) edit CSC and memory section definitions and references. In the next figure and the subsequent examples, the selected package is `mypkg`. Instructions for creating a user-defined package like `mypkg` appear in “Design Custom Storage Classes and Memory Sections”.

Select package: `mypkg`

Custom Storage Class | Memory Section

Custom storage class definitions:

- Default
- BitField
- Const
- Volatile
- ConstVolatile
- Define
- ImportedDefine
- ExportToFile
- ImportFromFile
- FileScope

New
New Reference
Copy
Up
Down
Remove
Validate

General | Comments | Structure Attributes

Name: `BitField`

Type: `FlatStructure` For parameters For signals

Memory section: `Default` Data scope: `Exported`

Data initialization: `Auto` Data access: `Direct`

Header file: `Specify`

Definition file: `Specify`

Owner: `Specify`

The list at the top of the panel displays the definitions/references for the currently selected package. To select a definition/reference for viewing and editing, click on the desired list entry. The properties of the selected definition/reference appear in the area

below the list. The number and type of properties vary for different types of CSC and memory section definitions. See:

- “Edit Custom Storage Class Properties” on page 9-14 for information about the properties of the predefined CSCs.
- “Create and Edit Memory Section Definitions” on page 9-23 for information about the properties of the predefined memory section definitions.

The buttons to the right of the list perform these functions, which are common to both custom storage classes and memory definitions:

- **New:** Creates a new CSC or memory section with default values.
- **New Reference:** Creates a reference to a CSC or memory section definition in another package. The default initially has a default name and properties. See “Use Custom Storage Class References” on page 9-19 and “Use Memory Section References” on page 9-27.
- **Copy:** Creates a copy of the selected definition / reference. The copy initially has a default name using the convention:

`definition_name_n`

where `definition_name` is the name of the original definition, and `n` is an integer indicating successive copy numbers (for example: `BitField_1`, `BitField_2`, ...)

- **Up:** Moves the selected definition one position up in the list.
- **Down:** Moves the selected definition one position down in the list
- **Remove:** Removes the selected definition from the list.
- **Validate:** Performs a consistency check on the currently selected definition. Errors are reported in the **Validation result** panel.

For example, if you click **New**, a new custom storage class is created with a default name:

Select package: mypkg

Custom Storage Class Memory Section

Custom storage class definitions:

- Default
- NewCSC_1
- BitField
- Const
- Volatile
- ConstVolatile
- Define
- ImportedDefine
- ExportToFile
- ImportFromFile

New
New Reference
Copy
Up
Down
Remove
Validate

General Comments

Name: NewCSC_1

Type: Unstructured For parameters For signals

Memory section: Default Data scope: Auto

Data initialization: Auto Data access: Direct

Header file: Specify

Definition file: Specify

Owner: Specify

You can now rename the new class by typing the desired name into the **Name** field, and specify other fields.

Note: The class name must be a valid MATLAB variable name. See “Variable Names”

Click **Apply** or **OK**.

Save Definitions

After you have created or edited a CSC or memory section definition or reference, you must save the changes to the CSC registration file. To do this, click **Save** in the **Filename** panel. When you click **Save**, the current CSC and memory section definitions that are in memory are validated, and the definitions are written out.

If errors occur, they are reported in the **Validation result** panel. The definitions are saved whether or not errors exist. However, you should resolve validation errors and resave your definitions. Trying to use definitions that were saved with validation errors can cause additional errors. Such problems can occur even if you do not try to use the specific parts of the definition that contain the validation errors, making the problems difficult to diagnose.

Restart MATLAB After Changing Definitions

If you add, change, or delete custom storage class or memory section definitions for a user-defined class, and objects of that class already exist, you must restart MATLAB to use the changed definitions and to eliminate obsolete objects. When you save the changed definitions, a message appears indicating that you must restart MATLAB.

Edit Custom Storage Class Properties

To view and edit the properties of a CSC, click the **Custom Storage Class** tab in the **Custom Storage Class / Memory Section** panel. Then, select a CSC name from the **Custom storage class definitions** list.

The CSC properties are divided into several categories, selected by tabs. Selecting a class, and setting property values for that class, can change the available tabs, properties, and values. As you change property values, the changes in the generated code is immediately displayed in the **Pseudocode preview** panel. In most cases, you can define your CSCs quickly and easily by selecting the **Pseudocode preview** panel and using the **Validate** button frequently.

The property categories and corresponding tabs are as follows:

General Category

Properties in the **General** category are common to CSCs. In the next figure and the subsequent examples, the selected custom storage class is `ByteField`. Instructions for creating a user-defined custom storage class like `ByteField` appear in “Manipulate Custom Storage Classes and Memory Sections” on page 9-11.

The screenshot shows the 'General' tab of a configuration window. The 'Name' field contains 'ByteField'. The 'Type' dropdown is set to 'Unstructured'. There are two checked checkboxes: 'For parameters' and 'For signals'. The 'Memory section' dropdown is 'Default', and the 'Data scope' dropdown is 'Auto'. The 'Data initialization' dropdown is 'Auto', and the 'Data access' dropdown is 'Direct'. The 'Header file', 'Definition file', and 'Owner' dropdowns are all set to 'Specify', with corresponding empty text input fields to their right.

Properties in the **General** category, and the possible values for each property, are as follows:

- **Name:** The CSC name, selected from the **Custom storage class definitions** list. The name cannot be a TLC keyword. Violating this rule causes an error.
- **Type:** Specifies how objects of this class are stored. Values:
 - **Unstructured:** Objects of this class generate unstructured storage declarations (for example, scalar or array variables), for example:


```
datatype dataname[dimension];
```
 - **FlatStructure:** Objects of this class are stored as members of a struct. A **Structure Attributes** tab is also displayed, allowing you to specify additional properties such as the struct name. See “Structure Attributes Category” on page 9-18.
 - **Other:** Used for certain data layouts, such as nested structures, that cannot be generated using the standard **Unstructured** and **FlatStructure** custom storage class types. If you want to generate other types of data, you can create a new custom storage class from scratch by writing TLC code. See “Define Advanced Custom Storage Classes Types” on page 9-50 for more information.
- **For parameters** and **For signals:** These options let you enable a CSC for use with only certain classes of data objects. For example, it does not make sense to assign storage class **Const** to a **Simulink.Signal** object. Accordingly, the **For signals** option for the **Const** class is deselected, while the **For parameters** is selected.

- **Memory section:** Selects one of the memory sections defined in the **Memory Section** panel. See “Create and Edit Memory Section Definitions” on page 9-23.
- **Data scope:** Controls the scope of symbols generated for data objects of this class. Values:
 - **Auto:** Symbol scope is determined internally by code generation. If possible, symbols have **File** scope. Otherwise, they have **Exported** scope.
 - **Exported:** Symbols are exported to external code in the header file specified by the **Header File** field. If a **Header File** is not specified, symbols are exported to external code in *model.h*.
 - **Imported:** Symbols are imported from external code in the header file specified by the **Header File** field. If you do not specify a header file, an `extern` directive is generated in *model_private.h*. For imported data, if the **Data initialization** value is **Macro**, a header file *must* be specified.
 - **File:** The scope of each symbol is the file that defines it. File scope requires each symbol to be used in a single file. If the same symbol is referenced in multiple files, an error occurs at code generation time.
 - **Instance specific:** Symbol scope is defined by the **Data scope** field of the `CoderInfo.CustomAttributes` property of each data object.
- **Data initialization:** Controls how storage is initialized in generated code. Values:
 - **Auto:** Storage initialization is determined internally by the code generation. Parameters have **Static** initialization, and signals have **Dynamic** initialization.
 - **None:** Initialization code is not generated.
 - **Static:** A static initializer of the following form is generated:

```
datatype dataname[dimension] = {...};
```
 - **Dynamic:** Variable storage is initialized at runtime, in the *model_initialize* function.
 - **Macro:** A macro definition of the following form is generated:

```
#define data numeric_value
```

The **Macro** initialization option is available only for use with unstructured parameters. It is not available when the class is configured for generation of structured data, or for signals. If the **Data scope** value is **Imported**, a header file *must* be specified.

- **Instance specific:** Initialization is defined by the **Data initialization** property of each data object.

Note: The code generator might include dynamic initialization code for signals and states even if the CSC has **Data initialization** set to **None** or **Static**, if the initialization is required.

- **Data access:** Controls whether imported symbols are declared as variables or pointers. This field is enabled only when **Data scope** is set to **Imported** or **Instance-specific**. Values:
 - **Direct:** Symbols are declared as simple variables, such as

```
extern myType myVariable;
```
 - **Pointer:** Symbols are declared as pointer variables, such as

```
extern myType *myVariable;
```
 - **Instance specific:** Data access is defined by the **Data access** property of each data object.
- **Header file:** Defines the name of a header file that contains exported or imported variable declarations for objects of this class. Values:
 - **Specify:** An edit field is displayed to the right of the property. This lets you specify a header file for exported or imported storage declarations. Specify the full filename, including the filename extension (such as `.h`). Use quotes or brackets as in C code to specify the location of the header file. Leave the edit field empty to not specify a header file.
 - **Instance specific:** The header file for each data object is defined by the **Header file** property of the object. Leave the property undefined to not specify a header file for that object.

If the **Data scope** is **Exported**, specifying a header file is optional. If you specify a header file name, the custom storage class generates a header file containing the storage declarations to be exported. Otherwise, the storage declarations are exported in `model.h`.

If the **Data scope** of the class is **Imported**, and **Data initialization** is **Macro**, you *must* specify a header file name. A `#include` directive for the header file is generated.

Comments Category

Comments

The **Comments** panel lets you specify comments to be generated with definitions and declarations.

Comments must conform to the ANSI C standard (`/* . . . */`). Use `\n` to specify a new line.

Properties in the **Comments** tab are as follows:

- **Comment rules:** If **Specify** is selected, edit fields are displayed for entering comments. If **Default** is selected, comments are generated under control of the code generation software.
- **Type comment:** The comment entered in this field precedes the **typedef** or **struct** definition for structured data.
- **Declaration comment:** Comment that precedes the storage declaration.
- **Definition comment:** Comment that precedes the storage definition.

Structure Attributes Category

The **Structure Attributes** panel gives you detailed control over code generation for structs (including bitfields). The **Structure Attributes** tab is displayed for CSCs whose **Type** parameter is set to **FlatStructure**. The following figure shows the **Structure Attributes** panel.

The **Structure Attributes** properties are as follows:

- **Struct name:** If you select **Instance specific**, specify the struct name when configuring each instance of the class.

If you select **Specify**, an edit field appears for entry of the name of the structure to be used in the **struct** definition. Edit fields **Type tag**, **Type token**, and **Type name** are also displayed.

- **Is typedef:** When this option is selected a **typedef** is generated for the struct definition, for example:

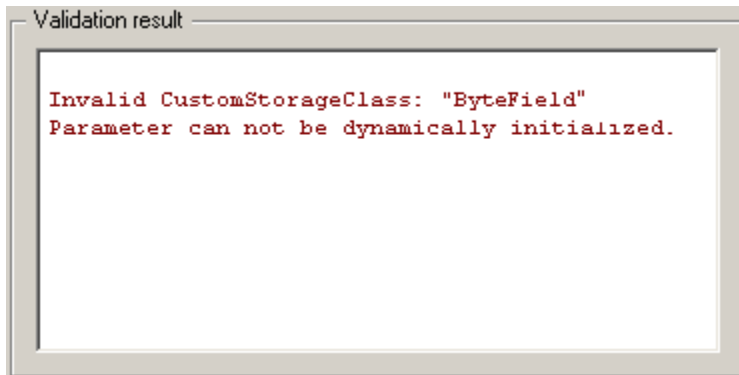
```
typedef struct {  
    . . .  
} SignalDataStruct;
```


Otherwise, a simple struct definition is generated.

- **Bit-pack booleans:** When this option is selected, signals and/or parameters that have Boolean data type are packed into bit fields in the generated struct.
- **Type tag:** Specifies a tag to be generated after the `struct` keyword in the struct definition.
- **Type name:** Specifies the string to be used in `typedef` definitions. This field is visible if **Is typedef** is selected.
- **Type token:** Some compilers support an additional token (which is simply another string) after the type tag. To generate such a token, enter the string in this field.

Validate Definitions Category

To validate a CSC definition, select the definition on the **Custom Storage Class** panel and click **Validate**. The Custom Storage Class Designer then checks the definition for consistency. The **Validation result** panel displays a errors encountered when the selected CSC definition is validated. The next figure shows the **Validation result** panel with a typical error message:



Validation is also performed whenever CSC definitions are saved. In this case, all CSC definitions are validated. (See “Save Definitions” on page 9-13.)

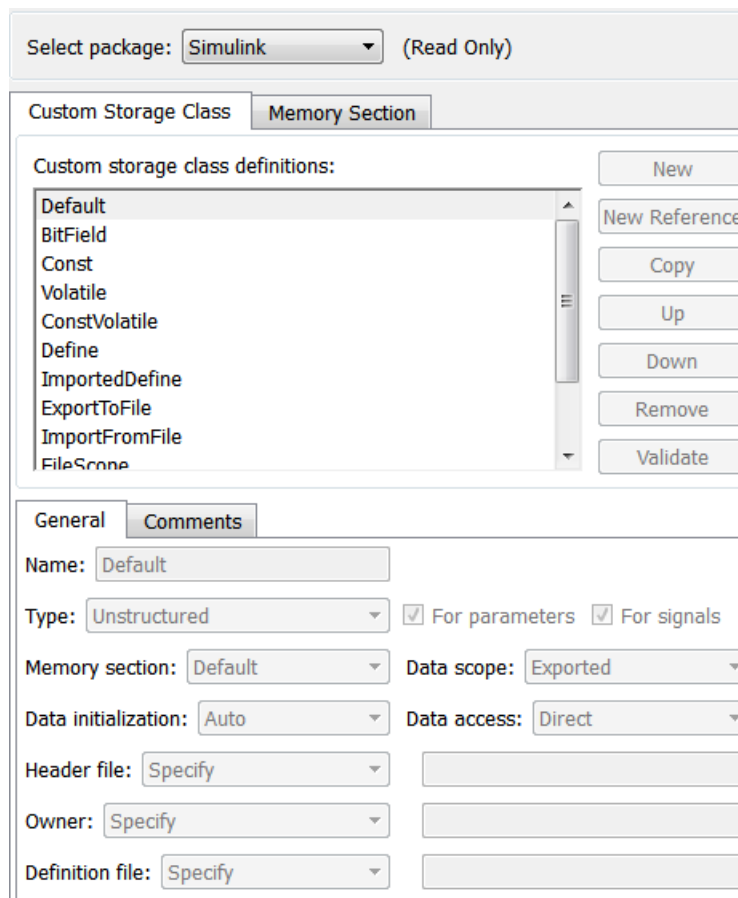
Use Custom Storage Class References

Packages can access and use custom storage classes that are defined in other packages, including both user-defined packages and predefined packages such as **Simulink** and **mpt**. Only one copy of the storage class exists, in the package that first defined

it. Other packages refer to it by pointing to it in its original location. Changes to the class, including changes to a predefined class in later MathWorks product releases, are immediately available in every referencing package.

To configure a package to use a custom storage class that is defined in another package:

- 1 Type `cscdesigner` to launch the Custom Storage Class Designer.



- 2 Select the **Custom Storage Class** tab.
- 3 Use **Select Package** to select the package in which you want to reference a class or section defined in some other package. The selected package must be writable.

- 4 In the **Custom storage class definitions** pane, select the existing definition below which you want to insert the reference. For example:

Select package: mypkg

Custom Storage Class Memory Section

Custom storage class definitions:

- Default
- BitField
- Const
- Volatile
- ConstVolatile
- Define
- ImportedDefine
- ExportToFile
- ImportFromFile
- FileScope

New

New Reference

Copy

Up

Down

Remove

Validate

General Comments Structure Attributes

Name: BitField

Type: FlatStructure For parameters For signals

Memory section: Default Data scope: Exported

Data initialization: Auto Data access: Direct

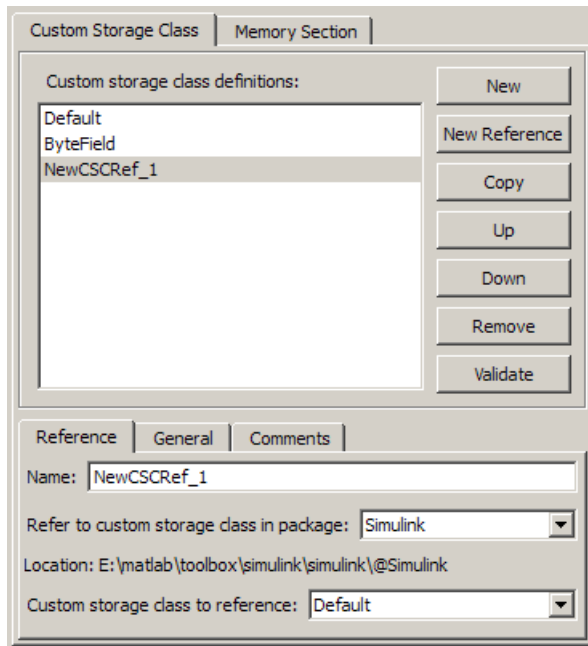
Header file: Specify

Definition file: Specify

Owner: Specify

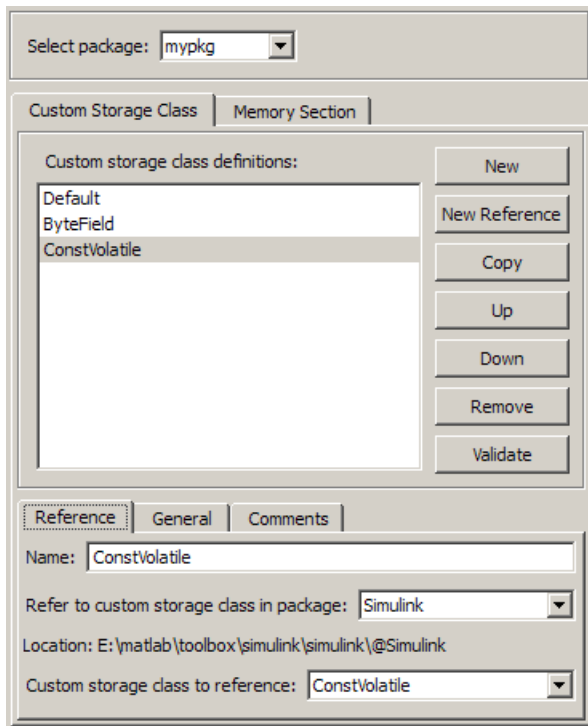
- 5 Click **New Reference**.

A new reference with a default name and properties appears below the previously selected definition. The new reference is selected, and a **Reference** tab appears that shows the reference's initial properties. A typical appearance is:



- 6 Use the **Name** field to enter a name for the new reference. The name must be unique in the importing package, but can duplicate the name in the source package. The name cannot be a TLC keyword. Violating this rule causes an error.
- 7 Set **Refer to custom storage class in package** to specify the package that contains the custom storage class you want to reference.
- 8 Set **Custom storage class to reference** to specify the custom storage class to be referenced. Trying to create a circular reference generates an error and leaves the package unchanged.
- 9 Click **OK** or **Apply** to save the changes to memory. See “Save Definitions” on page 9-13 for information about saving changes permanently.

For example, the next figure shows the custom storage class `ConstVolatile` imported from the `Simulink` package into `mypkg`, and given the same name that it has in the source package. Other names could have been used without affecting the properties of the storage class.



You can use Custom Storage Class Designer capabilities to copy, reorder, validate, and otherwise manage classes that have been added to a class by reference. However, you cannot change the underlying definitions. You can change a custom storage class only in the package where it was originally defined.

Change Existing Custom Storage Class References

To change an existing CSC reference, select it in the **Custom storage class definitions** pane. The **Reference** tab appears, showing the current properties of the reference. Make changes, then click **OK** or **Apply** to save the changes to memory. See “Save Definitions” on page 9-13 for information about saving changes permanently.

Create and Edit Memory Section Definitions

Memory section definitions add comments, qualifiers, and `#pragma` directives to generated symbol declarations. The **Memory Section** tab lets you create, view, edit, and

verify memory section definitions. The steps for creating a memory section definition are essentially the same as for creating a custom storage class definition:

- 1 Select a writable package in the **Select package** field.
- 2 Select the **Memory Section** tab. In a new package, only a **Default** memory section initially appears.
- 3 Select the existing memory section below which you want to create a new memory section.
- 4 Click **New**.

A new memory section definition with a default name appears below the selected memory section.

- 5 Set the name and other properties of the memory section.
- 6 Click **OK** or **Apply**.

The next figure shows `mypkg` with a memory section called `MyMemSect`:

Select package: mypkg

Custom Storage Class | **Memory Section**

Memory section definitions:

Default
MyMemSect

New
New Reference
Copy
Up
Down
Remove
Validate

Memory Section

Name: MyMemSect

Is const Is volatile Qualifier:

Comment:

Pragma surrounds: All variabl

Pre-memory-section pragma:

Post-memory-section pragma:

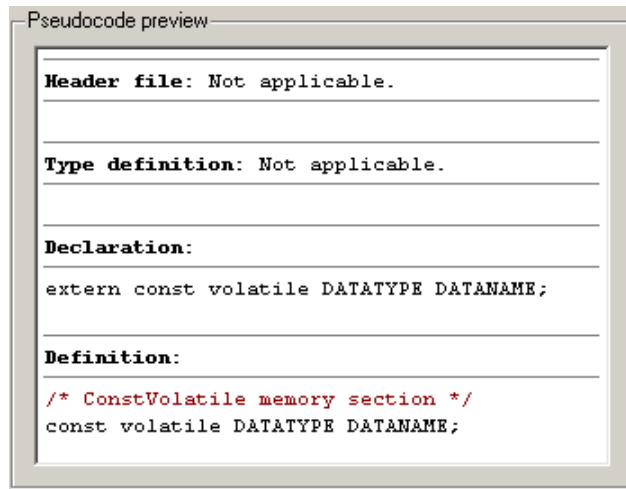
The **Memory section definitions** list lets you select a memory section definition to view or edit. The available memory section definitions also appear in the **Memory section name** menu in the **Custom Storage Class** panel. The properties of a memory section definition are as follows:

- **Memory section name:** Name of the memory section (displayed in **Memory section definitions** list).
- **Is const:** If selected, a `const` qualifier is added to the symbol declarations.

- **Is volatile:** If selected, a `volatile` qualifier is added to the symbol declarations.
- **Qualifier:** The string entered into this field is added to the symbol declarations as a further qualifier. Note that verification is not performed on this qualifier.
- **Memory section comment:** Comment inserted before declarations belonging to this memory section. Comments must conform to the ANSI C standard (`/* . . . */`). Use `\n` to specify a new line.
- **Pragma surrounds:** Specifies whether the pragma should surround `All variables` or `Each variable`. When **Pragma surrounds** is set to `Each variable`, the `%<identifier>` token is allowed in pragmas and will be replaced by the variable or function name.
- **Pre-memory section pragma:** pragma directive that precedes the storage definition of data belonging to this memory section. The directive must begin with `#pragma`.
- **Post-memory section pragma:** pragma directive that follows the storage definition of data belonging to this memory section. The directive must begin with `#pragma`.

Preview Generated Code

If you click **Validate** on the **Memory Section** panel, the **Pseudocode preview** panel displays a preview of code that is generated from objects of the given class. The panel also displays messages (in blue) to highlight changes as they are made. The code preview changes dynamically as you edit the class properties. The next figure shows a code preview for the `MemConstVolatile` memory section.



```
Pseudocode preview
Header file: Not applicable.
Type definition: Not applicable.
Declaration:
extern const volatile DATATYPE DATANAME;
Definition:
/* ConstVolatile memory section */
const volatile DATATYPE DATANAME;
```


Use Memory Section References

Packages can access and use memory sections that are defined in other packages, including both user-defined packages and predefined packages such as `Simulink` and `mpt`. Only one copy of the section exists, in the package that first defined it; other packages refer to it by pointing to it in its original location. Changes to the section, including changes to a predefined section in later MathWorks product releases, are immediately available in every referencing package.

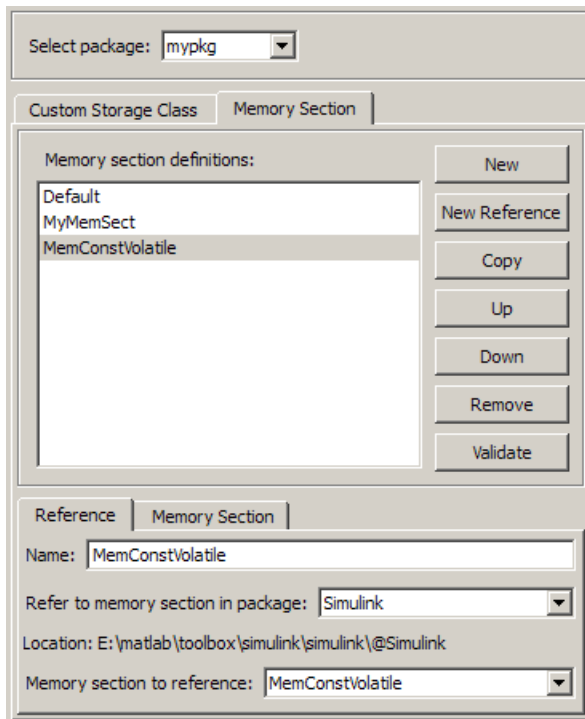
To configure a package to use a memory section that is defined in another package:

- 1 Type `cscdesigner` to launch the Custom Storage Class Designer.
- 2 Select the **Memory Section** tab.
- 3 Use **Select Package** to select the package in which you want to reference a class or section defined in some other package.
- 4 In the **Memory section definitions** pane, select the existing definition below which you want to insert the reference.
- 5 Click **New Reference**.

A new reference with a default name and properties appears below the previously selected definition. The new reference is selected, and a **Reference** tab appears that shows the reference's initial properties.

- 6 Use the **Name** field to enter a name for the new reference. The name must be unique in the importing package, but can duplicate the name in the source package.
- 7 Set **Refer to memory section in package** to specify the package that contains the memory section you want to reference.
- 8 Set **Memory section to reference** to specify the memory section to be referenced. Trying to create a circular reference generates an error and leaves the package unchanged.
- 9 Click **OK** or **Apply** to save the changes to memory. See “Save Definitions” on page 9-13 for information about saving changes permanently.

For example, the next figure shows the memory section `MemConstVolatile` imported from the `Simulink` package into `mypkg`, and given the same name that it has in the source package. Other names could have been used without affecting the properties of the memory section.



You can use Custom Storage Class Designer capabilities to copy, reorder, validate, and otherwise manage memory sections that have been added to a class by reference. However, you cannot change the underlying definitions. You can change a memory section only in the package where it was originally defined.

Change Existing Memory Section References

To change an existing memory section reference, select it in the **Memory section definitions** pane. The **Reference** tab appears, showing the current properties of the reference. Make changes, then click **OK** or **Apply** to save the changes to memory. See “Save Definitions” on page 9-13 for information about saving changes permanently.

Apply Custom Storage Classes

In this section...

- “About Applying Custom Storage Classes” on page 9-29
- “Apply Custom Storage Classes to Parameters” on page 9-30
- “Apply Custom Storage Classes to Signals” on page 9-31
- “Custom Storage Classes Using Signal Objects” on page 9-32
- “Custom Storage Classes Using Embedded Signal Objects” on page 9-33
- “Specify Custom Storage Classes Using GUI” on page 9-40
- “Specify Custom Storages Classes Using API” on page 9-42

About Applying Custom Storage Classes

You can apply a custom storage class to a parameter or a signal using the GUI or the API.

- To apply a custom storage class to a parameter, you specify the storage class in the `Simulink.Parameter` object that defines the parameter in the base workspace.
- To apply a custom storage class to a signal, you specify the storage class in a `Simulink.Signal` object that is bound to the signal. You can provide this object in two ways:
 - Create the object in the base workspace, then bind it to the signal as described in “Symbol Resolution”. When you save the model, you must save the object in a separate file, as with base workspace objects.
 - Use the Signal Properties dialog box to embed the object in the model on the port where the signal originates. When you save the model, Simulink automatically saves the embedded signal object as part of the model file.

Most of the GUI techniques, and most of the API techniques, are the same for parameter and signal objects, and for base workspace and embedded signal objects. Only the initial steps differ, after which you apply the same GUI or API instructions within the context that you established in the initial steps.


The following instructions assume that you have already created packages, custom storage classes, and memory sections, as described in “Design Custom Storage Classes and Memory Sections” on page 9-8.

Apply Custom Storage Classes to Parameters

To apply a custom storage class to a parameter, you specify the storage class in the `Simulink.Parameter` object that defines the parameter in the base workspace. The instructions that begin in this section show you how to create that object using the GUI or API. Later instructions show you how to specify a custom storage class and custom attributes.

For information about using parameter objects to specify block parameter values, see “Use Parameter Objects to Specify Parameter Values”. For information about parameter storage in generated code, see “Parameters”.

Create Parameter Objects Using GUI

- 1 In the Model window, choose **View > Model Explorer**.
- 2 In the **Model Hierarchy** pane, select the **Base Workspace**.
- 3 Click the **Add Parameter** tool  or choose **Add > Simulink Parameter**.

Simulink creates a `Simulink.Parameter` object in the base workspace with the default name, `Param`.

- 4 Change the parameter name by editing it in the **Contents** pane. Example: `MyParam`.
- 5 Set parameter attributes other than **Code generation options** in the **Dialog** pane.
- 6 Follow the instructions in “Specify Custom Storage Classes Using GUI” on page 9-40.

Create Parameter Objects Using API

- 1 In the MATLAB Command Window, enter:

```
ParamName=ParamClass
```

where *ParamClass* is `Simulink.Parameter` or a subclass of it that you have defined.

- 2 Simulink creates a *ParamClass* object with the specified name:

```
MyParam =
```

```
Simulink.Parameter (handle)  
Value: []
```

```

    CoderInfo: [1x1 Simulink.ParamCoderInfo]
Description: ''
    DataType: 'auto'
        Min: []
        Max: []
    DocUnits: ''
    Complexity: 'real'
    Dimensions: [0 0]

```

- 3 Set parameter attributes other than `CoderInfo`, which controls custom storage classes.
- 4 Follow the instructions in “Specify Custom Storage Classes Using API” on page 9-42.

Apply Custom Storage Classes to Signals

To apply a custom storage class to a signal, you specify the storage class in a `Simulink.Signal` object. This object can exist in either of two locations:

- In the MATLAB base workspace
- On the port where the signal originates

The object itself is the same in either case; only its location and some of the techniques for managing it differ. The instructions that begin in this section show you how to create a signal object in either location using the GUI or API. Later instructions show you how to specify the custom storage class and custom attributes.

A given signal can be associated with at most one signal object. The signal can refer to the object more than once, but every reference must resolve to exactly the same object. A different signal object that has exactly the same properties will not meet the requirement for uniqueness. A compile-time error occurs if a model associates more than one signal object with a signal.


Assigning a signal to a non-`Auto` storage class automatically makes the signal a test point, overriding the setting of **Signal Properties > Logging and accessibility > Test point**. See “Test Points” for more information.

For information about using signal objects to specify signal attributes, see “Using Signal Objects to Initialize Signals and Discrete States”. For information about signal storage in generated code, see “Signals”.

Custom Storage Classes Using Signal Objects

The first step is to create the signal object in the base workspace, after which you specify signal attributes and the custom storage class and attributes.

Create Signal Objects in Base Workspace Using GUI

- 1 In the Simulink Editor, select **View > Model Explorer**.
- 2 In the **Model Hierarchy** pane, select the **Base Workspace**.
- 3 Click the **Add Signal** tool  or choose **Add > Simulink Signal**.

Simulink creates a `Simulink.Signal` object in the base workspace, with a default name, `Sig`.

- 4 Change the signal name by editing it in the **Contents** pane. Example: `MySig`.
- 5 Set signal attributes other than **Code generation options** in the **Dialog** pane.
- 6 Give the signal the same name as the signal object, as described in “Signal Names”.
- 7 Arrange for the signal to resolve to the object, as described in “Symbol Resolution”.
- 8 Follow the instructions in “Specify Custom Storage Classes Using GUI” on page 9-40.

Create Signal Objects in Base Workspace Using API

- 1 In the MATLAB Command Window, enter:

```
SignalName=SignalClass
```

where *SignalClass* is `Simulink.Signal` or a subclass of it that you have defined.

- 2 Simulink creates a *SignalClass* object with the specified name:

```
MySig =
```

```
Simulink.Signal (handle)  
  CoderInfo: [1x1 Simulink.SignalCoderInfo]  
  Description: ''  
  DataType: 'auto'  
  Min: []  
  Max: []  
  DocUnits: ''  
  Dimensions: -1
```

```

    Complexity: 'auto'
    SampleTime: -1
    SamplingMode: 'auto'
    InitialValue: ''

```

- 3 Set parameter attributes other than `CoderInfo`, which controls custom storage classes.
- 4 Give the signal the same name as the signal object, as described in “Signal Names”.
- 5 Arrange for the signal to resolve to the object, as described in “Symbol Resolution”.
- 6 Follow the instructions in “Specify Custom Storages Classes Using API” on page 9-42.

Custom Storage Classes Using Embedded Signal Objects

You can use the GUI or the API to apply a CSC using an embedded signal object.

- If you use the GUI, you use the Signal Properties dialog box to specify the attributes you want. The software then creates a `Simulink.Signal` object and assigns it to the output port where the signal originates.
- If you use the API, you instantiate `Simulink.Signal` or a subclass of it, set the attribute values that you want, and assign the object to the output port where the signal originates.

In either case, the generated code is the same as if you had created a base workspace signal object that specified the same name, CSC, and custom attributes as the embedded signal object. For more information, see “Custom Storage Classes Using Signal Objects” on page 9-32.

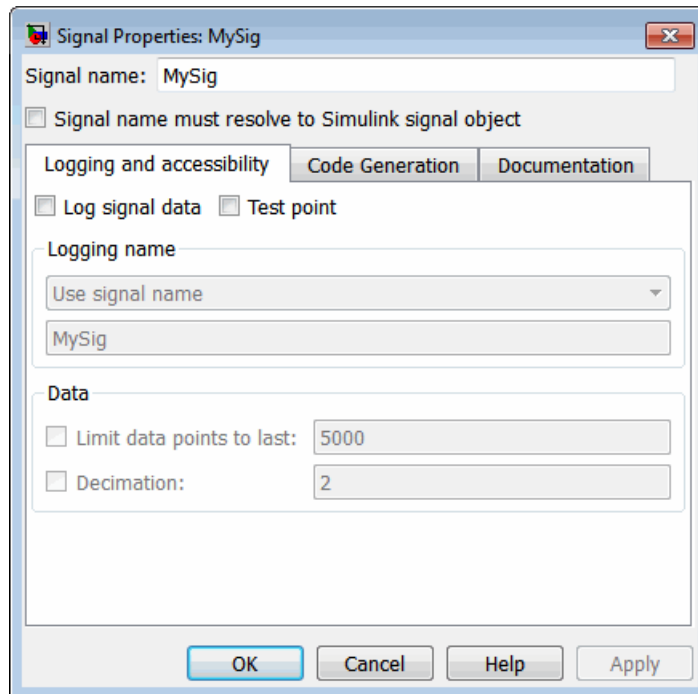
The advantages of using embedded signal objects are that they do not clutter the base workspace, and they do not need to be saved separately from the model, as base workspace objects do. When you save a model, Simulink saves embedded signal objects in the model file, and reloads the objects when you later reload the model.

The disadvantage of embedded signal objects is that you can use such an object *only* to specify a custom storage class, custom attributes, and an alias; you must accept the default values for other signal attributes. You cannot work around this restriction by providing additional information in a base workspace signal object on the same signal, because a signal object can have at most one associated signal object, as described in “Multiple Signal Objects”.

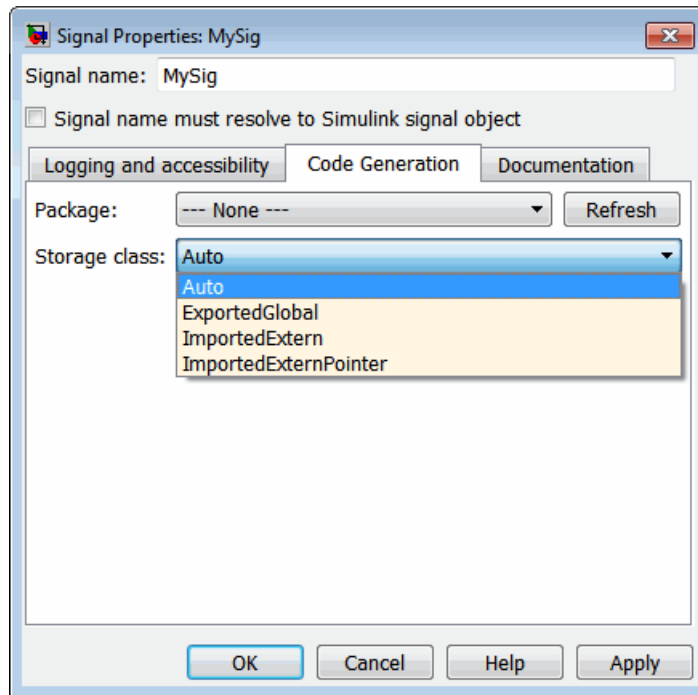
Create Embedded Signal Objects Using GUI

- 1 Give the signal a name, which must be a valid ANSI C identifier. Example: `MySig`.
- 2 Right-click the signal and choose **Properties** from the context menu.

The Signal Properties dialog box opens:

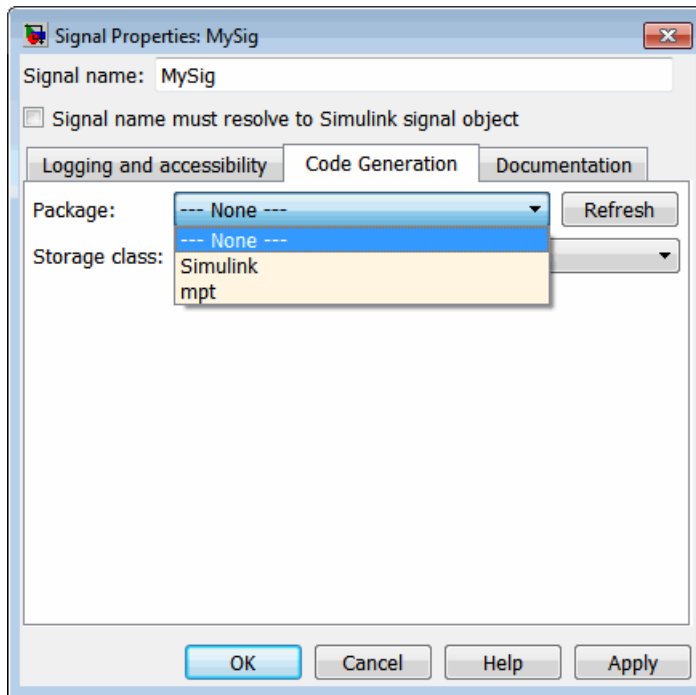


- 3 *Do not* select **Signal name must resolve to Simulink signal object**. Selecting it would require a base workspace signal object, which would conflict with the embedded signal object.
- 4 Click the **Code Generation** tab.
- 5 The **Package** is initially `---`**None**`---`. When a package is not specified, only the non-custom built-in storage classes defined for both GRT and ERT targets are available:

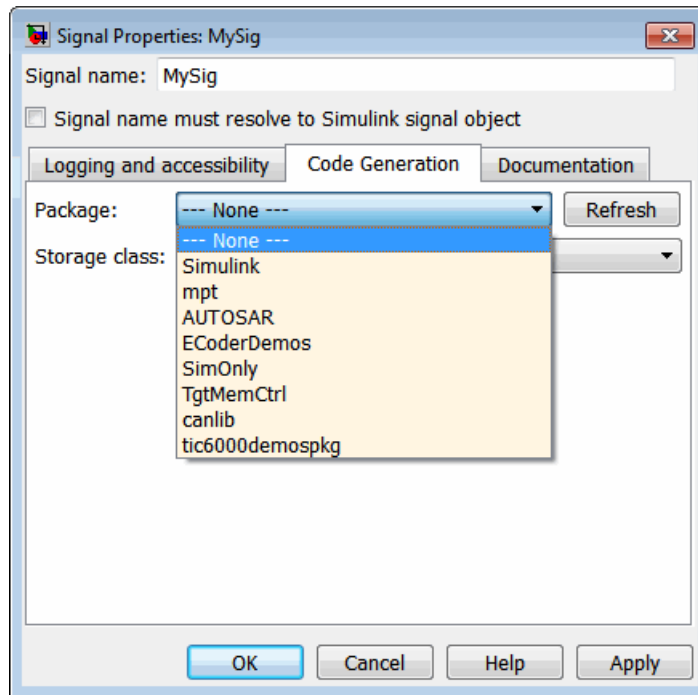


Applying a storage class when the package is ---None--- sets internal storage class attributes rather than creating an embedded signal object. For information about built-in storage classes, see “Signals” in the Simulink Coder documentation.

- 6 To apply a custom storage class, you must first specify the package where it is defined. Initially, viewing the **Package** menu displays only the built-in **Simulink** and **mpt** packages:

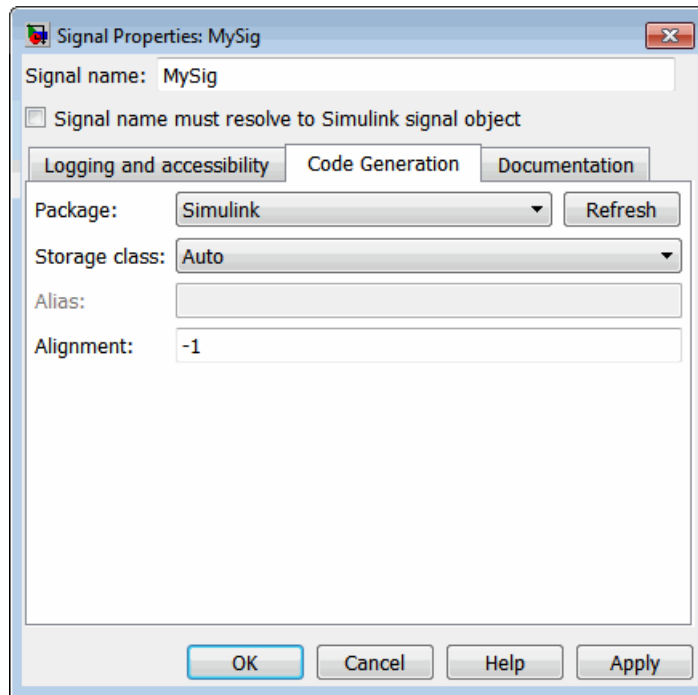


- 7 Click **Refresh** to load other available packages, including user-defined packages, available on the MATLAB path. After a brief delay, a timer box tracks the progress of the package search. After the search completes, viewing the **Package** menu displays available packages:



Once you have used **Refresh** in the Signal Properties dialog, Simulink saves the information for later use, so you do not have to click **Refresh** again during the current MATLAB session.

- 8 Select the package that contains the custom storage class you want to apply, e.g. Simulink:



- 9 Follow the instructions in “Specify Custom Storage Classes Using GUI” on page 9-40.

Delete Embedded Signal Objects Using GUI

To delete an embedded signal object with the Model Explorer, delete the name of the signal to which the object applies, by editing the name in the graphical model or in the Signal Properties dialog box. Simulink automatically deletes the embedded signal object as soon as its signal does not have a name.

Create Embedded Signal Objects Using API

To provide an embedded signal object using the API, you create the object, set its custom storage class and custom attributes, then assign the object to the output port on which it will be embedded.

- 1 Name the signal if it does not already have a name. The name must be a valid ANSI C identifier.

- 2 In the MATLAB Command Window, enter:

```
SignalName=SignalClass
```

where *SignalClass* is `Simulink.Signal` or a subclass of it that you have defined. The name of the signal object does not need to match the name of the signal to which the object will be applied.

- 3 Simulink creates a *SignalClass* object with the specified name. Example:

```
MySig =
Simulink.Signal (handle)
    CoderInfo: [1x1 Simulink.SignalCoderInfo]
    Description: ''
    DataType: 'auto'
        Min: []
        Max: []
    DocUnits: ''
    Dimensions: -1
    Complexity: 'auto'
    SampleTime: -1
    SamplingMode: 'auto'
    InitialValue: ''
```

- 4 *Do not* set attributes. An embedded signal object can specify *only* custom storage class information.
- 5 Follow the instructions in “Specify Custom Storage Classes Using API” on page 9-42. After specifying the custom storage class, be sure to assign the signal object to its output port, as described under “Assign Embedded Signal Objects to Output Ports” on page 9-45.

Change Embedded Signal Objects Using API

To change an embedded signal object using the API, you obtain a copy of the object from the output port on which it is embedded, change the object, then assign the changed object back to the port.

- 1 Obtain a copy of the signal object using a handle to the output port. Example:

```
hps=get_param(gcf,'PortHandles')
hp=hps.Outputport(1)
MySig=get_param(hp,'SignalObject')
```

- 2 Change the signal object using the techniques described in “Specify Custom Storage Classes Using API” on page 9-42. After making the changes, be sure to copy the signal object to its output port, as described in “Assign Embedded Signal Objects to Output Ports” on page 9-45.

Delete Embedded Signal Objects Using API

To delete an embedded signal object with the API, obtain a handle to the output port where the signal object is embedded, then set the port's `SignalObject` parameter to []:

```
hps=get_param(gcb, 'PortHandles')
hp=hps.Output(1)
set_param(hp, 'SignalObject', [])
```

Resolve Signal Names to Simulink Signal Objects Using API

To set the name of a signal to resolve to a Simulink signal object using the API, obtain the handle to the output port and then set the port's `MustResolveToSignalObject` property to on:

```
hps=get_param(gcb, 'PortHandles')
hp=hps.Output(1)
set_param(hp, 'MustResolveToSignalObject', 'on')
```

Specify Custom Storage Classes Using GUI

The initial steps for applying a CSC with the GUI differ depending on whether you are applying the CSC to a parameter using a base workspace object, to a signal using a base workspace object, or to a signal using an embedded object. The initial steps for each of these three cases appear in:

- “Create Parameter Objects Using GUI” on page 9-30
- “Create Signal Objects in Base Workspace Using GUI” on page 9-32
- “Create Embedded Signal Objects Using GUI” on page 9-34

After the initial steps, applying a CSC with the GUI is the same for the three cases. The following instructions show you how to finish applying a CSC with the GUI. The instructions assume that you have completed one of the previous sets of instructions, and that the dialog you used to execute those instructions is still open.

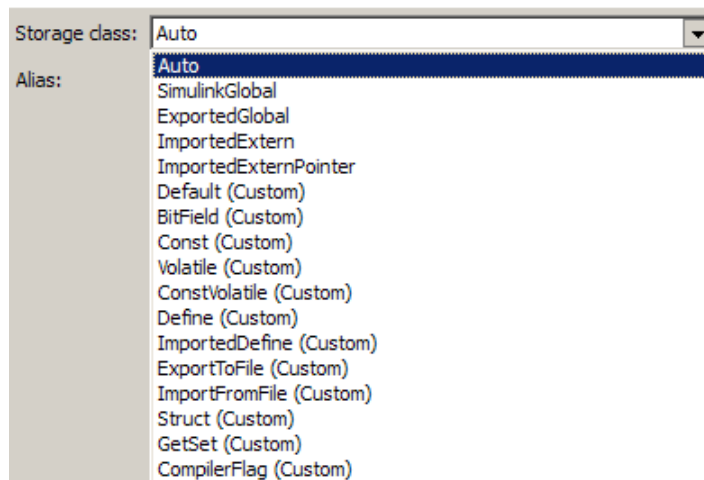
The available custom storage classes and custom attributes depend on the package that you select. The examples in this section assume that you are using the `Simulink` package.

The dialog that you used to begin the process of applying a CSC with the GUI by providing an object contains two fields: one for specifying a custom storage class and one for optionally specifying an alias.

Storage class is **Auto** because that is the default storage class in the **Simulink** package. Other packages may have different defaults. You can specify an **Alias** whenever the **Storage class** is not **Auto**. If **Storage class** is **Auto**, Simulink deletes aliases you try to specify, leaving the field blank. If you specify an alias, it appears in generated code instead of the name of the object.

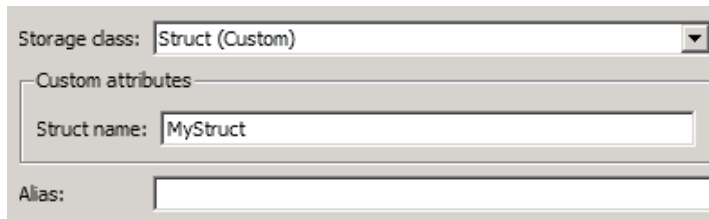
To specify a custom storage class and its custom attributes:

- 1 View the **Storage Class** menu, which looks like this for the **Simulink** package:



Each custom storage class has **(custom)** suffixed to its name. The storage classes **SimulinkGlobal**, **ExportedGlobal**, **ImportedExtern**, and **ImportedExternPointer** are the built-in non-custom storage classes described in “Signals” in the Simulink Coder documentation.

- 2 Choose the desired custom storage class from **Storage class**, for example, **Struct**.
- 3 Provide values for custom attributes. **Struct** has only one, **Struct name**. For example, set **Struct name** to **MyStruct**:



The image shows a configuration dialog box for a custom storage class. At the top, there is a dropdown menu labeled 'Storage class:' with 'Struct (Custom)' selected. Below this is a section titled 'Custom attributes' which contains a text input field labeled 'Struct name:' with the value 'MyStruct'. At the bottom of the dialog is another text input field labeled 'Alias:' which is currently empty.

4 Click **Apply**.

In generated code, data whose storage is controlled by this custom storage class specification will appear in a structure named `MyStruct`. See “Generate Code with Custom Storage Classes” on page 9-46 for an example.

Specify Custom Storage Classes Using API

The initial steps for applying a CSC with the API differ depending on whether you are applying the CSC to a parameter using a base workspace object, to a signal using a base workspace object, or to a signal using an embedded object. The initial steps for each of these three cases appear in:

- “Create Parameter Objects Using API” on page 9-30
- “Create Signal Objects in Base Workspace Using API” on page 9-32
- “Create Embedded Signal Objects Using API” on page 9-38

After the initial steps, applying a CSC with the API is the same for the three cases, except for the case of an assignment for an embedded signal object. The following instructions show you how to finish applying a CSC with the API. The instructions assume that you have completed one of the previous sets of instructions, and that the resulting objects and attributes are unchanged.

The available custom storage classes and custom attributes depend on the package that you select. The examples in this section assume that you are using the `Simulink` package. The examples also assume that the object for which you want to specify a custom storage class is named `MyObj`, which is a parameter or signal object that exists in the base workspace, or a signal object that will be assigned to an output port.

The rest of this section provides information that is specific to custom storage classes in Embedded Coder. See “Simulink Package Custom Storage Classes” on page 9-6 for a

list of the custom storage classes that are built into the `Simulink` package for use by Embedded Coder software.

CoderInfo Properties

Each Simulink parameter object or signal object defines properties called `CoderInfo` properties. Code generation software uses these properties to control storage class assignment in the generated code. The `CoderInfo` properties and their default values are as follows:

```
StorageClass: 'Auto'
Alias: ''
CustomStorageClass: 'Default'
CustomAttributes: [1x1 SimulinkCSC.AttribClass_Simulink_Default]
```

For more information about `CoderInfo` properties, see “Signals” in the Simulink Coder documentation.

Specify a Custom Storage Class

To specify a custom storage class using `CoderInfo` properties:

- 1 Set `StorageClass` to 'Custom'.
- 2 Set `CustomStorageClass` to the name of the storage class.

For example, to specify the `Struct` custom storage class:

```
MyObj.CoderInfo.StorageClass='Custom'
MyObj.CoderInfo.CustomStorageClass='Struct'
```

Whenever you have specified a custom storage class other than `Auto`, you can specify an alias by setting the `Alias` attribute. If you specify an alias, it appears in generated code instead of the name of the object.

Specify Instance-Specific Attributes

A custom storage class can have properties that define attributes that are specific to that CSC. Such properties are called **instance-specific attributes**. For example, if you specify the `Struct` custom storage class, you must specify the name of the C language structure that will store the data. That name is an instance-specific attribute of the `Struct` CSC.

Instance-specific attributes are stored in the `CoderInfo` property `CustomAttributes`. This property is initially defined as follows:

```
SimulinkCSC.AttribClass_Simulink_Default
1x1 struct array with no fields
```

When you specify a custom storage class, Simulink automatically populates `CoderInfo.CustomAttributes` with fields to represent instance-specific attributes of that CSC. For example, if you set the `MySig` CSC to `Struct`, as described in “Specify a Custom Storage Class” on page 9-43, then enter:

```
MyObj.CoderInfo.CustomAttributes
```

MATLAB displays:

```
SimulinkCSC.AttribClass_Simulink_Struct
  StructName: ''
```

To specify that `StructName` is `MyStruct`, enter:

```
MyObj.CoderInfo.CustomAttributes.StructName='MyStruct'
```

MATLAB displays:

```
SimulinkCSC.AttribClass_Simulink_Struct
  StructName: 'MyStruct'
```

Class Name	Instance-Specific Property	Purpose
BitField	<code>CustomAttributes.StructName</code>	Name of the bitfield struct into which the code generator packs the object's Boolean data.
ExportToFile	<code>CustomAttributes.HeaderFile</code>	Name of header (.h) file that contains exported variable declarations and export directives for the object.
GetSet	<code>CustomAttributes.HeaderFile</code>	Name of header (.h) file to <code>#include</code> in the generated code. See “GetSet Custom Storage Class” on page 9-53.
	<code>CustomAttributes.GetFunction</code>	String that specifies the name of a function call to read data.
	<code>CustomAttributes.SetFunction</code>	String that specifies the name of a function call to write data.

Class Name	Instance-Specific Property	Purpose
ImportedDefine	CustomAttributes.HeaderFile	The header file that defines the values of code variant preprocessor conditionals. See “Generate Preprocessor Conditionals for Variant Systems”.
ImportFromFile	CustomAttributes.HeaderFile	Name of header (.h) file containing global variable declarations the code generator imports for the object.
Struct	CustomAttributes.StructName	Name of the struct into which the code generator packs the object's data.

Assign Embedded Signal Objects to Output Ports

If you are operating on an embedded signal object with the API, you must copy the object to the port after providing or changing its `CoderInfo` properties. For example, if `MyObj` is a signal object that you want to copy to the output port, enter:

```
hps=get_param(gcb, 'PortHandles')
hp=hps.Output(1)
set_param(hp, 'SignalObject', 'MyObj')
```

Subsequent changes to the source object in the base workspace have no effect on the output port copy, and you can delete the source object:

```
clear ('MyObj')
```

Generate Code with Custom Storage Classes

In this section...

“Code Generation Prerequisites” on page 9-46

“Code Generation With Custom Storage Classes” on page 9-46

Code Generation Prerequisites

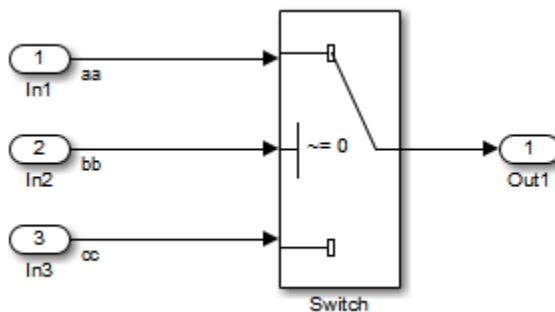
Before you generate code for a model that uses custom storage classes, set model options as follows:

- If your model assigns custom storage classes to parameters, select **Configuration Parameters > Optimization > Signals and Parameters > Inline parameters**. Otherwise, the code generator ignores CSC specifications for parameters. This requirement also applies to models that assign built-in storage classes to parameters.
- Clear **Configuration Parameters > Code Generation > Data specification override > Ignore custom storage classes**.

Otherwise, the code generator ignores CSC specifications and treats data objects as if their **Storage class** were Auto.

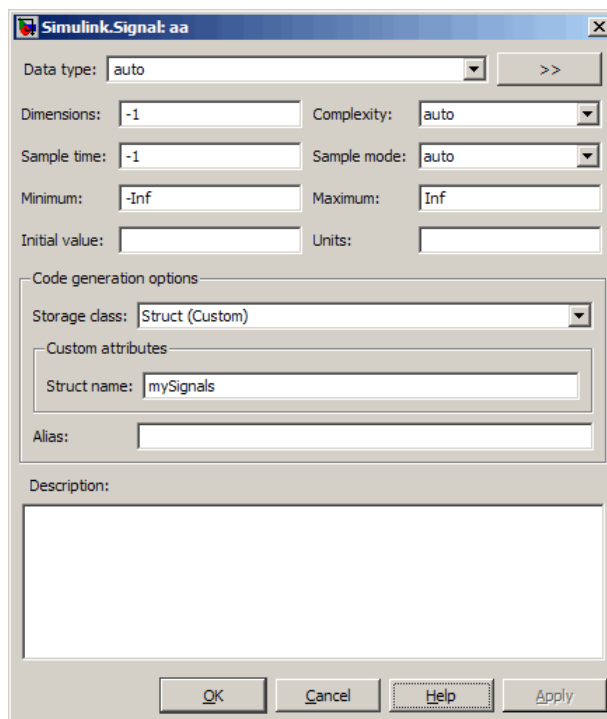
Code Generation With Custom Storage Classes

This section presents an example of code generation with CSCs, based on this model:



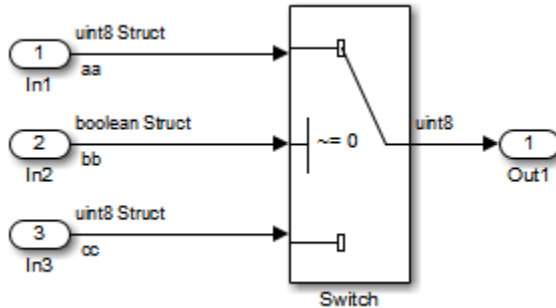
The model contains three named signals: **aa**, **bb**, and **cc**. Using the predefined **Struct** custom storage class, the example generates code that packs these signals into a **struct** named **mySignals**. The **struct** declaration is then exported to externally written code.

To specify the **struct**, you provide **Simulink.Signal** objects that specify the **Struct** custom storage class, and associate the objects with the signals as described in “Apply Custom Storage Classes” on page 9-29. The three objects have the same properties. This figure shows the signal object properties for **aa**:



The association between identically named model signals and signal objects is formed as described in “Symbol Resolution”. In this example, the symbols **aa**, **bb**, and **cc** resolve to the signal objects **aa**, **bb**, and **cc**, which have custom storage class **Struct**. In the generated code, storage for the three signals will be allocated within a **struct** named **mySignals**.

To display the storage class of the signals in the model, select **Display > Signals & Ports > Storage Class** in the Simulink editor. The figure below shows the block diagram with signal data types and signal storage classes displayed.



With the model configured as described in “Code Generation Prerequisites” on page 9-46, and the signal objects defined and associated with the signals, you can generate code that uses the custom storage classes to generate the desired data structure for the signals. After code generation, the relevant definitions and declarations are located in three files:

- *model_types.h* defines the following struct type for storage of the three signals:

```
typedef struct MySignals_tag {
    boolean_T bb;
    uint8_T aa;
    uint8_T cc;
} mySignals_type;
```

- *model.c* (or *.cpp*) defines the variable `mySignals`, as specified in the object's instance-specific `StructName` attribute. The variable is referenced in the code generated for the Switch block:

```
/* Definition for Custom Storage Class: Struct */

mySignals_type mySignals = {
    /* cc */
    FALSE,
    /* bb */
    0,
    /* aa */
    0
}
```

```
};  
...  
/* Switch: '<Root>/Switch1' */  
  if(mySignals.cc) {  
    rtb_Switch1 = mySignals.aa;  
  } else {  
    rtb_Switch1 = mySignals.bb;  
  }  
}
```

- `model.h` exports the `mySignals` Struct variable:

```
/* Declaration for Custom Storage Class: Struct */  
  
extern mySignals_type mySignals;
```

Grouped Custom Storage Classes

A custom storage class that results in multiple data objects being referenced with a single variable in the generated code, in the previous example, is called a *grouped custom storage class*. In the Simulink package, `Bitfield` and `Struct` (shown in the preceding example) are grouped CSCs. Data grouped by a CSC is referred to as *grouped data*.

Note: If you use a grouped custom storage class, you cannot specify its properties on an instance-specific basis. This is because a grouped custom storage class combines multiple pieces of data into a single data structure. Data in this structure must have the same properties such as **Header file**, **Data scope**, and **Data initialization**.

Define Advanced Custom Storage Classes Types

In this section...

“Introduction” on page 9-50

“Create Your Own Parameter and Signal Classes” on page 9-50

“Create Custom Attributes Classes for Custom Storage Classes” on page 9-50

“Write TLC Code for Custom Storage Classes” on page 9-51

“Register Custom Storage Class Definitions” on page 9-51

Introduction

Certain data layouts, such as nested structures, cannot be generated using the standard `Unstructured` and `FlatStructure` custom storage class types. You can define an *advanced custom storage class* if you want to generate other types of data. Creating advanced CSCs requires understanding TLC programming and using a special advanced mode of the Custom Storage Class Designer. This sections explain how to define advanced CSC types.

Create Your Own Parameter and Signal Classes

The first step is to create your own package containing classes derived from `Simulink.Parameter` or `Simulink.Signal`. This procedure is described in “Define Data Classes” in the Simulink documentation.

Create Custom Attributes Classes for Custom Storage Classes

If you have instance-specific properties that are relevant only to your CSC, you should create a *custom attributes class* for the package. A custom attributes class is a subclass of `Simulink.CustomStorageClassAttributes`. The name, type, and default value properties you set for the custom attributes class define the user view of instance-specific properties. For instructions, see “Define Data Classes” in the Simulink documentation.

For example, the `ExportToFile` custom storage class requires that you set the `CoderInfo.CustomAttributes.HeaderFile` property to specify a `.h` file used for exporting each piece of data. See “Simulink Package Custom Storage Classes” on page 9-6 for further information on instance-specific properties.

Note: If you rename or remove custom attributes, you may need to manually edit the `csc_registration` file for the associated package to remove references to the custom attributes that you renamed or removed.

Write TLC Code for Custom Storage Classes

The next step is to write TLC code that implements code generation for data of your new custom storage class. A template TLC file is provided for this purpose. To create your TLC code, follow these steps:

- 1 Create a `tlc` directory inside your package's `+directory` (if it does not already exist). The naming convention to follow is

`+PackageName/tlc`
- 2 Copy `TEMPLATE_v1.tlc` (or another CSC template) from `matlabroot/toolbox/rtw/targets/ecoder/csc_templates` into your `tlc` directory to use as a starting point for defining your custom storage class.
- 3 Write your TLC code, following the comments in the CSC template file. Comments describe how to specify code generation for data of your custom storage class (for example, how data structures are to be declared, defined, and whether they are accessed by value or by reference).

Alternatively, you can copy a custom storage class TLC file from another existing package as a starting point for defining your custom storage class.

Register Custom Storage Class Definitions

After you have created a package for your new custom storage class and written its associated TLC code, you must register your class definitions with the Custom Storage Class Designer, using its advanced mode.

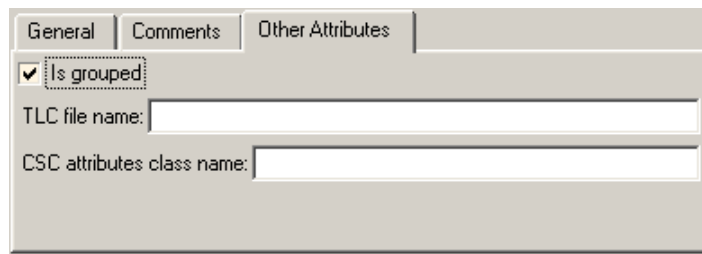
The advanced mode supports selection of an additional storage class **Type**, designated **Other**. The **Other** type is designed to support special CSC types that cannot be accommodated by the standard **Unstructured** and **FlatStructure** custom storage class types. The **Other** type cannot be assigned to a CSC except when the Custom Storage Class Designer is in advanced mode.

To register your class definitions:

- 1 Launch the Custom Storage Class Designer in advanced mode by typing the following command at the MATLAB prompt:

```
cscdesigner -advanced
```

- 2 Select your package and create a new custom storage class.
- 3 Set the **Type** of the custom storage class to **Other**. Note that when you do this, the **Other Attributes** pane is displayed. This pane is visible only for CSCs whose **Type** is set to **Other**.



If you specify a customized package, additional options, as defined by the package, also appear on the **Other Attributes** pane.

- 4 Set the properties shown on the **Other Attributes** pane. The properties are:
 - **Is grouped:** Select this option if you intend to combine multiple data objects of this CSC into a single variable in the generated code. (for example, a struct).
 - **TLC file name:** Enter the name of the TLC file corresponding to this custom storage class. The location of the file is assumed to be in the `/tlc` subdirectory for the package, so you should not enter the path to the file.
 - **CSC attributes class name:** (optional) If you created a custom attributes class corresponding to this custom storage class, enter the full name of the custom attributes class. (see “Create Custom Attributes Classes for Custom Storage Classes” on page 9-50).
- 5 Set the remaining properties on the **General** and **Comments** panes based on the layout of the data that you wish to generate (as defined in your TLC file).

GetSet Custom Storage Class

In this section...

- “About GetSet Custom Storage Class” on page 9-53
- “GetSet Custom Storage Class Properties” on page 9-53
- “Apply the GetSet Custom Storage” on page 9-54
- “GetSet Custom Storage Class Restrictions” on page 9-54
- “Increase Code Efficiency With GetSet CSC” on page 9-54

About GetSet Custom Storage Class

GetSet is a built-in advanced custom storage class that generates specialized function calls to read from (get) and write to (set) the memory associated with a Data Store Memory block that is read and written many times in a single model. See “Data Stores” for information about data stores and the Data Store Memory block, and “Define Advanced Custom Storage Classes Types” on page 9-50 for information about advanced CSCs.

The GetSet custom storage class is designed primarily for use with the state of the Data Store Memory block. However, GetSet is capable of handling signals other than data stores, and is supported for the outputs of most built-in blocks provided by MathWorks. For more about the definition of the GetSet storage class, look at its associated TLC code in the file:

```
matlabroot\toolbox\simulink\simulink\@Simulink\tlc\GetSet.tlc
```

GetSet Custom Storage Class Properties

The next table summarizes the instance-specific properties of the GetSet storage class:

Property	Description
GetFunction	String that specifies the name of a function call to read data.
SetFunction	String that specifies the name of a function call to write data.
HeaderFile (optional)	String that specifies the name of a header (.h) file to add as an #include in the generated code.

For example, if the `GetFunction` of signal `X` is specified as `'get_X'` then the generated code calls `get_X()` wherever the value of `X` is used. Similarly, if the `SetFunction` of signal `X` is specified as `'set_X'` then the generated code calls `set_X(value)` wherever the value of `X` is assigned.

Apply the GetSet Custom Storage

The `GetSet` storage class cannot be represented by the standard `Unstructured` or `FlatStructure` custom storage class types, so it is an advanced CSC, as described in “Define Advanced Custom Storage Classes Types” on page 9-50. To access the CSC definition for `GetSet`, you must launch Custom Storage Class designer in advanced mode:

```
cscdesigner -advanced
```

If you omit the `HeaderFile` property for a `GetSet` data object, you must specify a header file by an alternative means, such as the **Header file** field of the **Code Generation > Custom Code** pane of the Configuration Parameters dialog box. Otherwise, the generated code might not compile or might function improperly.

For wide signals, an additional index argument is passed, so the calls to the get and set functions are `get_X(idx)` and `set_X(idx, value)` respectively.

GetSet Custom Storage Class Restrictions

- The `GetSet` supports only signals of noncomplex data types.
- Some built-in blocks do not directly support `GetSet`.
- User-written S-functions do not directly support `GetSet`.

To use `GetSet` with a nonsupporting built-in block or a user-written S-function:

- 1 Insert a Signal Conversion block at the output of the block or function.
- 2 Select the Signal Conversion Block's **Exclude this block from 'Block reduction' optimization** property.
- 3 Assign the `GetSet` storage class to the output of the Signal Conversion block.

Increase Code Efficiency With GetSet CSC

The model below contains a Data Store Memory block that resolves to the Simulink signal object `X`:



The following specifications configure the signal object X to use the **GetSet** custom storage class:

```

X = Simulink.Signal;
X.CoderInfo.StorageClass           = 'Custom';
X.CoderInfo.CustomStorageClass    = 'GetSet';
X.CoderInfo.CustomAttributes.GetFunction = 'get_X';
X.CoderInfo.CustomAttributes.SetFunction = 'set_X';
X.CoderInfo.CustomAttributes.HeaderFile = 'user_file.h';
  
```

The **GetSet** CSC appears as follows in the code generated for the model:

```

/* Includes for objects with custom storage classes. */
#include "user_file.h"

void getset_csc_step(void)
{
  /* local block i/o variables */
  real_T rtb_DSRead_o;

  /* DataStoreWrite: '<Root>/DSWrite' incorporates:
   *   Inport: '<Root>/In1'
   */
  set_X(getset_csc_U.In1);

  /* DataStoreRead: '<Root>/DSRead' */
  rtb_DSRead_o = get_X();

  /* Outport: '<Root>/Out1' */
  getset_csc_Y.Out1 = rtb_DSRead_o;
}
  
```

Note that the code uses a local variable `rtb_DSRead_o` rather than multiple calls to the `get_X` function. This technique increases code efficiency and prevents changes to the value within a simulation step.

Custom Storage Class Implementation

You can skip this section unless you want to ship custom storage class definitions in an uneditable format, or you intend to bypass the Custom Storage Class designer and work directly with files that contain custom storage class definitions.

The file that defines a package's custom storage classes is called a *CSC registration file*. The file is named `csc_registration` and resides in the `+package` directory that defines the package. A CSC registration file can be a P-file (`csc_registration.p`) or a MATLAB file (`csc_registration.m`). A built-in package defines custom storage classes in both a P-file and a functionally equivalent MATLAB file. A user-defined package initially defines custom storage classes only in a MATLAB file.

P-files take precedence over MATLAB files, so when MATLAB looks for a package's CSC registration file and finds both a P-file and a MATLAB file, MATLAB loads the P-file and ignores the MATLAB file. The capabilities and tools, including the Custom Storage Class Designer, then use the CSC definitions stored in the P-file. P-files cannot be edited, so CSC Designer editing capabilities are disabled for CSCs stored in a P-file. If a P-file does not exist, MATLAB loads CSC definitions from the MATLAB file. MATLAB files are editable, so CSC Designer editing capabilities are enabled for CSCs stored in a MATLAB file.

Because CSC definitions for a built-in package exist in both a P-file and a MATLAB file, they are uneditable. You can make the definitions editable by deleting the P-file, but it is not recommended to modify built-in CSC registration files or other files under `matlabroot`. The preferred technique is to create packages, data classes, and custom storage classes, as described in “Define Data Classes” in the Simulink documentation.

The CSC Designer saves CSC definitions for user-defined packages in a MATLAB file, so the definitions are editable. You can make the definitions uneditable by using the `pcode` function to create an equivalent P-file, which will then shadow the MATLAB file. However, you should preserve the MATLAB file if you may need to make further changes, because you cannot modify CSC definitions that exist only in a P-file.

You can also use tools or techniques other than the Custom Storage Class Designer to create and edit MATLAB files that define CSCs. However, that practice is vulnerable to syntax errors and can give unexpected results. When MATLAB finds an older P-file that shadows a newer MATLAB file, it displays a warning in the MATLAB Command Window.

Custom Storage Class Limitations

- Data objects cannot have a CSC and a multi-word data type.
- The Fcn block does not support parameters with custom storage class in code generation.
- For CSCs in models that use referenced models:
 - If data is assigned a grouped CSC, such as **Struct** or **Bitfield**, the CSC's **Data scope** property must be **Imported** and the data declaration must be provided in a user-supplied header file. See “Grouped Custom Storage Classes” on page 9-49 for more information about grouped CSCs.
 - If data is assigned an ungrouped CSC, such as **Const**, and the data's **Data scope** property is **Exported**, its **Header file** property must be unspecified. This results in the data being exported with the standard header file, *model.h*. Note that for ungrouped data, the **Data scope** and **Header file** properties are either specified by the selected CSC, or as one of the data object's instance-specific properties.

User Package Registration

- “About Data Object Wizard and User Packages” on page 10-2
- “Register User Packages Using sl_customization.m” on page 10-3
- “User Package Customization Using sl_customization.m” on page 10-5

About Data Object Wizard and User Packages

Data Object Wizard (DOW) can be run in connection with a Simulink model to quickly determine which model data are not associated with data objects and to create and associate data objects with the data. (For more information about Data Object Wizard, see “Data Object Wizard” in the Simulink documentation and “Create Data Objects with Data Object Wizard”.) If you want the wizard to use data object classes from a package other than the standard Simulink class package to create the data objects, you select the package from the wizard's **Choose package for selected data objects** list. You can customize the package list by adding and removing packages and modifying the list order.

Note: User-defined packages that you add to the list must contain a `Simulink.Signal` subclass named `Signal` and a `Simulink.Parameter` subclass named `Parameter`.

To register Data Object Wizard user package customizations, use the Simulink customization file `sl_customization.m`. This file is a mechanism that allows you to use MATLAB code to perform customizations of the standard Simulink user interface. The Simulink software reads the `sl_customization.m` file, if present on the MATLAB path, when it starts and the customizations specified in the file are applied to the Simulink session. For more information on the `sl_customization.m` customization file, see “Registering Customizations”.

Register User Packages Using `sl_customization.m`

To register Data Object Wizard user package customizations, you create an instance of `sl_customization.m` and include it on the MATLAB path of the Simulink installation that you want to customize. The `sl_customization` function accepts one argument: a handle to a customization manager object. For example,

```
function sl_customization(cm)
```

As a starting point for your customizations, the `sl_customization` function must first get the default (factory) customizations, using the following assignment statement:

```
hObj = cm.slDataObjectCustomizer;
```

You then invoke methods to register your customizations. The customization manager object includes the following methods for registering DOW user package customizations:

- `addUserPackage(hObj, packageName)`

`addUserPackage(hObj, cellArrayOfStrings)`

Adds the specified user package(s) to the top of the package list, as displayed in the **Choose package for selected data objects** pull-down list in Data Object Wizard.

- `moveUserPackageToTop(hObj, packageName)`

Moves the specified user package to the top of the package list, as displayed in the **Choose package for selected data objects** pull-down list in Data Object Wizard.

- `moveUserPackageToEnd(hObj, packageName)`

Moves the specified user package to the end of the package list, as displayed in the **Choose package for selected data objects** pull-down list in Data Object Wizard.

- `removeUserPackage(hObj, packageName)`

Removes the specified user package from the package list.

- `setUserPackages(hObj, cellArrayOfStrings)`

Replaces the entire package list with a specified list of user packages.

Your instance of the `sl_customization` function should use these methods to register DOW user package customizations for your Simulink installation.

The Simulink software reads the `sl_customization.m` file when it starts. If you subsequently change the file, in order to use your changes, you must restart your Simulink session or enter the following command at the MATLAB command line:

```
sl_refresh_customizations
```

User Package Customization Using `sl_customization.m`

The `sl_customization.m` file shown in `sl_customization.m` for User Package Customizations uses the following methods:

- `addUserPackage` to add the user packages `ECoderDemos` and `SimulinkDemos` (present by default in the MATLAB path) to the top of the package list, as displayed in the **Choose package for selected data objects** pull-down list in Data Object Wizard

Note: Packages `ECoderDemos` and `SimulinkDemos` must contain a `Simulink.Signal` subclass named `Signal` and a `Simulink.Parameter` subclass named `Parameter`.

- `moveUserPackageToEnd` to move `SimulinkDemos` to the end of the package list

`sl_customization.m` for User Package Customizations

```
function sl_customization(cm)
% Register user customizations

% Get default (factory) customizations
hObj = cm.slDataObjectCustomizer;

% Add user packages
hObj.addUserPackage({'ECoderDemos', 'SimulinkDemos'});

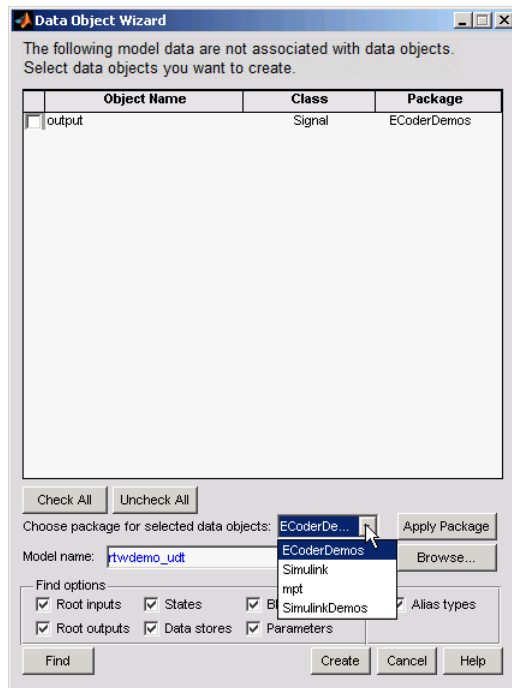
% Move SimulinkDemos to end of list
hObj.moveUserPackageToEnd('SimulinkDemos');

end
```

If you include the above file on the MATLAB path of the Simulink installation that you want to customize, the specified customizations will appear in Data Object Wizard. For example, you could view the customizations as follows:

- 1 Start a MATLAB session.
- 2 Launch a model, such as `rtwdemo_udt`.
- 3 Open Data Object Wizard, for example, by selecting **Code > Data Objects > Data Object Wizard** in the Simulink window.

- 4 In the Data Object Wizard dialog box, click the **Find** button to generate a list of one or more data objects.
- 5 Examine the **Choose package for selected data objects** drop-down list, noting the impact of the changes specified in `sl_customization.m` for User Package Customizations.



To replace the entire Data Object Wizard package list with a specified list of user packages, you can use a method invocation similar to the following:

```
hObj.setUserPackages({'myPackage1', 'ECoderDemos', 'mpt'});
```

Function and Class Interfaces

- “Function Prototype Control” on page 11-2
- “C++ Class Interface Control” on page 11-23
- “Atomic Subsystem Code” on page 11-50

Function Prototype Control

In this section...

“About Function Prototype Control” on page 11-2

“Configure Function Prototypes Using Graphical Interfaces” on page 11-3

“Sample Procedure for Configuring Function Prototypes” on page 11-11

“Configure Function Prototypes Programmatically” on page 11-16

“Sample Script for Configuring Function Prototypes” on page 11-20

“Verify Generated Code for Customized Functions” on page 11-20

“Function Prototype Control Limitations” on page 11-21

About Function Prototype Control

The Embedded Coder software provides a **Configure Model Functions** button, located on the **Code Generation > Interface** pane of the Configuration Parameters dialog box, that allows you to control the model function prototypes that are generated for ERT-based Simulink models.

By default, the function prototype of an ERT-based model's generated *model_step* function resembles the following:

```
void model_step(void);
```

The function prototype of an ERT-based model's generated *model_initialize* function resembles the following:

```
void model_initialize(void);
```

(For more detailed information about the default calling interface for the *model_step* function, see the *model_step* reference page.)

The **Configure Model Functions** button on the **Interface** pane provides you flexible control over the model function prototypes that are generated for your model. Clicking **Configure Model Functions** launches a Model Interface dialog box (see “Configure Function Prototypes Using Graphical Interfaces” on page 11-3). Based on the **Function specification** value you specify for your model function (supported values include `Default model initialize` and `step` functions and `Model specific C` prototypes), you can preview and modify the function prototypes. Once you validate

and apply your changes, you can generate code based on your function prototype modifications.

For more information about using the **Configure Model Functions** button and the Model Interface dialog box, see “Sample Procedure for Configuring Function Prototypes” on page 11-11 and the model `rtwdemo_fcncnprotctrl`, which is preconfigured to demonstrate function prototype control.

Alternatively, you can use function prototype control functions to programmatically control model function prototypes. For more information, see “Configure Function Prototypes Programmatically” on page 11-16.

You can also control model function prototypes for nonvirtual subsystems, if you generate subsystem code using right-click build. To launch the **Model Interface for subsystem** dialog box, use the `RTW.configSubsystemBuild` function.

Right-click building the subsystem generates the step and initialization functions according to the customizations you make. For more information, see “Configure Function Prototypes for Nonvirtual Subsystems” on page 11-8.

For limitations that apply, see “Function Prototype Control Limitations” on page 11-21.

Configure Function Prototypes Using Graphical Interfaces

- “Launch the Model Interface Dialog Boxes” on page 11-3
- “Default Model Initialize and Step Functions View” on page 11-4
- “Model Specific C Prototypes View” on page 11-5
- “Configure Function Prototypes for Nonvirtual Subsystems” on page 11-8

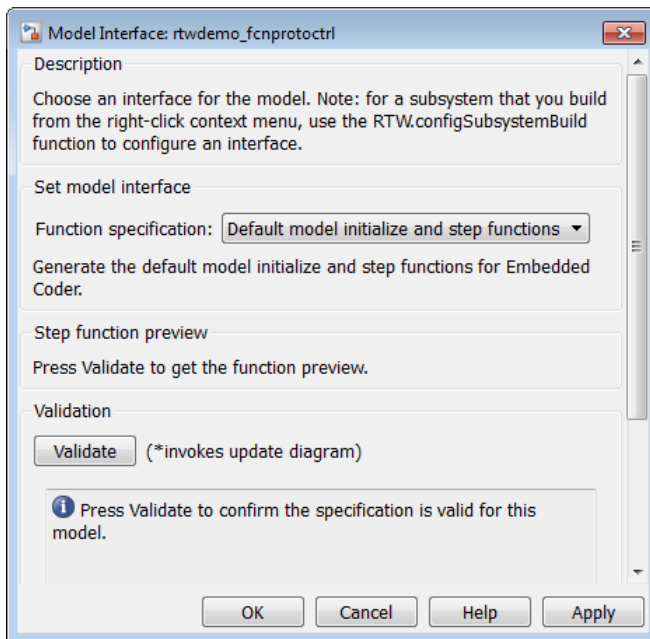
Launch the Model Interface Dialog Boxes

Clicking the **Configure Model Functions** button on the **Interface** pane of the Configuration Parameters dialog box launches the Model Interface dialog box. This dialog box is the starting point for configuring the model function prototypes that are generated during code generation for ERT-based Simulink models. Based on the **Function specification** value you select for your model function (supported values include `Default model initialize and step functions` and `Model specific C prototypes`), you can preview and modify the function prototype. Once you validate and apply your changes, you can generate code based on your function prototype modifications.

To configure function prototypes for a right-click build of a nonvirtual subsystem, invoke the `RTW.configSubsystemBuild` function, which launches the Model Interface for subsystem dialog box. For more information, see “Configure Function Prototypes for Nonvirtual Subsystems” on page 11-8

Default Model Initialize and Step Functions View

The figure below shows the Model Interface dialog box in the `Default model initialize and step functions` view.



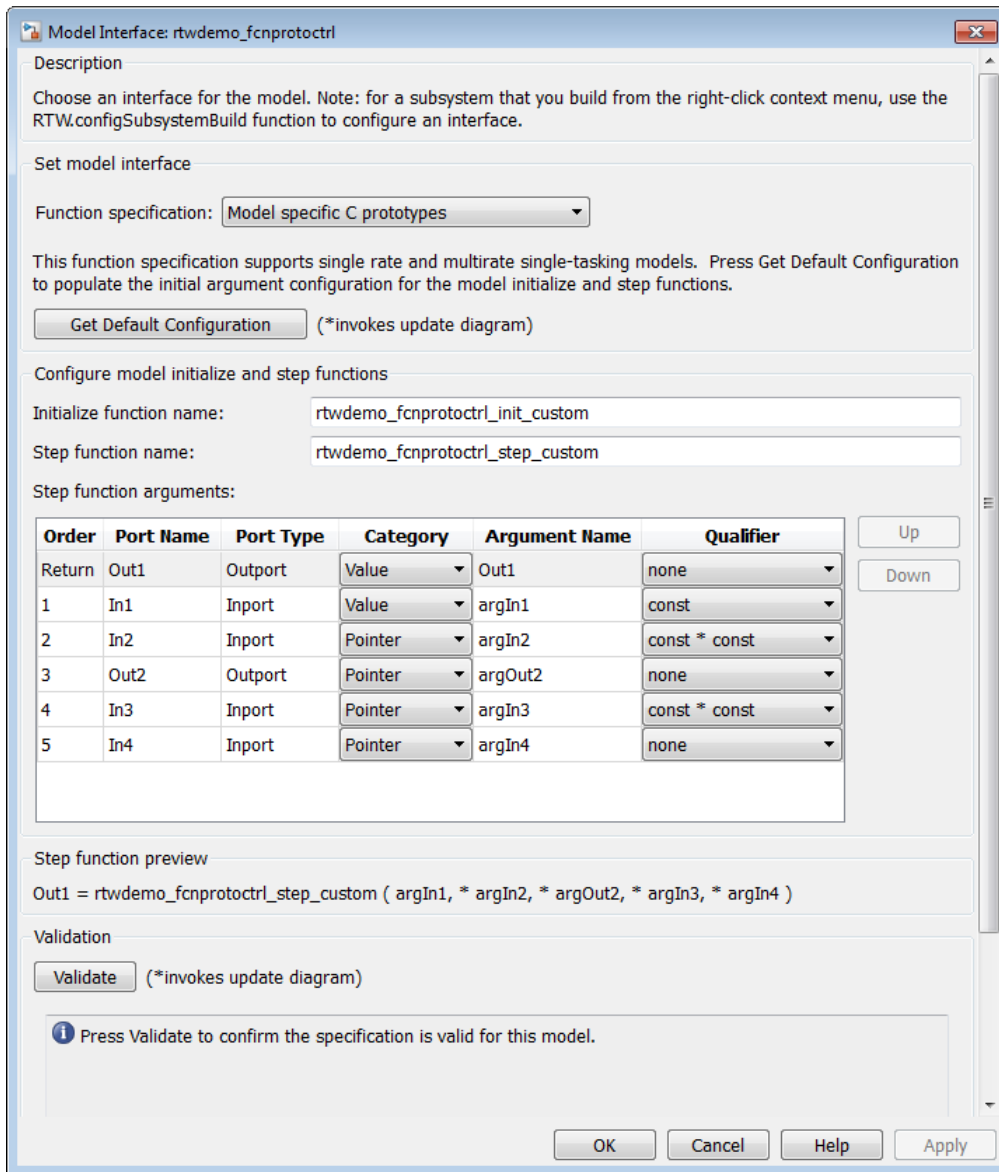
The `Default model initialize and step functions` view allows you to validate and preview the predicted default model step and initialization function prototypes. To validate the default function prototype configuration against your model, click the **Validate** button. If the validation succeeds, the predicted step function prototype appears in the **Step function preview** subpane.

Note: You cannot use the `Default model initialize and step functions` view to modify the function prototype configuration.

Model Specific C Prototypes View

Selecting **Model specific C prototypes** for the **Function specification** parameter displays the **Model specific C prototypes** view of your model function prototypes. This view provides controls that you can use to customize the function names, the order of arguments, and argument attributes including name, passing mechanism, and type qualifier for each of the model's root-level I/O ports.

To begin configuring your function control prototype configuration, click the **Get Default Configuration** button. This activates and initializes the function names and properties in the **Configure model initialize and step functions** subpane, as shown below. If you click **Get Default Configuration** again later, only the properties of the step function arguments are reset to default values.



In the **Configure model initialize and step functions** subpane:

Parameter	Description
Step function name	Name of the <i>model_step</i> function.
Initialize function name	Name of the <i>model_initialize</i> function. Note: This parameter only applies to the model initialize function. It does not apply to the model registration function for a referenced model.
Order	Order of the argument. A return argument is listed as Return .
Port Name	Name of the port.
Port Type	Type of the port.
Category	Specifies how an argument is passed in or out from the customized step function, either by copying a value (Value) or by a pointer to a memory space (Pointer).
Argument Name	Name of the argument.

Parameter	Description
Qualifier (optional)	<p>Specifies a const type qualifier for a function argument. The available values are dependent on the Category specified. When you change the Category, if the specified type is not available, the Qualifier changes to none. The possible values are:</p> <ul style="list-style-type: none"> • none • const (value) • const* (value referenced by the pointer) • const*const (value referenced by the pointer and the pointer itself) <hr/> <p>Note: When a model includes a referenced model, the const type qualifier for the root input argument of the referenced model's specified step function interface is set to none, and the qualifier for the source signal in the referenced model's parent is set to a value other than none, code generation honors the referenced model's interface specification by generating a type cast that discards the const type qualifier from the source signal. To override this behavior, add a const type qualifier to the referenced model.</p>

The **Step function preview** subpane provides a preview of how your step function prototype is interpreted in generated code. The preview is updated dynamically as you make modifications.

An argument `foo` whose **Category** is **Pointer** is previewed as `* foo`. If its **Category** is **Value**, it is previewed as `foo`. Notice that argument types and qualifiers are not represented in the **Step function preview** subpane.

Configure Function Prototypes for Nonvirtual Subsystems

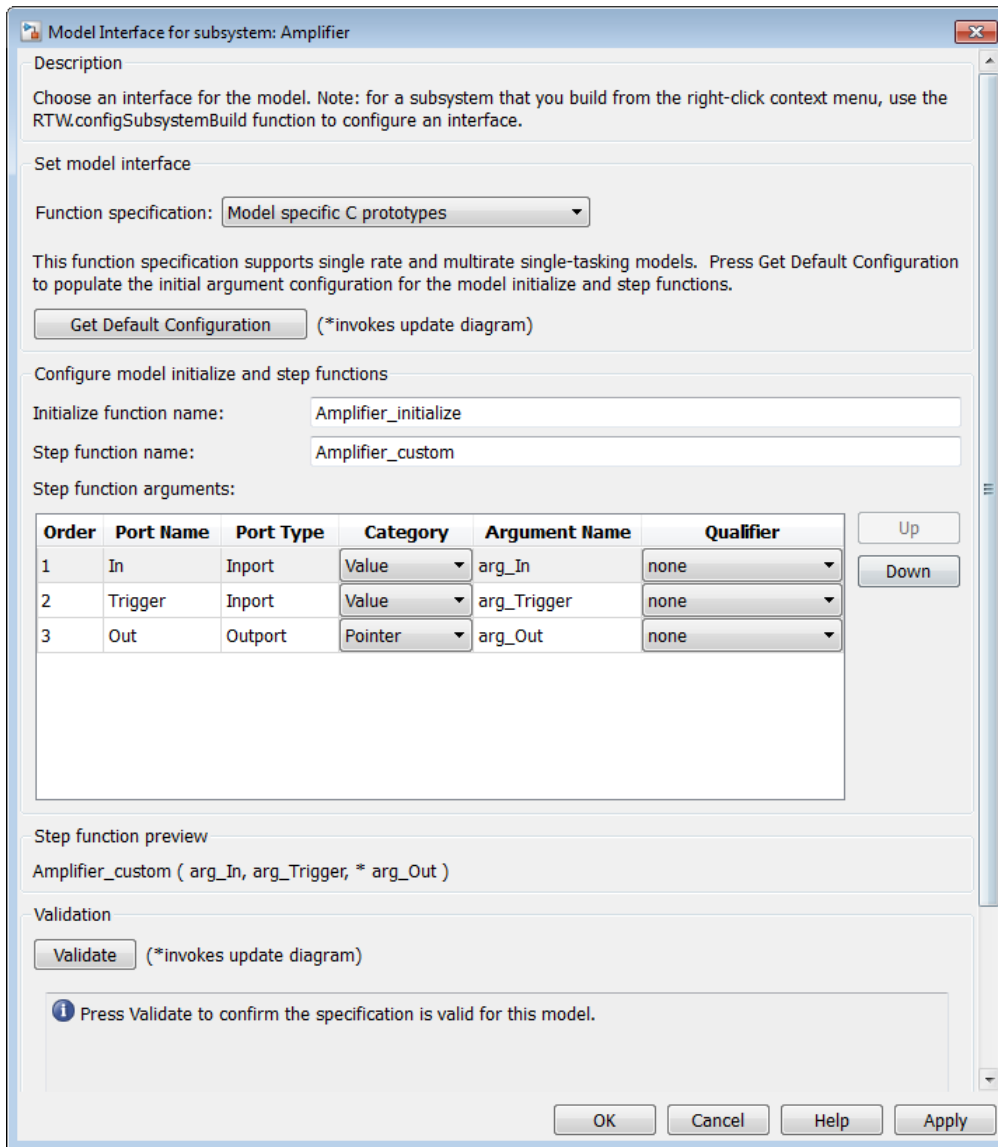
You can control step and initialization function prototypes for nonvirtual subsystems in ERT-based Simulink models, if you generate subsystem code using right-click build. Function prototype control is supported for the following types of nonvirtual blocks:

- Triggered subsystems

- Enabled subsystems
- Enabled trigger subsystems
- While subsystems
- For subsystems
- Stateflow blocks
- MATLAB function block

To launch the Model Interface for Subsystem dialog box, open the model containing the subsystem and invoke the `RTW.configSubsystemBuild` function.

The Model Interface dialog box for modifying the model-specific C prototypes for the `rtwdemo_counter/Amplifier` subsystem appears as follows:



Right-click building the subsystem generates the step and initialization functions according to the customizations you make.

Sample Procedure for Configuring Function Prototypes

The following procedure shows how to use the **Configure Model Functions** button on the **Code Generation > Interface** pane of the Configuration Parameters dialog box to control the model function prototypes when generating code for your Simulink model.

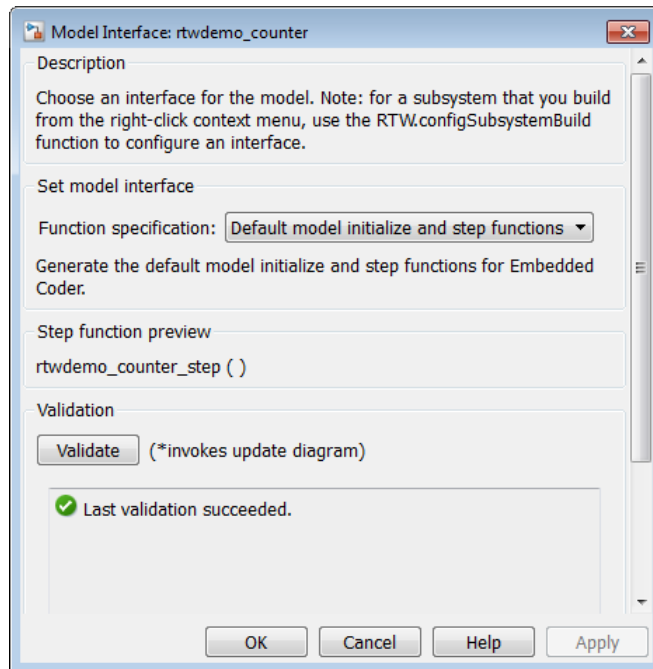
- 1 Open a MATLAB session and launch the `rtwdemo_counter` model.
- 2 In the `rtwdemo_counter` Model Editor, double-click the **Generate Code Using Embedded Coder (double-click)** button to generate code for an ERT-based version of `rtwdemo_counter`. The code generation report for `rtwdemo_counter` appears.
- 3 In the code generation report, click the link for `rtwdemo_counter.c`.
- 4 In the `rtwdemo_counter.c` code display, locate and examine the generated code for the `rtwdemo_counter_step` and the `rtwdemo_counter_initialize` functions:

```
/* Model step function */
void rtwdemo_counter_step(void)
{
    ...
}

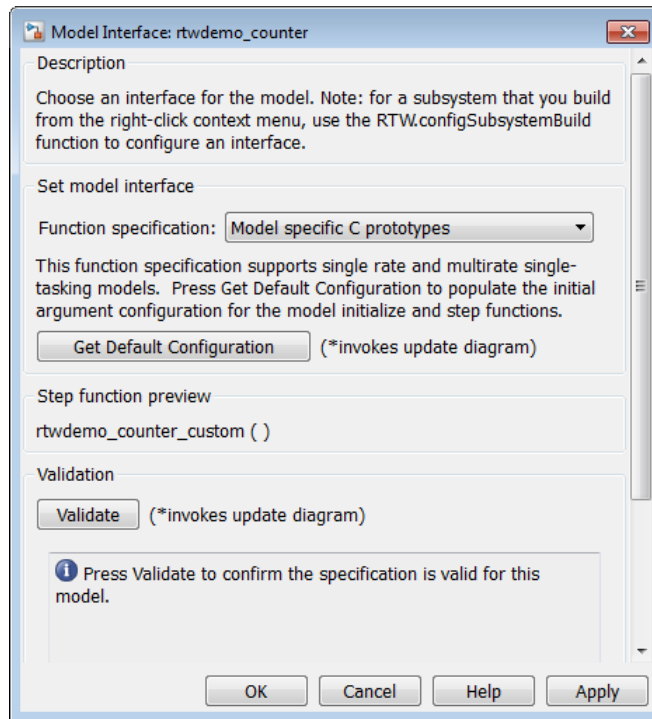
/* Model initialize function */
void rtwdemo_counter_initialize(void)
{
    ...
}
```

You can close the report window after you have examined the generated code. Optionally, you can save `rtwdemo_counter.c` and other generated files to a different location for later comparison.

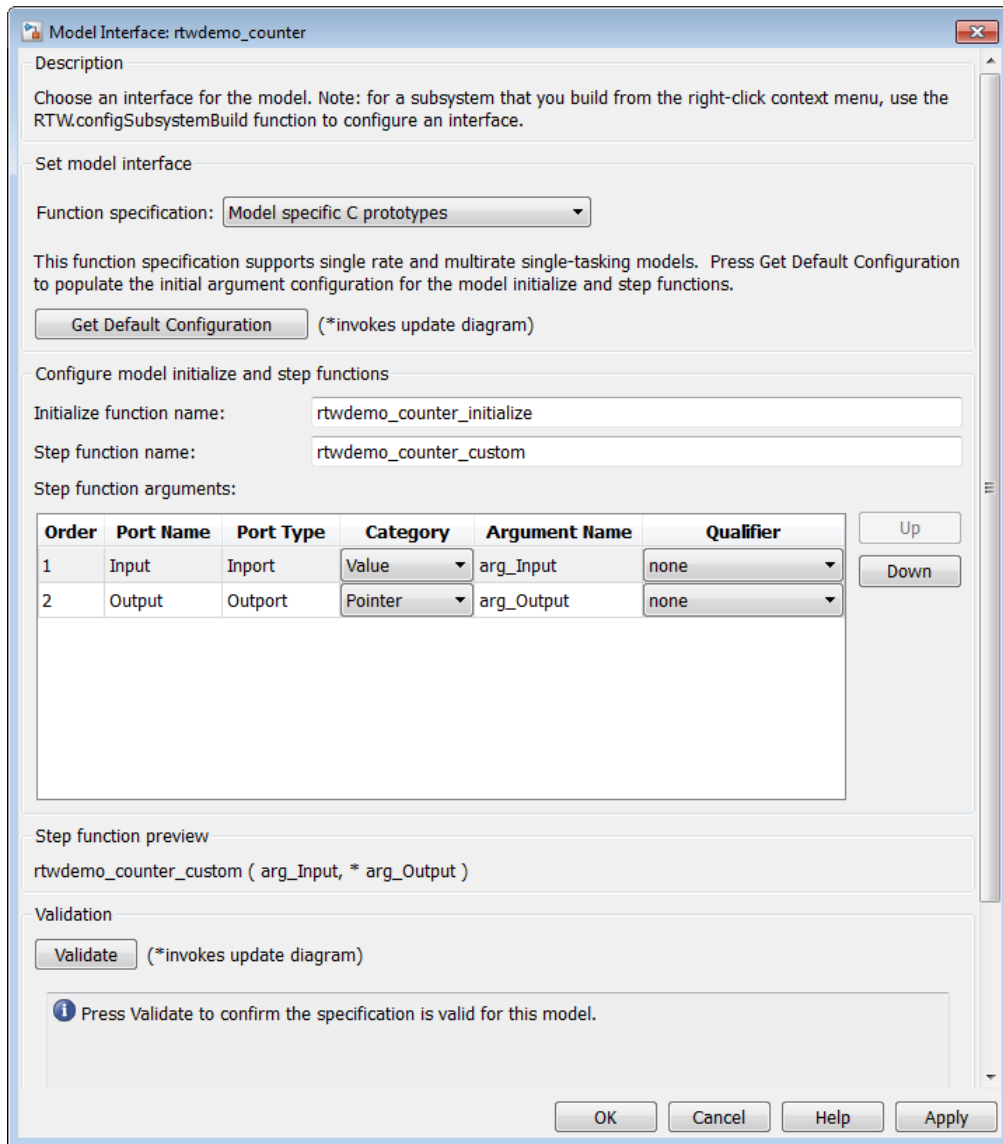
- 5 From the `rtwdemo_counter` model, open the Configuration Parameters dialog box.
- 6 Navigate to the **Code Generation > Interface** pane and click the **Configure Model Functions** button. The Model Interface dialog box appears.
- 7 In the initial (Default model initialize and step functions) view of the Model Interface dialog box, click the **Validate** button to validate and preview the default function prototype for the `rtwdemo_counter_step` function. The function prototype arguments under **Step function preview** should correspond to the default prototype in step 4.



- 8 In the Model Interface dialog box, set **Function specification** field to **Model specific C prototypes**. Making this selection switches the dialog box from the **Default model initialize and step functions** view to the **Model specific C prototypes** view.

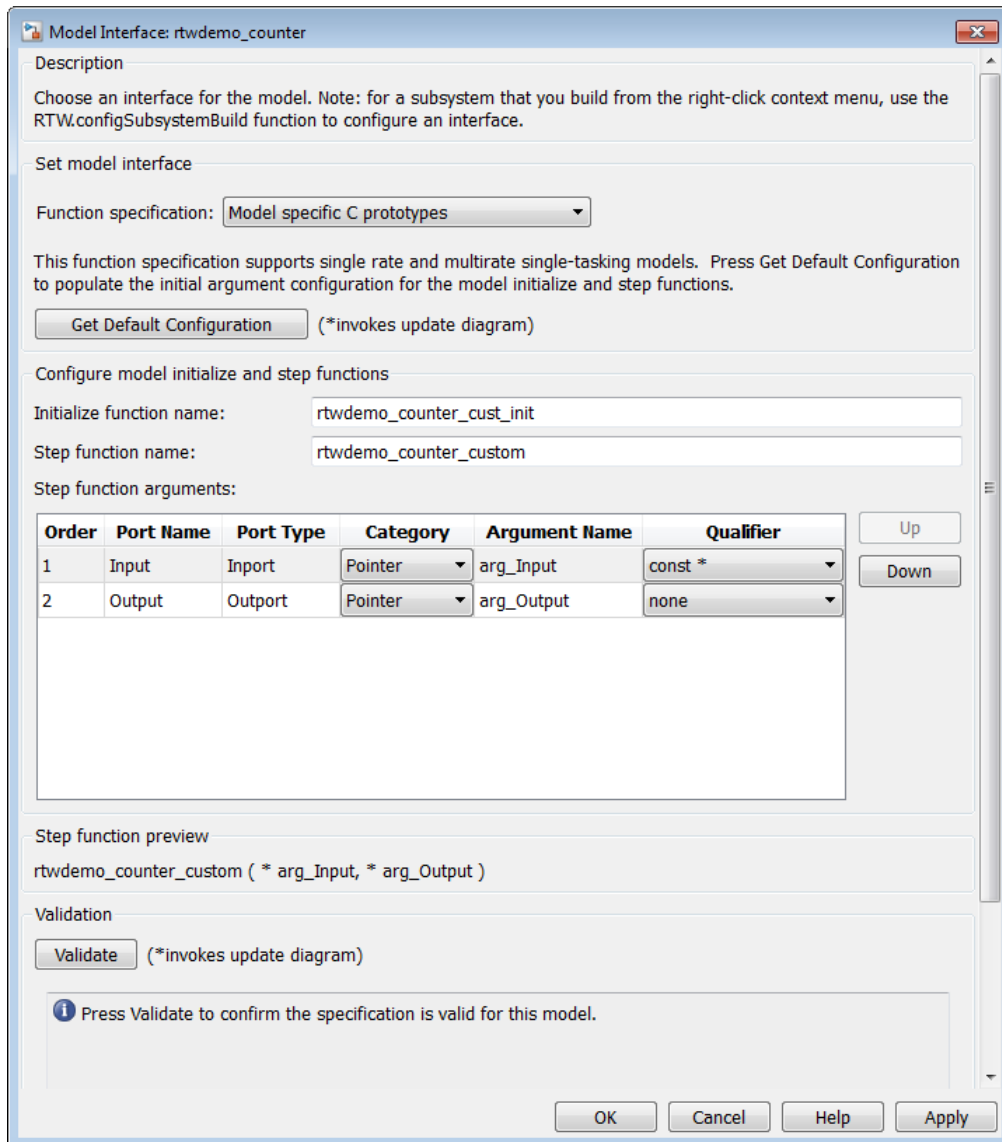


- 9 In the Model specific C prototypes view, click the **Get Default Configuration** button to activate the **Configure model initialize and step functions** subpane.



10 In the **Configure model initialize and step functions** subpane, change **Initialize function name** to `rtwdemo_counter_cust_init`.

- 11 In the **Configure model initialize and step functions** subpane, in the row for the **Input** argument, change the value of **Category** from **Value** to **Pointer** and change the value of **Qualifier** from **none** to **const ***. The preview reflects your changes.



- 12 Click the **Validate** button to validate the modified function prototype. The **Validation** subpane displays a message that the validation succeeded.
- 13 Click **OK** to exit the Model Interface dialog box.
- 14 Generate code for the model. When the build completes, the code generation report for `rtwdemo_counter` appears.
- 15 In the code generation report, click the link for `rtwdemo_counter.c`.
- 16 Locate and examine the generated code for the `rtwdemo_counter_custom` and `rtwdemo_counter_cust_init` functions:

```
/* Model step function */
void rtwdemo_counter_custom(const int32_T *arg_Input, int32_T *arg_Output)
{
    ...
}

/* Model initialize function */
void rtwdemo_counter_cust_init(void)
{
    ...
}
```

- 17 Verify that the generated code is consistent with the function prototype modifications that you specified in the Model Interface dialog box.

Configure Function Prototypes Programmatically

You can use the function prototype control functions (listed in Function Prototype Control Functions), to programmatically control model function prototypes. Typical uses of these functions include:

- **Create and validate a new function prototype**
 - 1 Create a model-specific C function prototype with `obj = "RTW.ModelSpecificCPrototype"`, where `obj` returns a handle to a newly created, empty function prototype.
 - 2 Add argument configuration information for your model ports using `RTW.ModelSpecificCPrototype.addArgConf`.
 - 3 Attach the function prototype to your loaded ERT-based Simulink model using `RTW.ModelSpecificCPrototype.attachToModel`.
 - 4 Validate the function prototype using `RTW.ModelSpecificCPrototype.runValidation`.

- 5 If validation succeeds, save your model and then generate code using the `rtwbuild` function.

- **Modify and validate an existing function prototype**

- 1 Get the handle to an existing model-specific C function prototype that is attached to your loaded ERT-based Simulink model with `obj = RTW.getFunctionSpecification(modelName)`, where `modelName` is a string specifying the name of a loaded ERT-based Simulink model, and `obj` returns a handle to a function prototype attached to the specified model.

You can use other function prototype control functions on the returned handle only if the test `isa(obj, 'RTW.ModelSpecificCPrototype')` returns 1. If the model does not have a function prototype configuration, the function returns []. If the function returns a handle to an object of type `RTW.FcnDefault`, you cannot modify the existing function prototype.

- 2 Use the `Get` and `Set` functions listed in Function Prototype Control Functions to test and reset such items as the function names, argument names, argument positions, argument categories, and argument type qualifiers.
- 3 Validate the function prototype using `RTW.ModelSpecificCPrototype.runValidation`.
- 4 If validation succeeds, save your model and then generate code using the `rtwbuild` function.

- **Create and validate a new function prototype, starting with default configuration information from your Simulink model**

- 1 Create a model-specific C function prototype using `obj = "RTW.ModelSpecificCPrototype"`, where `obj` returns a handle to a newly created, empty function prototype.
- 2 Attach the function prototype to your loaded ERT-based Simulink model using `RTW.ModelSpecificCPrototype.attachToModel`.
- 3 Get default configuration information from your model using `RTW.ModelSpecificCPrototype.getDefaultConf`.
- 4 Use the `Get` and `Set` functions listed in Function Prototype Control Functions to test and reset such items as the function names, argument names, argument positions, argument categories, and argument type qualifiers.
- 5 Validate the function prototype using `RTW.ModelSpecificCPrototype.runValidation`.

- 6 If validation succeeds, save your model and then generate code using the `rtwbuild` function.
- **Reset the model function prototype to the default ERT function configuration** Create an object of the ERT default function signature. Reset the model function prototype and undo any custom settings, by calling the `RTW.FcnDefault` method, `attachToModel`, as follows:

```
obj = RTW.FcnDefault;
obj.attachToModel(model);
model must be a loaded ERT-based model.
```

Note: You should not use the same model-specific C function prototype object across multiple models. If you do, changes that you make to the step and initialization function prototypes in one model are propagated to other models, which is usually not desirable.

Function Prototype Control Functions

Function	Description
<code>RTW.ModelSpecificCPrototype.addArgConf</code>	Add step function argument configuration information for Simulink model port to model-specific C function prototype
<code>RTW.ModelSpecificCPrototype.attachToModel</code>	Attach model-specific C function prototype to loaded ERT-based Simulink model
<code>RTW.ModelSpecificCPrototype.getArgCategory</code>	Get step function argument category for Simulink model port from model-specific C function prototype
<code>RTW.ModelSpecificCPrototype.getArgName</code>	Get step function argument name for Simulink model port from model-specific C function prototype
<code>RTW.ModelSpecificCPrototype.getArgPosition</code>	Get step function argument position for Simulink model port from model-specific C function prototype
<code>RTW.ModelSpecificCPrototype.getArgQualifier</code>	Get step function argument type qualifier for Simulink model port from model-specific C function prototype

Function	Description
RTW.ModelSpecificCPrototype.getDefaultCon	Get default configuration information for model-specific C function prototype from Simulink model to which it is attached
RTW.ModelSpecificCPrototype.getFunctionNa	Get function names from model-specific C function prototype
RTW.ModelSpecificCPrototype.getNumArgs	Get number of step function arguments from model-specific C function prototype
RTW.ModelSpecificCPrototype.getPreview	Get model-specific C function prototype code previews
RTW.configSubsystemBuild	Launch GUI to configure C function prototype or C++ class interface for right-click build of specified subsystem
RTW.getFunctionSpecification	Get handle to model-specific C function prototype object
RTW.ModelSpecificCPrototype.runValidation	Validate model-specific C function prototype against Simulink model to which it is attached
RTW.ModelSpecificCPrototype.setArgCategor	Set step function argument category for Simulink model port in model-specific C function prototype
RTW.ModelSpecificCPrototype.setArgName	Set step function argument name for Simulink model port in model-specific C function prototype
RTW.ModelSpecificCPrototype.setArgPositio	Set step function argument position for Simulink model port in model-specific C function prototype
RTW.ModelSpecificCPrototype.setArgQualifi	Set step function argument type qualifier for Simulink model port in model-specific C function prototype
RTW.ModelSpecificCPrototype.setFunctionNa	Set function names in model-specific C function prototype

Sample Script for Configuring Function Prototypes

The following sample MATLAB script configures the model function prototypes for the `rtwdemo_counter` model, using the Function Prototype Control Functions.

```
%% Open the rtwdemo_counter model
rtwdemo_counter

%% Select ert.tlc as the System Target File for the model
set_param(gcs,'SystemTargetFile','ert.tlc')

%% Create a model-specific C function prototype
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 model-specific C function prototype to the model
attachToModel(a,gcs)

%% Rename the initialization function
setFunctionName(a,'InitFunction','init')

%% Rename the step function and change some argument attributes
setFunctionName(a,'StepFunction','step')
setArgPosition(a,'Output',1)
setArgCategory(a,'Input','Value')
setArgName(a,'Input','InputArg')
setArgQualifier(a,'Input','none')

%% Validate the function prototype against the model
[status,message]=runValidation(a)

%% if validation succeeded, generate code and build
if status
    rtwbuild(gcs)
end
```

Verify Generated Code for Customized Functions

You can use software-in-the-loop (SIL) testing to verify the generated code for your customized step and initialization functions. This involves creating a SIL block with your generated code, which then can be integrated into a Simulink model to verify that the generated code provides the same result as the original model or nonvirtual subsystem. For more information, see “Choose a SIL or PIL Approach”.

Function Prototype Control Limitations

The following limitations apply to controlling model function prototypes:

- Function prototype control supports only step and initialization functions generated from a Simulink model.
- Function prototype control supports only single-instance implementations. For standalone targets, you must set **Code interface packaging** to **Nonreusable function** (on the **Code Generation > Interface** pane of the Configuration Parameters dialog box). For model reference targets, you must select **One** for the **Total number of instances allowed per top model** parameter (on the **Model Referencing** pane of the Configuration Parameters dialog box).
- For model reference targets, if **Code interface packaging** is set to **Reusable function**, the code generator ignores the setting.
- You must select the **Single output/update function** parameter (on the **Interface** pane of the Configuration Parameters dialog box).
- Function prototype control does not support multitasking models. Multirate models are supported, but you must configure the models for single-tasking.
- You must configure root-level inports and outports to use **Auto** storage classes.
- Do not control function prototypes with the static `ert_main.c` provided by MathWorks. Specifying a function prototype control configuration other than the default creates a mismatch between the generated code and the default static `ert_main.c`.
- The code generator removes the data structure for the root inports of the model unless a subsystem implemented by a nonreusable function uses the value of one or more of the inports.
- The code generator removes the data structure for the root outports of the model except when you enable MAT-file logging, or if the sample time of one or more of the outports is not the fundamental base rate (including a constant rate).
- If you copy a subsystem block and paste it to create a new block in either a new model or the same model, the function prototype control interface information from the original subsystem block does not copy to the new subsystem block.
- If you have a Stateflow license, for a Stateflow chart that uses a model root inport value, or that calls a subsystem that uses a model root inport value, you must do one of the following to generate code:

- Clear the **Execute (enter) Chart At Initialization** check box in the Stateflow chart.
- Make the Stateflow function a nonreusable function.
- Insert a Simulink Signal Conversion block immediately after the root inport. In the Signal Conversion block parameters dialog box, select **Exclude this block from 'Block reduction' optimization**.
- If a model root inport value connects to a Simscape conversion block, you must insert a Simulink Signal Conversion block between the root inport and the Simscape conversion block. In the Signal Conversion block parameters dialog box, select **Exclude this block from 'Block reduction' optimization**.

C++ Class Interface Control

In this section...

- “About C++ Class Interface Control” on page 11-23
- “Simple Use of C++ Class Control” on page 11-24
- “Customize C++ Class Interfaces Using Graphical Interfaces” on page 11-30
- “Customize C++ Class Interfaces Programmatically” on page 11-42
- “Configure Step Method for Model Class” on page 11-45
- “Specify Custom Storage Class for C++ Class Code Generation” on page 11-46
- “Model Class Copy Constructor and Assignment Operator” on page 11-47
- “C++ Class Interface Control Limitations” on page 11-48

About C++ Class Interface Control

Using the **Code interface packaging** option `C++ class`, on the **Code Generation > Interface** pane of the Configuration Parameters dialog box, you can generate a C++ class interface to model code. The generated interface encapsulates required model data into C++ class attributes and model entry point functions into C++ class methods. The benefits of C++ class encapsulation include:

- Greater control over access to model data
- Ability to multiply instantiate model classes
- Easier integration of model code into C++ programming environments

C++ class encapsulation also works for right-click builds of nonvirtual subsystems. (For information on requirements that apply, see “Configure C++ Class Interfaces for Nonvirtual Subsystems” on page 11-42.)

If you have an Embedded Coder license and you have selected an ERT target for your model, you can use additional **Code Generation > Interface** pane parameters in the Configuration Parameter dialog box to customize and control the generated C++ class interface to model code. The general procedure for generating custom C++ class interfaces to model code is as follows:

- 1 Configure your model to use an `ert.tlc` system target file provided by MathWorks.
- 2 Select the C++ language for your model.

- 3 Select `C++ class` code interface packaging for your model.
- 4 Customize C++ class interface settings for your model code, using either a graphical user interface (GUI) or application programming interface (API).
- 5 Generate model code.
- 6 Examine the C++ class interfaces in the generated files and the HTML code generation report.

To get started with an example, see “Simple Use of C++ Class Control” on page 11-24. For more details about customizing C++ class interfaces for your model code, see “Customize C++ Class Interfaces Using Graphical Interfaces” on page 11-30 and “Customize C++ Class Interfaces Programmatically” on page 11-42. For limitations that apply, see “C++ Class Interface Control Limitations” on page 11-48.

Note: For an example of C++ class code generation, see the example model `rtwdemo_cppclass`.

Simple Use of C++ Class Control

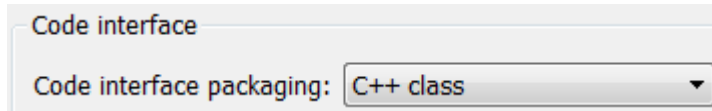
This example illustrates a simple use of `C++ class` code interface packaging. It generates C+ class code interfaces from an example model, without extensive modifications to default settings.

Note: For details about setting C++ class parameters, see the sections that follow this example, beginning with “Customize C++ Class Interfaces Using Graphical Interfaces” on page 11-30.

To generate C++ class interfaces for a Simulink model:

- 1 Open a model for which you would like to generate C++ class code interfaces. This example uses the model `rtwdemo_counter`.
- 2 Configure the model to use an `ert.tlc` system target file provided by MathWorks. For example, open the Configuration Parameters dialog box, go to the **Code Generation** pane, select a target value from the **System target file** menu, and click **Apply**.
- 3 On the **Code Generation** pane of the Configuration Parameters dialog box, set the **Language** parameter to `C++`.

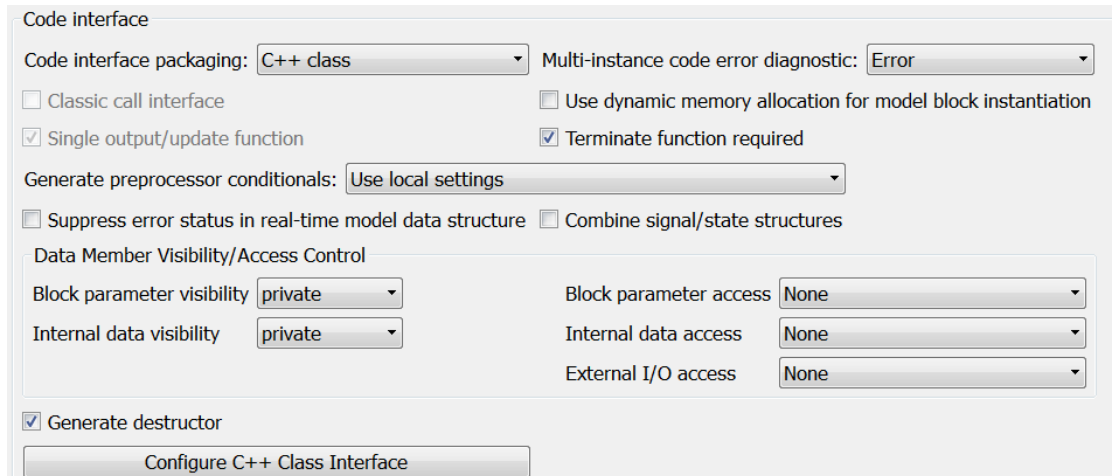
On the **Code Generation > Interface** pane, check that the **Code interface packaging** parameter is set to **C++ class**.



Click **Apply**.

Note: To immediately generate the default style of C++ class code, without exploring the related model configuration options, skip steps 4–8 and go directly to step 9.

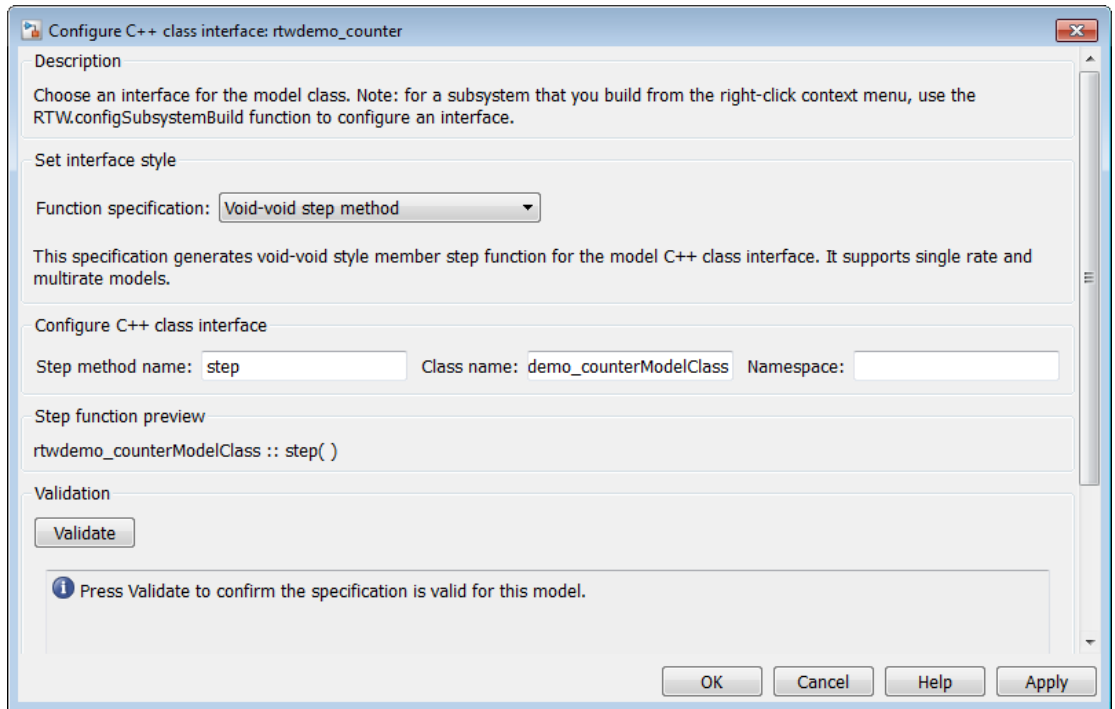
- 4 Go to the **Interface** pane of the Configuration Parameters dialog box and examine the **Code interface** subpane.



When you select **C++ class** code interface packaging for your model, additional C++ class interface controls become available in the **Code interface** subpane. See “Configure Code Interface Options” on page 11-31 for descriptions of these controls. You might want to modify the default settings according to your application.

- 5 Click the **Configure C++ Class Interface** button. This action opens the Configure C++ class interface dialog box, which allows you to configure the step method for your generated model class. The dialog box initially displays a view for configuring a

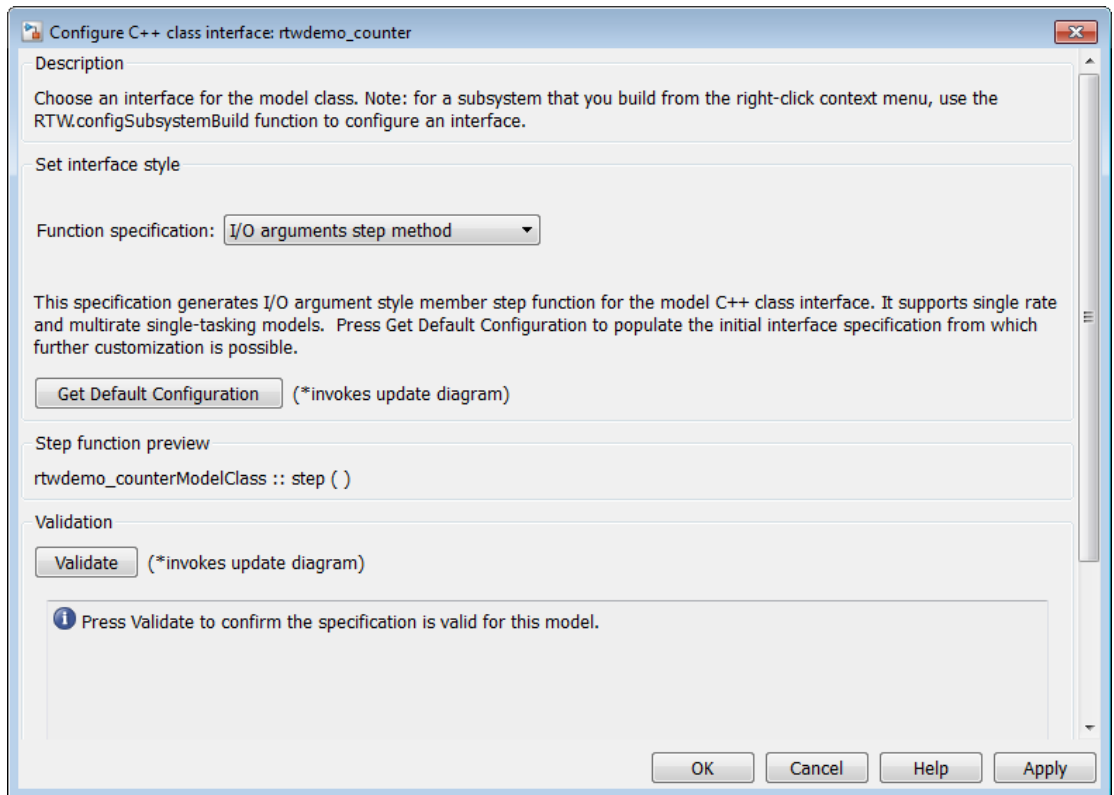
void-void style step method (passing no I/O arguments) for the model class. In this view, you can specify the model class name, step method name, and namespace for your model.



See “Configure Step Method for Your Model Class” on page 11-35 for descriptions of these controls.

Note: If the void-void interface style meets your needs, you can skip steps 6–8 and go directly to step 9.

- 6 If you want root-level model input and output to be arguments on the step method, select the value **I/O arguments step method** from the **Function specification** menu. The dialog box displays a view for configuring an I/O arguments style step method for the model class.



See “Configure Step Method for Your Model Class” on page 11-35 for descriptions of these controls.

- 7 Click the **Get Default Configuration** button. This action causes a **Configure C++ class interface** subpane to appear in the dialog box. The subpane displays the initial interface configuration for your model, which provides a starting point for further customization.

Configure C++ class interface

Step method name: Class name: Namespace:

Order	Port Name	Port Type	Category	Argument Name	Qualifier
1	Input	Inport	Value	arg_Input	none
2	Output	Output	Pointer	arg_Output	none

Up
Down

- See “Passing I/O Arguments” on page 11-37 for descriptions of these controls.
- 8** Perform this optional step only if you want to customize the configuration of the I/O arguments generated for your model step method.

Note: If you choose to skip this step, you should click **Cancel** to exit the dialog box.

If you choose to perform this step, first you must check that the required option **Remove root level I/O zero initialization** is selected on the **Optimization** pane, and then navigate back to the I/O arguments step method view of the Configure C++ class interface dialog box.

Now you can use the dialog box controls to configure I/O argument attributes. For example, in the **Configure C++ class interface** subpane, in the row for the **Input** argument, you can change the value of **Category** from **Value** to **Pointer** and change the value of **Qualifier** from **none** to **const ***. The preview updates to reflect your changes. Click the **Validate** button to validate the modified interface configuration.

Continue modifying and validating until you are satisfied with the step method configuration.

Configure C++ class interface

Step method name: Class name: Namespace:

Order	Port Name	Port Type	Category	Argument Name	Qualifier
1	Input	Inport	Pointer	arg_Input	const *
2	Output	Output	Pointer	arg_Output	none

Up


Down

Step function preview

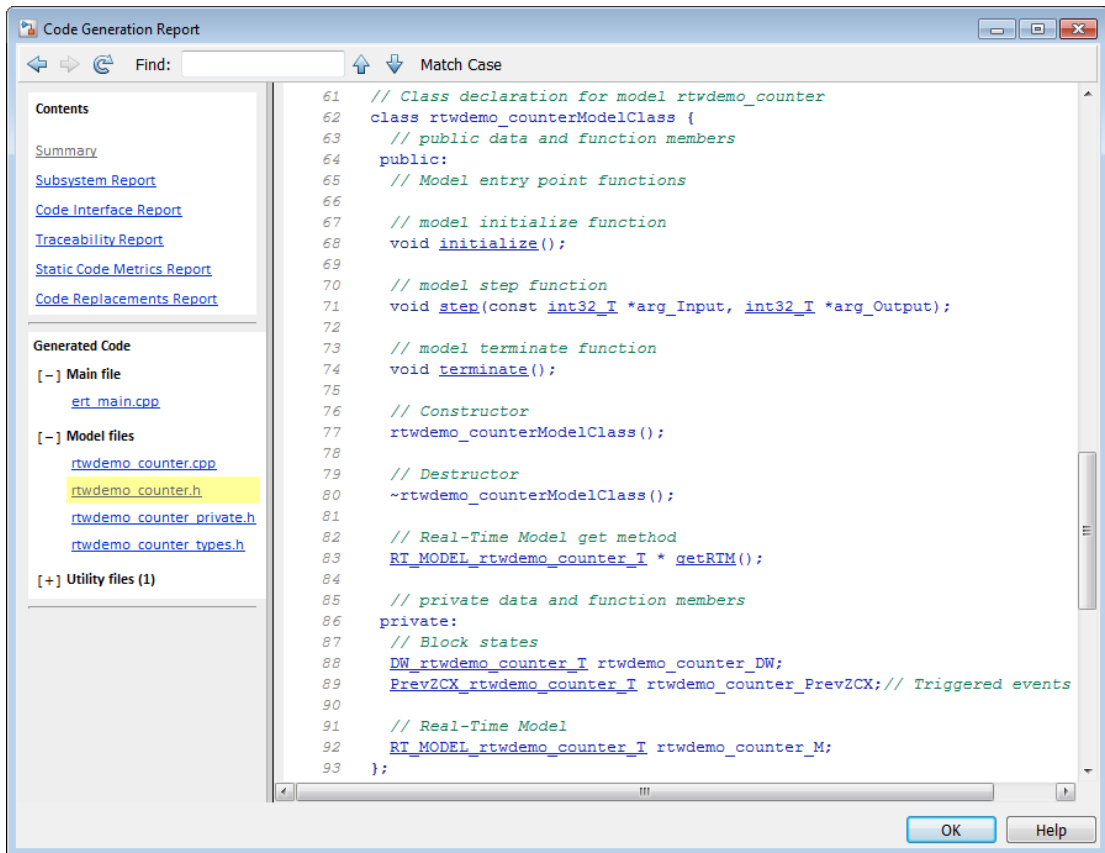
```
rtwdemo_counterModelClass :: step ( * arg_Input, * arg_Output )
```

Validation

(*invokes update diagram)

 Last validation succeeded.

- Click **Apply** and **OK**.
- 9 Generate code for the model. When the build completes, the code generation report for `rtwdemo_counter` appears. Examine the report and observe that required model data is encapsulated into C++ class attributes and model entry point functions are encapsulated into C++ class methods. For example, click the link for `rtwdemo_counter.h` to see the class declaration for the model.



Note: If you configured custom I/O arguments for the model step method (optional step 8), examine the generated code for the step method in `rtwdemo_counter.h` and `rtwdemo_counter.cpp`. The arguments should reflect your changes. For example, if you performed the Input argument modifications in step 8, the input argument should appear as `const int32_T *arg_Input`.

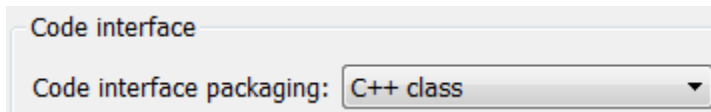
Customize C++ Class Interfaces Using Graphical Interfaces

- “Select C++ Class Code Interface Packaging” on page 11-31
- “Configure Code Interface Options” on page 11-31

- “Configure Step Method for Your Model Class” on page 11-35
- “Use Namespaces to Scope C++ Model Classes” on page 11-40
- “Configure C++ Class Interfaces for Nonvirtual Subsystems” on page 11-42

Select C++ Class Code Interface Packaging

To select `C++ class` code interface packaging, in the Configuration Parameters dialog box, on the **Code Generation** pane, set the **Language** parameter to `C++`. Then, in the **Code Generation > Interface** pane, check that the **Code interface packaging** parameter is set to `C++ class`:



Selecting this value:

- Disables model configuration options that `C++ class` does not support. For details, see “C++ Class Interface Control Limitations” on page 11-48.
- Adds additional C++ class interface parameters, which are described in the next section.

Configure Code Interface Options

When you select `C++ class` code interface packaging for your model, the **Code interface** parameters shown below are displayed on the **Interface** pane.

- **Multi-instance code error diagnostic**

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

- None — Proceed with build without displaying a diagnostic message.
- Warning — Proceed with build after displaying a warning message.
- Error (default) — Abort build after displaying an error message.

- **Terminate function required**

Specifies whether to generate the `model_terminate` method (on by default). This function contains model termination code and should be called as part of system shutdown.

- **Generate preprocessor conditionals**

For a model containing Model blocks, specifies whether to generate preprocessor conditional directives globally for a model, locally for each variant Model block, or conditionally based on the **Generate preprocessor conditionals** setting in the Model Reference Parameter dialog for each variant Model block (`Use local settings` by default).

- **Suppress error status in real-time model data structure**

Specifies whether to omit the error status field from the generated real-time model data structure `rtModel` (off by default). Selecting this option reduces memory usage.

Be aware that selecting this option can cause the code generator to omit the `rtModel` data structure from generated code.

- **Combine signal/state structures**

Specifies whether to combine global block signals and global state data into one data structure in the generated code (off by default). Selecting this option reduces RAM and improves readability of the generated code.

- **Block parameter visibility**

Specifies whether to generate the block parameter structure as a `public`, `private`, or `protected` data member of the C++ model class (`private` by default).

- **Internal data visibility**

Specifies whether to generate internal data structures, such as Block I/O, DWork vectors, Runtime model, Zero-crossings, and continuous states, as `public`, `private`, or `protected` data members of the C++ model class (`private` by default).

- **Block parameter access**

Specifies whether to generate access methods for block parameters for the C++ model class (`None` by default). You can select noninlined access methods (`Method`) or inlined access methods (`Inlined method`).

- **Internal data access**

Specifies whether to generate access methods for internal data structures, such as Block I/O, DWork vectors, Runtime model, Zero-crossings, and continuous states, for the C++ model class (`None` by default). You can select noninlined access methods (`Method`) or inlined access methods (`Inlined method`).

- **External I/O access**

Specifies whether to generate access methods for root-level I/O signals for the C++ model class (`None` by default). If you want to generate access methods, you have the following options:

- Generate either noninlined or inlined access methods.

- Generate either *per-signal* or *structure-based* access methods. That is, you can generate a series of set and get methods on a per-signal basis, or generate just one set method that takes the address of an external input structure as an argument and, for external outputs (if applicable), just one get method that returns a reference to an external output structure. The generated code for structure-based access methods has the following general form:

```
class ModelClass {  
    ...  
    // Root inports set method  
    void setExternalInputs(const ExternalInputs* pExternalInputs);  
  
    // Root outputs get method  
    const ExternalOutputs & getExternalOutputs() const;  
}
```

Note: This parameter affects generated code only if you are using the default (void-void style) step method for your model class; *not* if you are explicitly passing arguments for root-level I/O signals using an I/O arguments style step method. For more information, see “Passing No Arguments (void-void)” on page 11-35 and “Passing I/O Arguments” on page 11-37.

- **Generate destructor**

Specifies whether to generate a destructor for the C++ model class (on by default).

- **Use dynamic memory allocation for model block instantiation**

For a model containing Model blocks, specifies whether generated code should use dynamic memory allocation, during model object registration, to instantiate objects for referenced models configured with a C++ class interface (off by default). If you select this option, during instantiation of an object for the top model in a model reference hierarchy, the generated code uses the operator `new` to instantiate objects for referenced models.

Selecting this option frees a parent model from having to maintain information about referenced models beyond its direct children. Clearing this option means that a parent model maintains information about its referenced models, including its direct and indirect children.

Note:

- If you select this option, be aware that a `bad_alloc` exception might be thrown, per the C++ standard, if an out-of-memory error occurs during the use of `new`. You must provide code to catch and process the `bad_alloc` exception in case an out-of-memory error occurs for a `new` call during construction of a top model object.
 - If **Use dynamic memory allocation for model block instantiation** is selected and the base model contains a Model block, the build process might generate copy constructor and assignment operator functions in the private section of the model class. The purpose of the functions is to prevent pointer members within the model class from being copied by other code. For more information, see “Model Class Copy Constructor and Assignment Operator” on page 11-47.
-

- **Configure C++ Class Interface**

Opens the Configure C++ class interface dialog box, which allows you to configure the step method for your model class. For more information, see “Configure Step Method for Your Model Class” on page 11-35.

Configure Step Method for Your Model Class

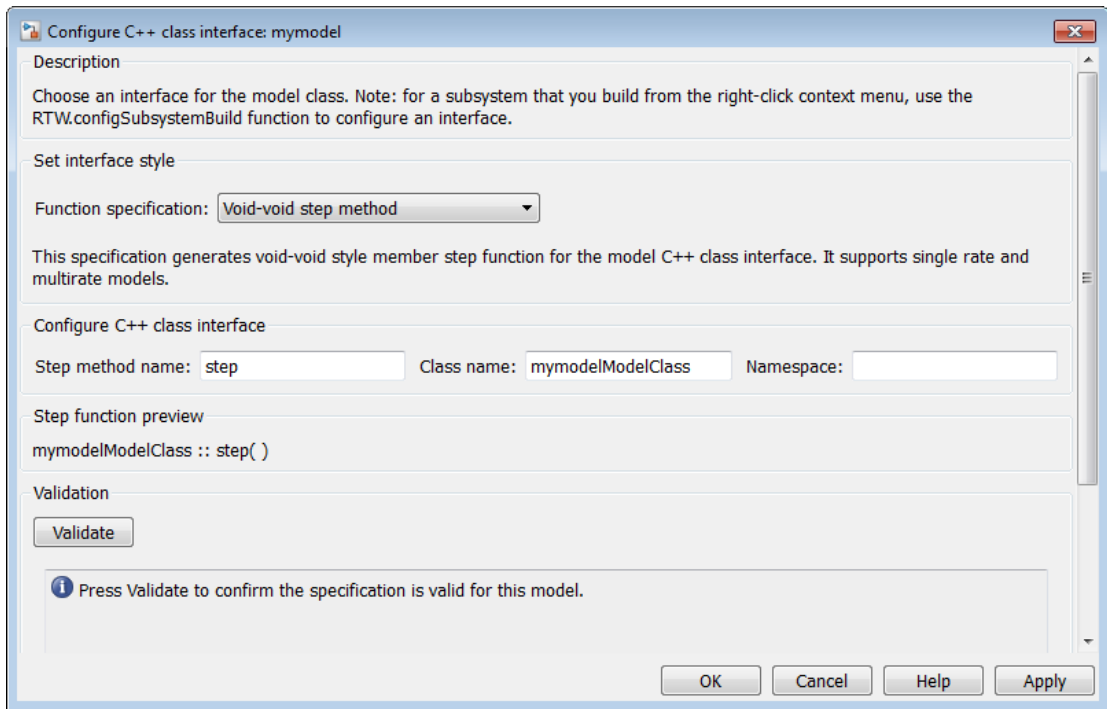
To configure the step method for your model class, on the **Code Generation > Interface** pane, click the **Configure C++ Class Interface** button, which is available when you select C++ `class` code interface packaging for your model. This action opens the Configure C++ class interface dialog box, where you can configure the step method for your model class in either of two styles:

- “Passing No Arguments (void-void)” on page 11-35
- “Passing I/O Arguments” on page 11-37

Note: The `void-void` style of step method specification supports single-rate models and multirate models, while the I/O arguments style supports single-rate models and multirate single-tasking models.

Passing No Arguments (void-void)

The Configure C++ class interface dialog box initially displays a view for configuring a `void-void` style step method for the model class.



- **Step method name**

Allows you to specify a step method name other than the default, `step`.

- **Class name**

Allows you to specify a model class name other than the default, `modelModelClass`.

- **Namespace**

Allows you to specify a namespace for the model class. If specified, the namespace is emitted in the generated code for the model class. The **Namespace** parameter provides a means of scoping C++ model classes. In a model reference hierarchy, you can specify a different namespace for each referenced model.

- **Step function preview**

Displays a preview of the model step function prototype as currently configured. The preview display is dynamically updated as you make configuration changes.

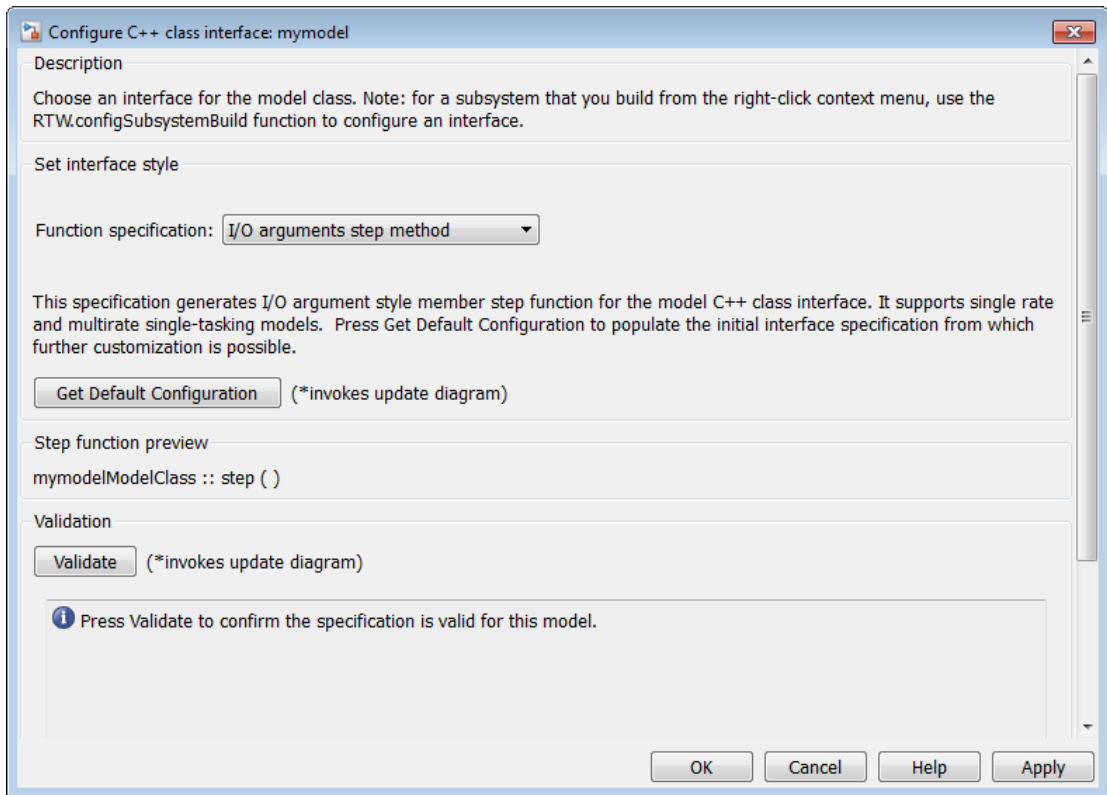
- **Validate**

Validates your current model step function configuration. The **Validation** pane displays the status and an explanation of any failure.

Passing I/O Arguments

If you select **I/O arguments step method** from the **Function specification** menu, the dialog box displays a view for configuring an I/O arguments style step method for the model class.

Note: To use the I/O arguments style step method, you must select the option **Remove root level I/O zero initialization** on the **Optimization** pane of the Configuration Parameters dialog box.



- **Get Default Configuration**

Click this button to get the initial interface configuration that provides a starting point for further customization.

- **Step function preview**

Displays a preview of the model step function prototype as currently configured. The preview dynamically updates as you make configuration changes.

- **Validate**

Validates your current model step function configuration. The **Validation** pane displays the status and an explanation of any failure.

When you click **Get Default Configuration**, the **Configure C++ class interface** subpane appears in the dialog box, displaying the initial interface configuration. For example:

Configure C++ class interface

Step method name: Class name: Namespace:

Order	Port Name	Port Type	Category	Argument Name	Qualifier
1	In1	Inport	Value	arg_In1	none
2	In2	Inport	Value	arg_In2	none
3	In3	Inport	Value	arg_In3	none
4	Out1	Outport	Pointer	arg_Out1	none
5	Out2	Outport	Pointer	arg_Out2	none

Up
Down

- **Step method name**

Allows you to specify a step method name other than the default, `step`.

- **Class name**

Allows you to specify a model class name other than the default, `modelModelClass`.

- **Namespace**

Allows you to specify a namespace for the model class. If specified, the namespace is emitted in the generated code for the model class. The **Namespace** parameter provides a means of scoping C++ model classes. In a model reference hierarchy, you can specify a different namespace for each referenced model.

- **Order**

Displays the numerical position of each argument. Use the **Up** and **Down** buttons to change argument order.

- **Port Name**

Displays the port name of each argument (not configurable using this dialog box).

- **Port Type**

Displays the port type, `Inport` or `Outport`, of each argument (not configurable using this dialog box).

- **Category**

Displays the passing mechanism for each argument. To change the passing mechanism for an argument, select **Value**, **Pointer**, or **Reference** from the argument's **Category** menu.

- **Argument Name**

Displays the name of each argument. To change an argument name, click in the argument's **Argument name** field, position the cursor for text entry, and enter the new name.

- **Qualifier**

Displays the **const** type qualifier for each argument. To change the qualifier for an argument, select an available value from the argument's **Qualifier** menu. The possible values are:

- none
- **const** (value)
- **const*** (value referenced by the pointer)
- **const*const** (value referenced by the pointer and the pointer itself)
- **const &** (value referenced by the reference)

Tip When a model includes a referenced model, the **const** type qualifier for the root input argument of the referenced model's specified step function interface is set to **none** and the qualifier for the source signal in the referenced model's parent is set to a value other than **none**, code generation honors the referenced model's interface specification by generating a type cast that discards the **const** type qualifier from the source signal. To override this behavior, add a **const** type qualifier to the referenced model.

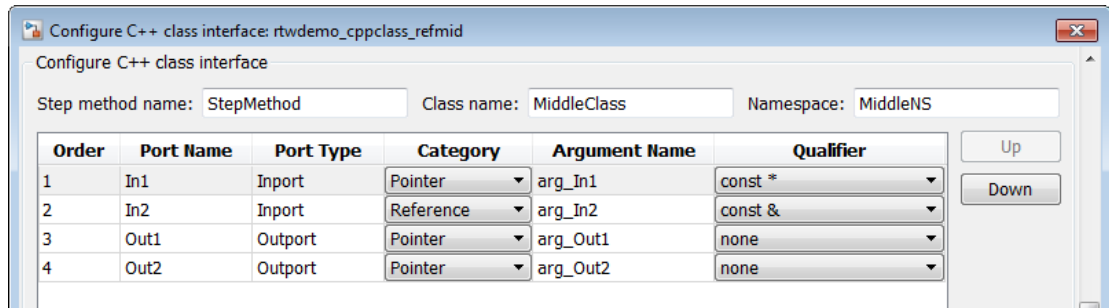
Use Namespaces to Scope C++ Model Classes

Embedded Coder provides namespace control for scoping model classes generated using C++ class code interface packaging. In the Configure C++ class interface dialog box, use the **Namespace** parameter to specify a namespace for a model class. If specified, the namespace is emitted in the generated code for the model class. To scope the C++ model classes in a model reference hierarchy, you can specify a different namespace for each referenced model.

For an example of namespace control, see the example model `rtwdemo_cppclass`. This model assigns namespaces as follows:

- `TopNS` for top-level model `rtwdemo_cppclass`
- `MiddleNS` for referenced model `rtwdemo_cppclass_refmid`
- `BottomNS` for referenced model `rtwdemo_cppclass_refbot`

If you build the model with its default settings, you can examine the generated header and source files for each model to see where the namespace is emitted. For example, the **Namespace** setting for the model `rtwdemo_cppclass_refmid` is shown below, followed by excerpts of the emitted namespace code in the model header and source files.



```

42 // Class declaration for model rtwdemo_cppclass_refmid
43 namespace MiddleNS {
44     class MiddleClass {
45         // public data and function members
46     public:
47         // Model entry point functions
48     ...
49         // model step function
50         void StepMethod(const real_T *arg_In1, const real_T &arg_In2, real_T
51             *arg_Out1, real_T *arg_Out2);
52     ...
53     };
54 }

15 #include "rtwdemo_cppclass_refmid.h"
16 #include "rtwdemo_cppclass_refmid_private.h"
17
18 namespace MiddleNS
19 {
20     // Model step function
21     void MiddleClass::StepMethod(const real_T *arg_In1, const real_T &arg_In2,
22         real_T *arg_Out1, real_T *arg_Out2)
23     {
24     ...

```

```
43     }  
...  
83 }
```

Configure C++ Class Interfaces for Nonvirtual Subsystems

You can configure C++ class interfaces for right-click builds of nonvirtual subsystems in Simulink models, if the following requirements are met:

- The model is configured for the C++ language and C++ class code interface packaging.
- 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`.

To configure C++ class interfaces for a subsystem that meets the requirements:

- 1 Open the containing model and select the subsystem block.
- 2 Enter the following MATLAB command:

```
RTW.configSubsystemBuild(gcb)
```

where `gcb` is the Simulink function `gcb`, returning the full block path name of the current block.

This command opens a subsystem equivalent of the Configure C++ class interface dialog sequence that is described in detail in the preceding section, “Configure Step Method for Your Model Class” on page 11-35. (For more information about using the MATLAB command, see `RTW.configSubsystemBuild`.)

- 3 Use the Configure C++ class interface dialog boxes to configure C++ class settings for the subsystem.
- 4 Right-click the subsystem and select **C/C++ Code > Build This Subsystem**.
- 5 When the subsystem build completes, you can examine the C++ class interfaces in the generated files and the HTML code generation report.

Customize C++ Class Interfaces Programmatically

If you select the **Code interface packaging** option C++ class for your model, you can use the C++ class interface control functions (listed in C++ Class Interface Control Functions) to programmatically configure the step method for your model class.

Typical uses of these functions include:

- **Create and validate a new step method interface, starting with default configuration information from your Simulink model**
 - 1 Create a model-specific C++ class interface with `obj = "RTW.ModelCPPVoidClass"` or `obj = "RTW.ModelCPPArgsClass"`, where `obj` returns a handle to a newly created, empty C++ class interface.
 - 2 Attach the C++ class interface to your loaded ERT-based Simulink model using `"attachToModel"`.
 - 3 Get default C++ class interface configuration information from your model using `"getDefaultConf"`.
 - 4 Use the `Get` and `Set` functions listed in C++ Class Interface Control Functions to test or reset the model class name and model step method name. Additionally, if you are using the I/O arguments style step method, you can test and reset argument names, argument positions, argument categories, and argument type qualifiers.
 - 5 Validate the C++ class interface using `"runValidation"`. (If validation fails, use the error message information that `runValidation` returns to address the issues.)
 - 6 Save your model and then generate code using the `rtwbuild` function.
- **Modify and validate an existing step method interface for a Simulink model**
 - 1 Get the handle to an existing model-specific C++ class interface that is attached to your loaded ERT-based Simulink model using `obj = RTW.getClassInterfaceSpecification(modelName)`, where `modelName` is a string specifying the name of a loaded ERT-based Simulink model, and `obj` returns a handle to a C++ class interface attached to the specified model. If the model does not have an attached C++ class interface configuration, the function returns `[]`.
 - 2 Use the `Get` and `Set` functions listed in C++ Class Interface Control Functions to test or reset the model class name and model step method name. Additionally, if the returned interface uses the I/O arguments style step method, you can test and reset argument names, argument positions, argument categories, and argument type qualifiers.
 - 3 Validate the C++ class interface using `"runValidation"`. (If validation fails, use the error message information that `runValidation` returns to address the issues.)

- 4 Save your model and then generate code using the `rtwbuild` function.

Note: You should not use the same model-specific C++ class interface control object across multiple models. If you do, changes that you make to the step method configuration in one model propagate to other models, which is usually not desirable.

C++ Class Interface Control Functions

Function	Description
“attachToModel”	Attach model-specific C++ class interface to loaded ERT-based Simulink model
“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
“getClassName”	Get class name from model-specific C++ class interface
“getDefaultConf”	Get default configuration information for model-specific C++ class interface from Simulink model to which it is attached
“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
<code>RTW.configSubsystemBuild</code>	Open GUI to configure C function prototype or C++ class interface for right-click build of specified subsystem
<code>RTW.getClassInterfaceSpecification</code>	Get handle to model-specific C++ class interface control object
“runValidation”	Validate model-specific C++ class interface against Simulink model to which it is attached

Function	Description
“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
“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

Configure Step Method for Model Class

The following sample MATLAB script configures the step method for the `rtwdemo_counter` model class, using the C++ Class Interface Control Functions.

```

%% Open the rtwdemo_counter model
rtwdemo_counter

%% Select ert.tlc as the System Target File for the model
set_param(gcs,'SystemTargetFile','ert.tlc')

%% Select C++ as the target language for the model
set_param(gcs,'TargetLang','C++')

%% Select C++ class as the code interface packaging for the model
set_param(gcs,'CodeInterfacePackaging','C++ class')

%% Set required option for I/O arguments style step method (cmd off = GUI on)
set_param(gcs,'ZeroExternalMemoryAtStartup','off')

%% Create a C++ class interface using an I/O arguments style step method
a=RTW.ModelCPPArgsClass

%% Attach the C++ class interface to the model
attachToModel(a,gcs)

%% Get the default C++ class interface configuration from the model
getDefaultConf(a)

%% Move the Output port argument from position 2 to position 1
setArgPosition(a,'Output',1)

```

```

%% Reset the model step method name from step to StepMethod
setStepMethodName(a,'StepMethod')

%% Change the Input port argument name, category, and qualifier
setArgName(a,'Input','inputArg')
setArgCategory(a,'Input','Pointer')
setArgQualifier(a,'Input','const *')

%% Validate the function prototype against the model
[status,message]=runValidation(a)

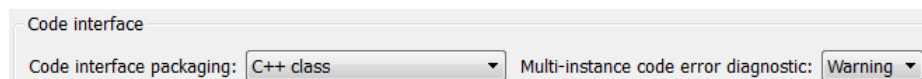
%% if validation succeeded, generate code and build
if status
    rtwbuild(gcs)
end

```

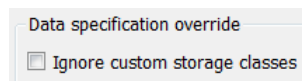
Specify Custom Storage Class for C++ Class Code Generation

To configure a Simulink parameter, signal, or state to use a custom storage class (CSC) with C++ class code generation:

- 1 Open an ERT-based model for which **Language** is set to C++ and **Code interface packaging** is set to C++ class.
- 2 Open the Configuration Parameters dialog box.
- 3 On the **Code Generation > Interface** pane, set the **Multi-instance code error diagnostic** parameter to a value other than Error.



- 4 On the **Code Generation** pane, if the option **Ignore custom storage classes** is selected, clear the option.



Apply the changes.

- 5 In the model, select a custom storage class for a parameter, signal, or state. For example, select a signal, open its Properties dialog box, and view its code generation options. In the **Storage class** drop-down list, select a custom storage class, and then configure its attributes. Apply the changes.

Note: C++ class code generation does not support the following CSCs:

- CSCs with `Volatile` specifications.
 - CSCs of type `Other`, except `GetSet`.
-

- 6 Build the model.
- 7 In the code generation report, examine the files `model.h` and `model.cpp` to observe the use of CSCs in the generated C++ code.

Model Class Copy Constructor and Assignment Operator

Code generation automatically adds a copy constructor and an assignment operator to C++ class declarations when required to securely handle pointer members. The constructor and operator are added as private member functions when both of the following conditions exist:

- The model option “**Use dynamic memory allocation for model block instantiation**” is set to on.
- The base model contains a Model block. The Model block is not directly or indirectly within a subsystem for which **Function packaging** is set to `Reusable function`.

Under these conditions, the software generates a private copy constructor and assignment operator to prevent pointer members within the model class from being copied by other code.

Note: To prevent generation of these functions, consider clearing the option **Use dynamic memory allocation for model block instantiation**.

The code excerpt below shows generated `model.h` code for a model class that has a pointer member. (Look for instances of `MiddleClass_ptr`). The copy constructor and assignment operator declarations are shown in **bold**.

```
class MiddleClass;    // class forward declaration for <S1>/Bottom model instance
typedef MiddleClass* MiddleClass_ptr;
...

// Class declaration for model cppclass_top
```

```
class Top {
...
// private data and function members
private:
// Block signals
BlockIO_cppclass_top cppclass_top_B;

// Block states
D_Work_cppclass_top cppclass_top_DWork;

// Real-Time Model
RT_MODEL_cppclass_top cppclass_top_M;

// private member function(s) for subsystem '<Root>/Subsystem'
void cppclass_top_Subsystem_Init();
void cppclass_top_Subsystem_Start();
void cppclass_top_Subsystem();

//Copy Constructor
Top(const Top &rhs);

//Assignment Operator
Top& operator= (const Top &rhs);

// model instance variable for '<S1>/Bottom model instance'
MiddleClass_ptr Bottom_model_instanceMDLOBJ1;
};
```

C++ Class Interface Control Limitations

- The C++ `class` code interface packaging option does not support some Simulink model configuration options. Selecting C++ `class` disables the following items in the Configuration Parameters dialog box:
 - **Identifier format control** subpane on the **Symbols** pane
 - **File customization template** parameter on the **Templates** pane

Note: The code and data templates on the **Templates** pane are supported for C++ `class` code generation. However, the following template file features that are supported for other language selections are not supported for C++ class generated code:

- Free-form text outside template sections
 - Custom tokens
 - TLC commands (<! > tokens)
-

- **Global data placement (custom storage classes only)** subpane on the **Code Placement** pane
- **Memory Sections** pane
- Among the data exchange interfaces available on the **Interface** pane of the Configuration Parameters dialog box, only the **C API** interface is supported for **C++ class** code generation. If you select **External mode** or **ASAP2**, code generation fails with a validation error.
- The **I/O arguments** style of step method specification supports single-rate models and multirate single-tasking models, but not multirate multitasking models.
- The **Code Generation > Export Functions** capability does not support **C++ class** code interface packaging.
- If you have a Stateflow license, for a Stateflow chart that resides in a root model configured to use the **I/O arguments step method** function specification, and that uses a model root inport value or calls a subsystem that uses a model root inport value, you must do one of the following to generate code:
 - Clear the **Execute (enter) Chart At Initialization** check box in the Stateflow chart.
 - Insert a Simulink Signal Conversion block immediately after the root inport. In the Signal Conversion block parameters dialog box, select **Exclude this block from 'Block reduction' optimization**.
- If a model root inport value connects to a Simscape conversion block, you must insert a Simulink Signal Conversion block between the root inport and the Simscape conversion block. In the Signal Conversion block parameters dialog box, select **Exclude this block from 'Block reduction' optimization**.
- When building a referenced model that is configured to generate a C++ class interface:
 - You must use the **I/O arguments step method** style of the C++ class interface. The **void-void step method** style is not supported for referenced models.
 - You cannot use a C++ class interface in cases when a referenced model cannot have a combined output/update function. Cases include a model that
 - Has a continuous sample time
 - Saves states

Atomic Subsystem Code

In this section...

“About Nonvirtual Subsystem Code Generation” on page 11-50

“Configure Subsystem for Generating Modular Function Code” on page 11-51

“Modular Function Code for Nonvirtual Subsystems” on page 11-56

“Nonvirtual Subsystem Modular Function Code Limitations” on page 11-61

About Nonvirtual Subsystem Code Generation

The Embedded Coder software provides a Subsystem Parameters dialog box option, **Function with separate data**, that allows you to generate modular function code for nonvirtual subsystems, including atomic subsystems and conditionally executed subsystems.

By default, the generated code for a nonvirtual subsystem does not separate a subsystem's internal data from the data of its parent Simulink model. This can make it difficult to trace and test the code, particularly for nonreusable subsystems. Also, in large models containing nonvirtual subsystems, data structures can become large and potentially difficult to compile.

Function with separate data allows you to generate subsystem function code in which the internal data for a nonvirtual subsystem is separated from its parent model and is owned by the subsystem. The subsystem data structure is declared independently from the parent model data structures. A subsystem with separate data has its own block I/O and DWork data structure. As a result, the generated code for the subsystem is easier to trace and test. The data separation also tends to reduce the maximum size of global data structures throughout the model, because they are split into multiple data structures.

To use the **Function with separate data** parameter,

- Your model must use an ERT-based system target file (requires a Embedded Coder license).
- Your subsystem must be configured to be atomic or conditionally executed. For more information, see “Systems and Subsystems”.
- Your subsystem must use the Nonreusable function setting for **Code Generation > Function packaging**.

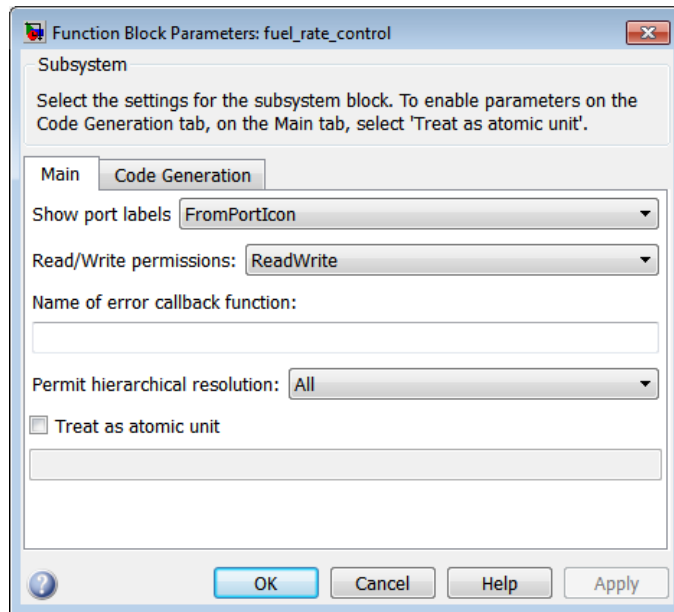
To configure your subsystem for generating modular function code, you invoke the Subsystem Parameters dialog box and make a series of selections to display and enable the **Function with separate data** option. See “Configure Subsystem for Generating Modular Function Code” on page 11-51 and “Modular Function Code for Nonvirtual Subsystems” on page 11-56 for details. For limitations that apply, see “Nonvirtual Subsystem Modular Function Code Limitations” on page 11-61.

For more information about generating code for atomic subsystems, see the sections “Code Generation of Subsystems” and “Generate Code and Executables for Individual Subsystem” in the Simulink Coder documentation.

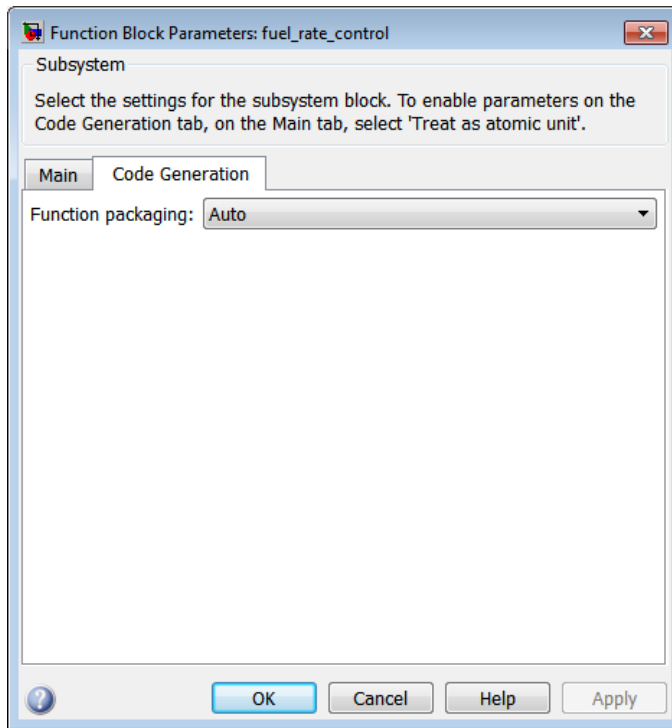
Configure Subsystem for Generating Modular Function Code

This section summarizes the steps to configure a nonvirtual subsystem in a Simulink model for modular function code generation.

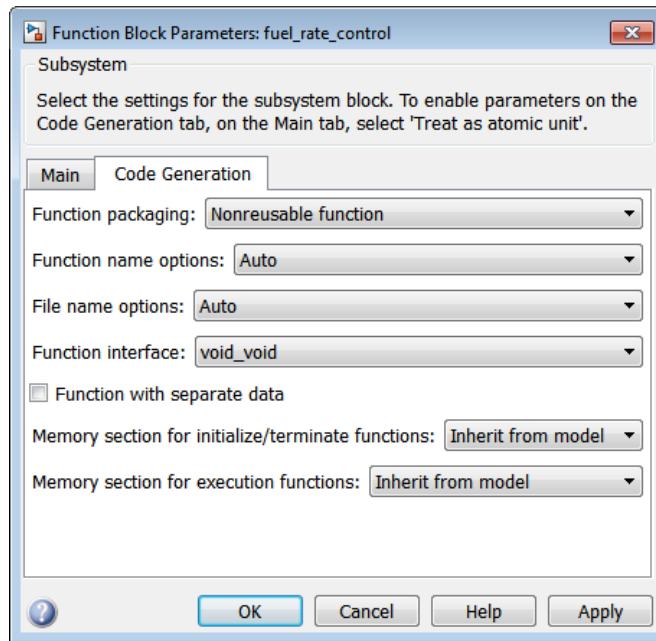
- 1 Verify that the Simulink model containing the subsystem uses an ERT-based system target file (see the **System target file** parameter on the **Code Generation** pane of the Configuration Parameters dialog box).
- 2 In your Simulink model, select the subsystem for which you want to generate modular function code and launch the Subsystem Parameters dialog box (for example, right-click the subsystem and select **Block Parameters (Subsystem)**). The dialog box for an atomic subsystem is shown below. (In the dialog box for a conditionally executed subsystem, the dialog box option **Treat as atomic unit** is greyed out, and you can skip Step 3.)



- 3 If the Subsystem Parameters dialog box option **Treat as atomic unit** is available for selection but not selected, the subsystem is neither atomic nor conditionally executed. Select the option **Treat as atomic unit**, which enables **Function packaging** on the **Code Generation** tab. Select the **Code Generation** tab.

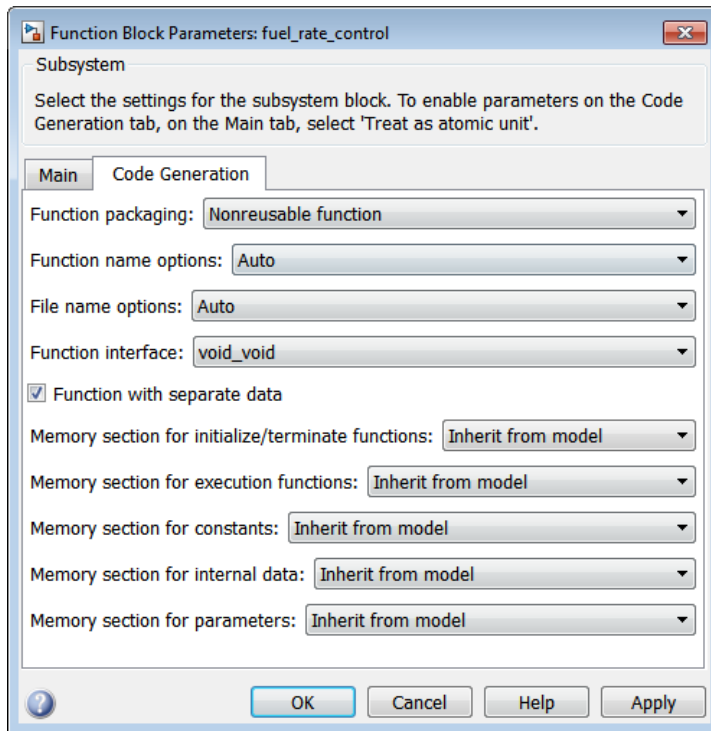


- 4 For the **Function packaging** parameter, select the value **Nonreusable function**. After you make this selection, the **Function with separate data** option is displayed.



Note: Before you generate nonvirtual subsystem function code with the **Function with separate data** option selected, you might want to generate function code with the option *deselected* and save the generated function `.c` and `.h` files in a separate directory for later comparison.

- 5 Select the **Function with separate data** option. After you make this selection, additional configuration parameters are displayed.



Note: To control the naming of the subsystem function and the subsystem files in the generated code, you can modify the subsystem parameters **Function name options** and **File name options**.

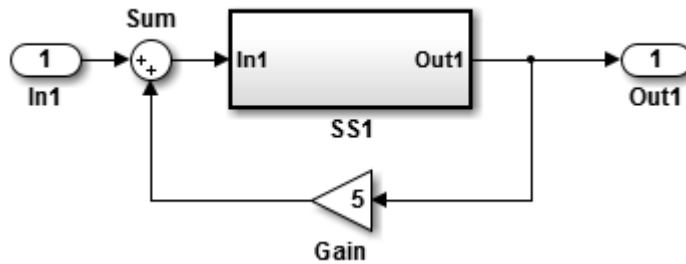
- 6 To save your subsystem parameter settings and exit the dialog box, click **OK**.

This completes the subsystem configuration for generating modular function code. You can now generate the code for the subsystem and examine the generated files, including the function `.c` and `.h` files named according to your subsystem parameter specifications. For more information on generating code for nonvirtual subsystems, see “Code Generation of Subsystems”. For examples of generated subsystem function code, see “Modular Function Code for Nonvirtual Subsystems” on page 11-56.

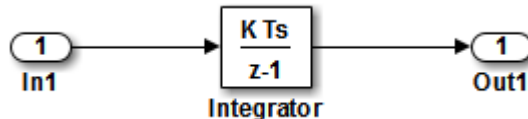
Modular Function Code for Nonvirtual Subsystems

To illustrate the selection of the **Function with separate data** option for a nonvirtual subsystem, the following procedure generates atomic subsystem function code with and without the option selected and compares the results.

- 1 Open MATLAB and launch the model `rtwdemo_atomic` using the MATLAB command `rtwdemo_atomic`. Examine the Simulink model.



- 2 Double-click the SS1 subsystem and examine the contents. (You can close the subsystem window when you are finished.)



- 3 Use the Configuration Parameters dialog box to change the model's **System target file** from GRT to ERT. For example, from the Simulink window, select **Simulation > Model Configuration Parameters**. On the Configuration Parameters dialog box, select the **Code Generation** pane and specify `ert.tlc` for the **System target file** parameter. Click **OK** twice to confirm the change.
- 4 Create a variant of `rtwdemo_atomic` that illustrates function code *without* data separation.
 - a In the Simulink view of `rtwdemo_atomic`, right-click the SS1 subsystem and select **Block Parameters (Subsystem)**. In the Subsystem Parameters dialog box, verify that
 - On the **Main** tab, **Treat as atomic unit** is selected

- On the **Code Generation** tab, `User specified` is selected for **Function name options**
 - On the **Code Generation** tab, `myfun` is specified for **Function name**
- b** In the Subsystem Parameters dialog box, on the **Code Generation** tab
- i** Select the value `Nonreusable function` for the **Function packaging** parameter. After this selection, additional parameters and options will appear.
 - ii** Select the value `Use function name` for the **File name options** parameter. This selection is optional but simplifies the later task of code comparison by causing the atomic subsystem function code to be generated into the files `myfun.c` and `myfun.h`.
- Do *not* select the option **Function with separate data**. Click **Apply** to apply the changes and click **OK** to exit the dialog box.
- c** Save this model variant to a personal work directory, for example, `rtwdemo_atomic1` in `d:/atomic`.
- 5** Create a variant of `rtwdemo_atomic` that illustrates function code *with* data separation.
- a** In the Simulink view of `rtwdemo_atomic1` (or `rtwdemo_atomic` with step 3 reapplied), right-click the SS1 subsystem and select **Block Parameters (Subsystem)**. In the Subsystem Parameters dialog box, verify that
- On the **Main** tab, **Treat as atomic unit** is selected
 - On the **Code Generation** tab, `Function` is selected for **Function packaging**
 - On the **Code Generation** tab, `User specified` is selected for **Function name options**
 - On the **Code Generation** tab, `myfun` is specified for **Function name**
 - On the **Code Generation** tab, `Use function name` is specified for **File name options**
- b** In the Subsystem Parameters dialog box, on the **Code Generation** tab, select the option **Function with separate data**. Click **Apply** to apply the change and click **OK** to exit the dialog box.
- c** Save this model variant, using a different name than the first variant, to a personal work directory, for example, `rtwdemo_atomic2` in `d:/atomic`.

- 6 Generate code for each model, `rtwdemo_atomic1` and `rtwdemo_atomic2`.
- 7 In the generated code directories, compare the `model.c/.h` and `myfun.c/.h` files generated for the two models. (In this example, there are not significant differences in the generated variants of `ert_main.c`, `model_private.h`, `model_types.h`, or `rtwtypes.h`.)

H File Differences for Nonvirtual Subsystem Function Data Separation

The differences between the H files generated for `rtwdemo_atomic1` and `rtwdemo_atomic2` help illustrate the selection of the **Function with separate data** option for nonvirtual subsystems.

- 1 Selecting **Function with separate data** causes typedefs for subsystem data to be generated in the `myfun.h` file for `rtwdemo_atomic2`:

```
/* Block signals for system '<Root>/SS1' */
typedef struct {
    real_T Integrator;          /* '<S1>/Integrator' */
} rtB_myfun;

/* Block states (auto storage) for system '<Root>/SS1' */
typedef struct {
    real_T Integrator_DSTATE;   /* '<S1>/Integrator' */
} rtDW_myfun;
```

By contrast, for `rtwdemo_atomic1`, typedefs for subsystem data belong to the model and appear in `rtwdemo_atomic1.h`:

```
/* Block signals (auto storage) */
typedef struct {
    ...
    real_T Integrator;          /* '<S1>/Integrator' */
} BlockIO_rtwdemo_atomic1;

/* Block states (auto storage) for system '<Root>' */
typedef struct {
    real_T Integrator_DSTATE;   /* '<S1>/Integrator' */
} D_Work_rtwdemo_atomic1;
```

- 2 Selecting **Function with separate data** generates the following external declarations in the `myfun.h` file for `rtwdemo_atomic2`:

```
/* Extern declarations of internal data for 'system '<Root>/SS1'' */
extern rtB_myfun rtwdemo_atomic2_myfunB;
```



```
extern rtDW_myfun rtwdemo_atomic2_myfunDW;
extern void myfun_initialize(void);
```

By contrast, the generated code for `rtwdemo_atomic1` contains model-level external declarations for the subsystem's `BlockIO` and `D_Work` data, in `rtwdemo_atomic1.h`:

```
/* Block signals (auto storage) */
extern BlockIO_rtwdemo_atomic1 rtwdemo_atomic1_B;

/* Block states (auto storage) */
extern D_Work_rtwdemo_atomic1 rtwdemo_atomic1_DWork;
```

C File Differences for Nonvirtual Subsystem Function Data Separation

The differences between the C files generated for `rtwdemo_atomic1` and `rtwdemo_atomic2` illustrate the selection of the **Function with separate data** option for nonvirtual subsystems.

- 1 Selecting **Function with separate data** causes a separate subsystem initialize function, `myfun_initialize`, to be generated in the `myfun.c` file for `rtwdemo_atomic2`:

```
void myfun_initialize(void) {
    {
        ((real_T*)&rtwdemo_atomic2_myfunB.Integrator)[0] = 0.0;
    }
    rtwdemo_atomic2_myfunDW.Integrator_DSTATE = 0.0;
}
```

The subsystem initialize function in `myfun.c` is invoked by the model initialize function in `rtwdemo_atomic2.c`:

```
/* Model initialize function */

void rtwdemo_atomic2_initialize(void)
{
    ...

    /* Initialize subsystem data */
    myfun_initialize();
}
```

By contrast, for `rtwdemo_atomic1`, subsystem data is initialized by the model initialize function in `rtwdemo_atomic1.c`:

```
/* Model initialize function */

void rtwdemo_atomic1_initialize(void)
{
    ...
    /* block I/O */
    {
        ...
        ((real_T*)&rtwdemo_atomic1_B.Integrator)[0] = 0.0;
    }

    /* states (dwork) */

    rtwdemo_atomic1_DWork.Integrator_DSTATE = 0.0;
    ...
}
```

- 2** Selecting **Function with separate data** generates the following declarations in the `myfun.c` file for `rtwdemo_atomic2`:

```
/* Declare variables for internal data of system '<Root>/SS1' */
rtB_myfun rtwdemo_atomic2_myfunB;

rtDW_myfun rtwdemo_atomic2_myfunDW;
```

By contrast, the generated code for `rtwdemo_atomic1` contains model-level declarations for the subsystem's `BlockIO` and `D_Work` data, in `rtwdemo_atomic1.c`:

```
/* Block signals (auto storage) */
BlockIO_rtwdemo_atomic1 rtwdemo_atomic1_B;

/* Block states (auto storage) */
D_Work_rtwdemo_atomic1 rtwdemo_atomic1_DWork;
```

- 3** Selecting **Function with separate data** generates identifier naming that reflects the subsystem orientation of data items. Notice the references to subsystem data in subsystem functions such as `myfun` and `myfun_update` or in the model's `model_step` function. For example, compare this code from `myfun` for `rtwdemo_atomic2`

```
/* DiscreteIntegrator: '<S1>/Integrator' */  
rtwdemo_atomic2_myfunB.Integrator = rtwdemo_atomic2_myfunDW.Integrator_DSTATE;
```

to the corresponding code from myfun for rtwdemo_atomic1.

```
/* DiscreteIntegrator: '<S1>/Integrator' */  
rtwdemo_atomic1_B.Integrator = rtwdemo_atomic1_DWork.Integrator_DSTATE;
```

Nonvirtual Subsystem Modular Function Code Limitations

The nonvirtual subsystem option **Function with separate data** has the following limitations:

- The **Function with separate data** option is available only in ERT-based Simulink models (requires a Embedded Coder license).
- The nonvirtual subsystem to which the option is applied cannot have multiple sample times or continuous sample times; that is, the subsystem must be single-rate with a discrete sample time.
- The nonvirtual subsystem cannot contain continuous states.
- The nonvirtual subsystem cannot output function call signals.
- The nonvirtual subsystem cannot contain noninlined S-functions.
- The generated files for the nonvirtual subsystem will reference model-wide header files, such as *model.h* and *model_private.h*.
- The **Function with separate data** option is incompatible with the **Classic call interface** option, located on the **Code Generation > Interface** pane of the Configuration Parameters dialog box. Selecting both generates an error.
- The **Function with separate data** option is incompatible with setting **Code interface packaging** to Reusable function (**Code Generation > Interface** pane). Selecting both generates an error.

Memory Sections

- “About Memory Sections” on page 12-2
- “Requirements for Defining Memory Sections” on page 12-3
- “Define Memory Sections” on page 12-5
- “Configure Memory Sections” on page 12-9
- “Declare Constant Data as Volatile” on page 12-10
- “Apply Memory Sections” on page 12-13
- “Generated Code with Memory Sections” on page 12-21
- “Model-Level Data Structures” on page 12-23

About Memory Sections

What Are Memory Sections?

The Embedded Coder software provides a memory section capability that allows you to insert comments and pragmas and qualify constants as `volatile` in generated code for

- Data in custom storage classes
- Model-level functions
- Model-level internal data
- Subsystem functions
- Subsystem internal data

Pragmas inserted into generated code can surround

- A contiguous block of function or data definitions
- Each function or data definition separately

When pragmas surround each function or data definition separately, the text of each pragma can contain the name of the definition to which it applies.

To see an example of memory sections, type `rtwdemo_memsec` at the MATLAB command line.

Requirements for Defining Memory Sections

Before you can define memory sections, you must do the following:

- 1 Set the Simulink model's code generation target to an embedded target such as `ert.tlc`.
- 2 To create packages, specify package properties, or create classes, including custom storage classes, see “Define Data Classes” in the Simulink documentation.

See also the instructions that appear when you click the **Custom Storage Classes** tab.

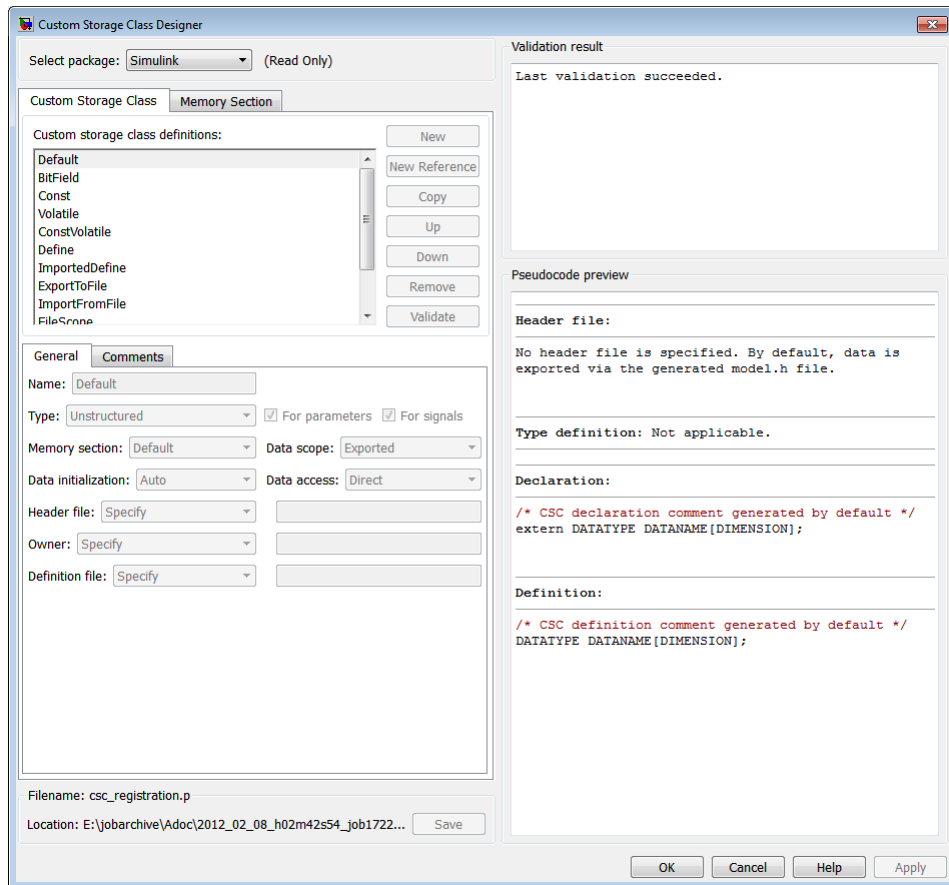
- 3 If you need to specify custom storage class properties,
 - a Choose **View > Model Explorer** in the model window.

The Model Explorer appears.

- b Choose **Tools > Custom Storage Class Designer** in the Model Explorer window.

A notification box appears that states **Please Wait ... Finding Packages**. After a brief pause, the notification box closes and the Custom Storage Class Designer appears.

- c Select the **Custom Storage Class** tab. The **Custom Storage Class** pane initially looks like this:



- d Use the **Custom Storage Class** pane to select a writable package and specify custom storage class properties. Instructions for using this pane appear in “Design Custom Storage Classes and Memory Sections”.

Define Memory Sections

In this section...

“Edit Memory Section Properties” on page 12-5

“Specify the Memory Section Name” on page 12-6

“Specify a Qualifier for Custom Storage Class Data Definitions” on page 12-7

“Specify Comment and Pragma Text” on page 12-7

“Surround Individual Definitions with Pragmas” on page 12-7

“Include Identifier Names in Pragmas” on page 12-8

Edit Memory Section Properties

After you have satisfied the requirements in “Requirements for Defining Memory Sections” on page 12-3, you can define memory sections and specify their properties. To create new memory sections or specify memory section properties,

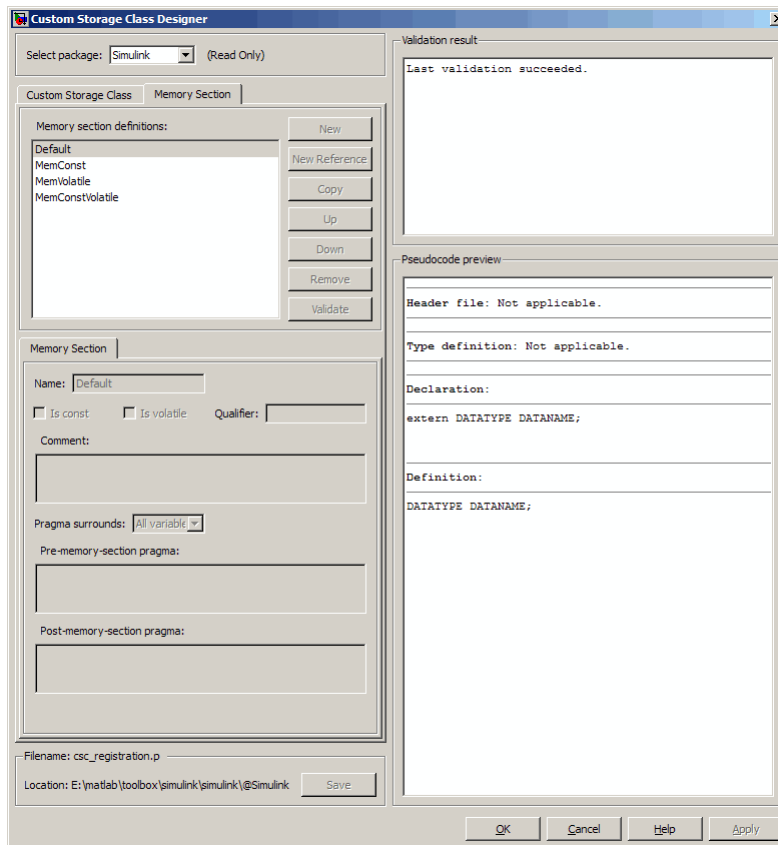
- 1 Choose **View > Model Explorer** in the model window.

The Model Explorer appears.

- 2 Choose **Tools > Custom Storage Class Designer** in the Model Explorer window.

A notification box appears that states **Please Wait ... Finding Packages**. After a brief pause, the notification box closes and the Custom Storage Class Designer appears.

- 3 Click the **Memory Section** tab of the Custom Storage Class Designer. The **Memory Section** pane initially looks like this:



- 4 If you intend to create or change memory section definitions, use the **Select package** field to select a writable package.

The rest of this section assumes that you have selected a writable package, and describes the use of the **Memory section** subpane on the lower left. For descriptions of the other subpanes, instructions for validating memory section definitions, and other information, see “Define Memory Sections” on page 12-5.

Specify the Memory Section Name

To specify the name of a memory section, use the **Name** field. A memory section name must be a legal MATLAB identifier.

Specify a Qualifier for Custom Storage Class Data Definitions

To specify a qualifier for custom storage class data definitions in a memory section, enter the components of the qualifier below the **Name** field.

- To specify **const**, check **Is const**.
- To specify **volatile**, check **Is volatile**.
- To specify anything else (e.g., **static**), enter the text in the **Qualifier** field.

The qualifier will appear in generated code with its components in the same left-to-right order in which their definitions appear in the dialog box. A preview appears in the **Pseudocode preview** subpane on the lower right.

Note: Specifying a qualifier affects only custom storage class data definitions. The code generator omits the qualifier from other definition categories.

Specify Comment and Pragma Text

To specify a comment, prepragma, or postpragma for a memory section, enter the comment in the text boxes on the left side of the Custom Storage Class Designer. In the text boxes, you can type multiple lines separated by ordinary Returns.

Surround Individual Definitions with Pragmas

If the **Pragma surrounds** field for a memory section specifies **Each variable**, the code generator will surround each definition in a contiguous block of definitions with the comment, prepragma, and postpragma defined for the section.

If the **Pragma surrounds** field for a memory section specifies **All variables**, the code generator will insert the comment and prepragma for the section before the first definition in a contiguous block of custom storage class data definitions, and the postpragma after the last definition in the block.

Note: Specifying **All variables** affects only custom storage class data definitions. For other definition categories, the code generator surrounds each definition separately regardless of the value of **Pragma surrounds**.

Include Identifier Names in Pragmas

When pragmas surround each separate definition in a contiguous block, you can include the string %<identifier> in a pragma. The string must appear without surrounding quotes.

- When %<identifier> appears in a prepragma, the code generator will substitute the identifier from the subsequent function or data definition.
- When %<identifier> appears in a postpragma, the code generator will substitute the identifier from the previous function or data definition.

You can use %<identifier> with pragmas *only* when pragmas to surround each variable. The **Validate** phase will report an error if you violate this rule.

Note: Although %<identifier> looks like a TLC variable, it is not: it is just a keyword that directs the code generator to substitute the applicable data definition identifier when it outputs a pragma. TLC variables cannot appear in pragma specifications in the **Memory Section** pane.

Configure Memory Sections

You configure memory sections by using the **Code Generation > Memory Sections** pane of the Configuration Parameters dialog box.

To...	Select...
Specify the package that contains memory sections that you want to apply	The name of a package for Package . Click Refresh package list to refresh the list of available packages in your configuration.
Apply memory sections to initialize/start and terminate functions	A value for Initialize/Terminate .
Apply memory sections to step, run-time initialization, derivative, enable, and disable functions	A value for Execution .
Apply memory sections to constant parameters, constant, block I/O, or zero representation	A value for Constants .
Apply memory sections to root inputs or outputs	A value for Inputs/Outputs .
Apply memory sections to block I/O, Dwork vectors, run-time models, zero-crossings	A value for Internal data .
Apply memory sections to parameters	A value for Parameters .

The interface checks whether the specified package is on the MATLAB path and that the selected memory sections are in the package. The results of this validation appear in the field **Validation results**.

Declare Constant Data as Volatile

In the C language, the value of data declared with the storage type qualifier, `volatile`, can be read from memory and written back to memory when changed without compiler control or detection. Examples of use include variables for initialization at system power-up or for system clock updates.

You can add the `volatile` qualifier to type definitions generated in code for model constant block I/O, constant parameters, and ground data (zero representation).

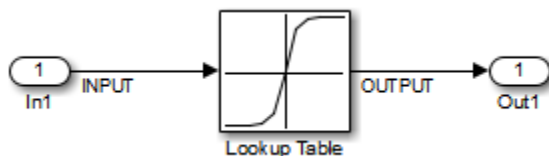
To add the `volatile` qualifier to type definitions, you must configure your model as follows:

- Enable inline parameters
- Specify an ERT target
- Set the memory section for constant data to `MemVolatile` or `MemConstVolatile`

If you choose to add the `volatile` qualifier to type definitions in your generated code, note the following:

- If constant data that is qualified with `volatile` is passed by pointer, the code generator casts away the volatility. This occurs because generated functions assume that data values do not change during execution and, therefore, pass their arguments as `const *` (not `const volatile *`).
- If a variable must be declared `const` and you specify `MemVolatile`, the code generator declares the variable with the `const` and `volatile` qualifiers.
- If you set **Constants** to `MemConst` or `MemConstVolatile`, and a variable cannot be declared as constant data, a TLC warning appears and the code generator does not qualify the variable with `const`.

Consider the following simple lookup table model.



- 1 On the Configuration Parameters dialog box, in the **Optimization > Signals and Parameters** pane, select **Inline parameters**.

- 2 In the **Code Generation** pane, set **System target file** to `ert.tlc`.
- 3 In the **Code Generation > Memory Sections** pane, set **Package** to `Simulink` or `mpt`, and **Constants** to `MemConstVolatile`.
- 4 Open the Signal Properties dialog box for signal `INPUT`. On the **Code Generation** tab, set the **Package** to `Simulink` or `mpt` and the **Storage class** to `ExportedGlobal` for storing state in a global variable.
- 5 Generate code. You should see the `volatile` qualifier in the generated files `model_data.c` and `model.h`.

`model_data.c`

```

/* Constant parameters (auto storage) */
/* ConstVolatile memory section */
const volatile ConstParam_simple_lookup simple_lookup_ConstP = {
  /* Expression: [-5:5]
   * Referenced by: '<Root>/Lookup Table'
   */
  { -5.0, -4.0, -3.0, -2.0, -1.0, 0.0, 1.0, 2.0, 3.0, 4.0, 5.0 },

  /* Expression: tanh([-5:5])
   * Referenced by: '<Root>/Lookup Table'
   */
  { -0.99990920426259511, -0.999329299739067,
    -0.99505475368673046, -0.9640275800758169,
    -0.76159415595576485, 0.0, 0.76159415595576485,
    0.9640275800758169, 0.99505475368673046,
    0.999329299739067, 0.99990920426259511 }
};

```

`model.h`

```

/* Real-time Model Data Structure */
struct RT_MODEL_simple_lookup {
  const char_T * volatile errorStatus;
};

/* Constant parameters (auto storage) */
extern const volatile ConstParam_simple_lookup simple_lookup_ConstP;

```

Also note in the `model.c` file that a typedef is inserted in the `rt_Lookup` function call, removing the `volatile` qualifier.

```

/* Lookup: '<Root>/Lookup Table' incorporates:

```

```
* Inport: '<Root>/In1'  
*/  
OUTPUT = rt_Lookup(((const real_T*)  
    &simple_lookup_ConstP.LookupTable_XData[0]), 11, INPUT, ((  
    const real_T*) &simple_lookup_ConstP.LookupTable_YData[0]));
```


Apply Memory Sections

In this section...

“Assign Memory Sections to Custom Storage Classes” on page 12-13

“Apply Memory Sections to Model-Level Functions and Internal Data” on page 12-15

“Apply Memory Sections to Atomic Subsystems” on page 12-17

Assign Memory Sections to Custom Storage Classes

To assign a memory section to a custom storage class,

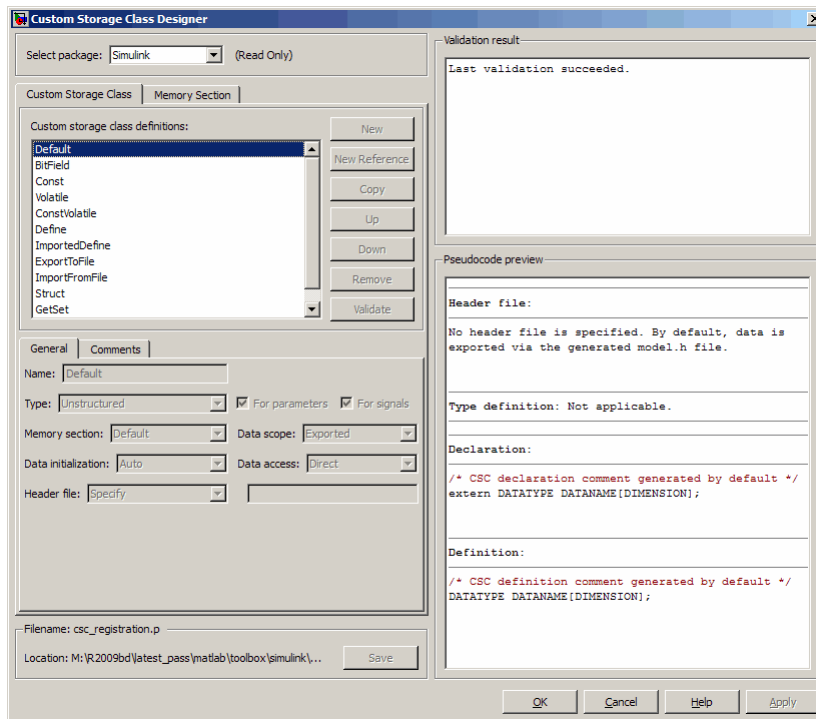
- 1 Choose **View > Model Explorer** in the model window.

The Model Explorer appears.

- 2 Choose **Tools > Custom Storage Class Designer** in the Model Explorer window.

A notification box appears that states **Please Wait ... Finding Packages**. After a brief pause, the notification box closes and the Custom Storage Class Designer appears.

- 3 Select the **Custom Storage Class** tab. The **Custom Storage Class** pane initially looks like this:



- 4 Use the **Select package** field to select a writable package. The rest of this section assumes that you have selected a writable package.
- 5 Select the desired custom storage class in the **Custom storage class definitions** pane.
- 6 Select the desired memory section from the **Memory section** pull-down.
- 7 Click **Apply** to apply changes to the open copy of the model; **Save** to apply changes and save them to disk; or **OK** to apply changes, save changes, and close the Custom Storage Class Designer.

Generated code for data definitions in the specified custom storage class are enclosed in the pragmas of the specified memory section. The pragmas can surround contiguous blocks of definitions or each definition separately, as described in “Surround Individual Definitions with Pragmas” on page 12-7. For more information, see “Design Custom Storage Classes and Memory Sections”.

Note: The code generator does not generate a `pragma` around definitions or declarations for data that has the following built-in storage classes:

- `ExportedGlobal`
- `ImportedExtern`
- `ImportedExternPointer`

The code generator treats data with these built-in storage classes like custom storage classes without a specified memory section.

Apply Memory Sections to Model-Level Functions and Internal Data

When using code generation software, you can apply memory sections to the following categories of model-level functions:

Function Category	Function Subcategory
Initialize/Terminate functions	Initialize/Start
	Terminate
Execution functions	Step functions
	Run-time initialization
	Derivative
	Enable
	Disable

When using code generation software, you can apply memory sections to the following categories of internal data:

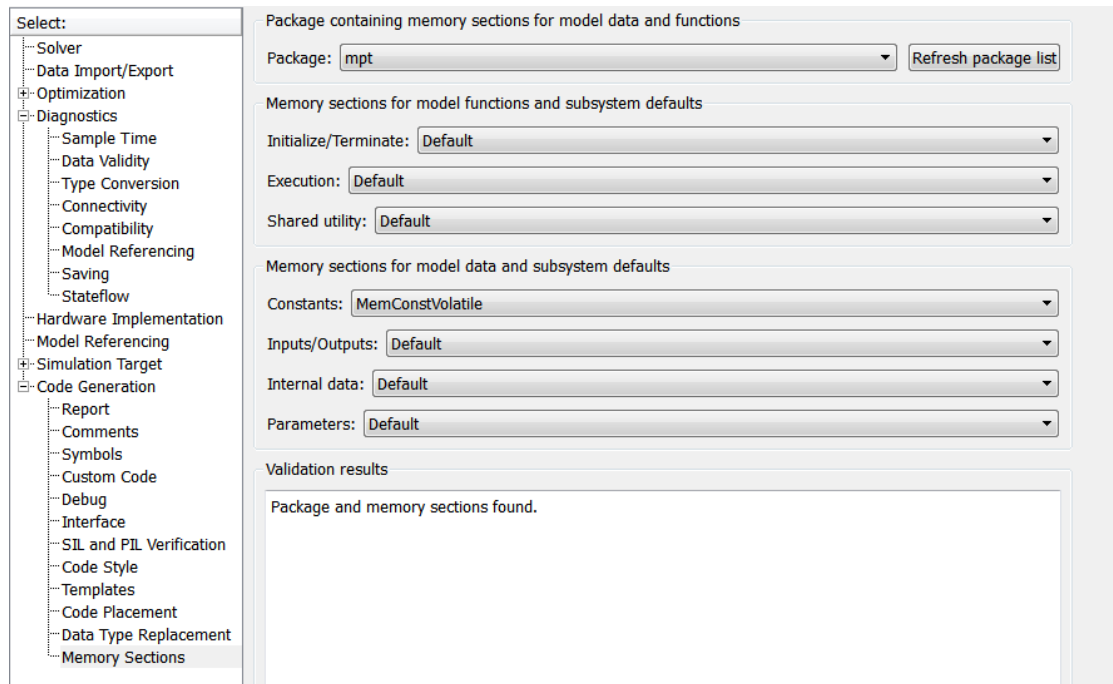
Data Category	Data Definition	Data Purpose
Constants	<code>model_constP</code>	Constant parameters
	<code>model_constB</code>	Constant block I/O
	<code>model_constZ</code>	Zero representation
	<code>model_constM</code>	Run-time model
Input/Output	<code>model_U</code>	Root inputs

Data Category	Data Definition	Data Purpose
	<i>model_Y</i>	Root outputs
Internal data	<i>model_B</i>	Block I/O
	<i>model_D</i>	D-work vectors
	<i>model_Zero</i>	Zero-crossings
Parameters	<i>model_P</i>	Parameters

Memory section specifications for model-level functions and internal data apply to the top level of the model and to its subsystems. However, these specifications are not applicable to atomic subsystems that contain overriding memory section specifications, as described in “Apply Memory Sections to Atomic Subsystems” on page 12-17.

To specify memory sections for model-level functions or internal data,

- 1 Open the Configuration Parameters dialog box and select **Code Generation > General**.
- 2 Specify the **System target file** as an ERT target, such as `ert.tlc`.
- 3 Select **Memory Sections**. The **Memory Sections** pane looks like this:



- 4 Initially, the **Package** field specifies `None` and the pull-down lists only built-in packages. 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.
- 5 In the **Package** pull-down, select the package that contains the memory sections that you want to apply.
- 6 In the pull-down for each category of internal data and model-level function, specify the memory section that you want to apply to that category. Accepting or specifying `Default` omits specifying memory section for that category.
- 7 Click **Apply** to save changes to the package and memory section selections.

Apply Memory Sections to Atomic Subsystems

For atomic subsystem whose generated code format is `Function` or `Reusable Function`, you can specify memory sections for functions and internal data that exist in that code format. Such specifications override model-level memory section specifications. Such overrides apply only to the atomic subsystem itself, not to subsystems within it.

Subsystems of an atomic subsystem inherit memory section specifications from the containing model, *not* from the containing atomic subsystem.

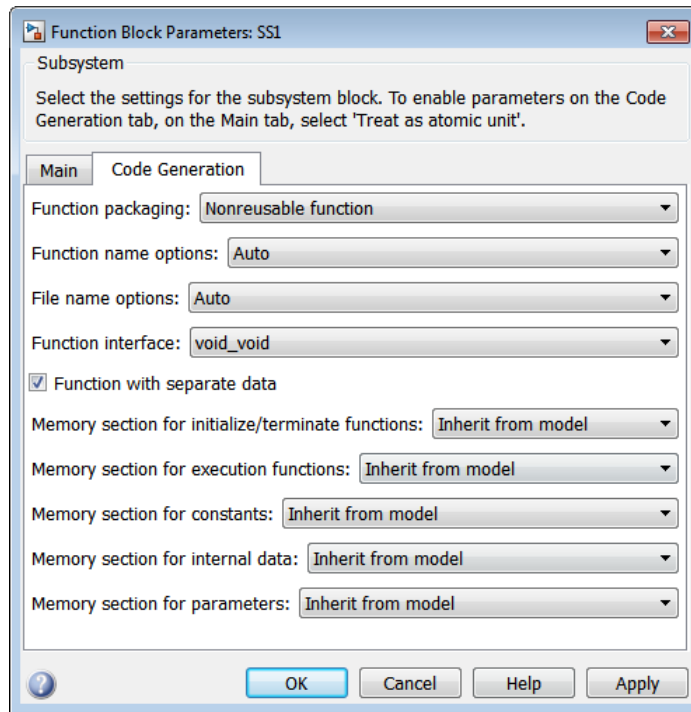
To specify memory sections for an atomic subsystem,

- 1 Right-click the subsystem in the model window.
- 2 Choose **Subsystem Parameters** from the context menu. The Function Block Parameters: *Subsystem* dialog box appears.
- 3 Select the **Treat as atomic unit** checkbox. If it is not selected, you cannot specify memory sections for the subsystem.

For an atomic system, on the **Code Generation** tab, you can use the **Function packaging** field to control the format of the generated code.

- 4 Specify **Function packaging** as **Nonreusable function** or **Reusable function**. Otherwise, you cannot specify memory sections for the subsystem.
- 5 If the code format is **Function** and you want separate data, check **Function with separate data**.

The **Code Generation** tab now shows applicable memory section options. The available options depend on the values of **Function packaging** and the **Function with separate data** check box. When the former is **Nonreusable function** and the latter is checked, the pane looks like this:



- 6 In the pull-down for each available definition category, specify the memory section that you want to apply to that category.
 - Selecting **Inherit from model** inherits the corresponding selection from the model level (not parent subsystem).
 - Selecting **Default** specifies that the category does not have an associated memory section, overriding model-level specifications for that category.
- 7 Click **Apply** to save changes, or **OK** to save changes and close the dialog box.

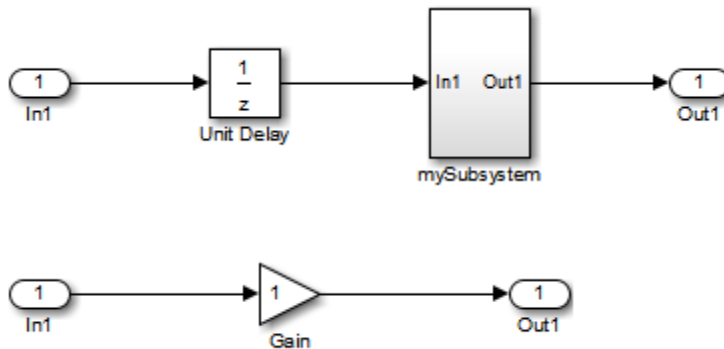
Caution If you use **Build This Subsystem** or **Build Selected Subsystem** to generate code for an atomic subsystem that specifies memory sections, the code generator ignores the subsystem-level specifications and uses the model-level specifications instead. The generated code is the same as if the atomic subsystem specified **Inherit from model** for every category of definition. For information about building subsystems, see “Generate Code and Executables for Individual Subsystem”.

It is not possible to specify the memory section for a subsystem in a library. However, you can specify the memory section for the subsystem after you have copied it into a Simulink model. This is because in the library it is unknown what code generation target will be used. You can copy a library block into many different models with different code generation targets and different memory sections available.

Generated Code with Memory Sections

Sample ERT-Based Model with Subsystem

The next figures show an ERT-based Simulink model that defines one subsystem, mySubsystem, and then the contents of that subsystem.



Assume that the subsystem is atomic. On the **Code Generation** tab, the **Function packaging** parameter is **Reusable function**. Memory sections have been created and assigned as shown in the next two tables; here, data memory sections specify **Pragma surrounds** to be **Each** variable.

Model-Level Memory Section Assignments and Definitions

Section Assignment	Section Name	Field Name	Field Value
Input/Output	MemSect1	Prepragma	#pragma IO_begin
		Postpragma	#pragma IO-end
Internal data	MemSect2	Prepragma	#pragma InData-begin(%<identifier>)
		Postpragma	#pragma InData-end
Parameters	MemSect3	Prepragma	#pragma Parameters-begin
		Postpragma	#pragma Parameters-end
Initialize/ Terminate	MemSect4	Prepragma	#pragma InitTerminate-begin
		Postpragma	#pragma InitTerminate-end

Section Assignment	Section Name	Field Name	Field Value
Execution functions	MemSect5	Prepragma	#pragma ExecFunc-begin(%<identifier>)
		Postpragma	#pragma ExecFunc-begin(%<identifier>)

Subsystem-Level Memory Section Assignments and Definitions

Section Assignment	Section Name	Field Name	Field Value
Execution functions	MemSect6	Prepragma	#pragma DATA_SEC(%<identifier>, "FAST_RAM")
		Postpragma	

Given the preceding specifications and definitions, the code generator would create the following code, with minor variations depending on the current version of the Target Language Compiler.

Model-Level Data Structures

```

#pragma IO-begin
ExternalInputs_mySample mySample_U;
#pragma IO-end

#pragma IO-begin
ExternalOutputs_mySample mySample_Y;
#pragma IO-end

#pragma InData-begin(mySample_B)
BlockIO_mySample mySample_B;
#pragma InData-end

#pragma InData-begin(mySample_DWork)
D_Work_mySample mySample_DWork;
#pragma InData-end

#pragma InData-begin(mySample_M_)
RT_MODEL_mySample mySample_M_;
#pragma InData-end

#pragma InData-begin(mySample_M)
RT_MODEL_mySample *mySample_M = &mySample_M_;
#pragma InData-end

#pragma Parameters-begin
Parameters_mySample mySample_P = {
    0.0 , {2.3}
};
#pragma Parameters-end

```

Model-Level Functions

```

#pragma ExecFunc-begin(mySample_step)
void mySample_step(void)
{
    real_T rtb_UnitDelay;
    rtb_UnitDelay = mySample_DWork.UnitDelay_DSTATE;
    mySubsystem(rtb_UnitDelay, &mySample_B.mySubsystem;,
        (rtP_mySubsystem *) &mySample_P.mySubsystem);
    mySample_Y.Out1_o = mySample_B.mySubsystem.Gain;
    mySample_DWork.UnitDelay_DSTATE = mySample_U.In1;
}

```

```
}
#pragma ExecFunc-end(mySample_step)

#pragma InitTerminate-begin
void mySample_initialize(void)
{
    rtmSetErrorStatus(mySample_M, (const char_T *)0);
    {
        ((real_T*)&mySample_B.mySubsystem.Gain)[0] = 0.0;
    }
    mySample_DWork.UnitDelay_DSTATE = 0.0;
    mySample_U.In1 = 0.0;
    mySample_Y.Out1_o = 0.0;
    mySample_DWork.UnitDelay_DSTATE = mySample_P.UnitDelay_X0;
}
#pragma InitTerminate-end
```

Subsystem Function

Because the subsystem specifies a memory section for execution functions that overrides that of the parent model, subsystem code looks like this:

```
/* File: mySubsystem.c */

#pragma DATA_SEC(mySubsystem, "FAST_RAM")
void mySubsystem(real_T rtu_In1,
rtB_mySubsystem *localB,
rtP_mySubsystem *localP)
{
    localB->Gain = rtu_In1 * localP->Gain_Gain;
}

```

If the subsystem had not defined its own memory section for execution functions, but inherited that of the parent model, the subsystem code would have looked like this:

```
/* File: mySubsystem.c */

#pragma ExecFunc-begin(mySubsystem)
void mySubsystem(real_T rtu_In1,
rtB_mySubsystem *localB,
rtP_mySubsystem *localP)
{
    localB->Gain = rtu_In1 * localP->Gain_Gain;
}

```

```
#pragma ExecFunc-end(mySubsystem)
```


Code Generation

Configuration

- “Application Objectives” on page 13-2
- “High-Level Code Generation Objectives” on page 13-3
- “Determine Model Configuration for Specified Objectives” on page 13-5
- “Check and Configure Model for Code Generation Objectives” on page 13-6
- “Check and Configure Model for Code Generation Objectives Using Configuration Parameters Dialog Box” on page 13-10
- “Configure Code Generation Objectives Programmatically” on page 13-14
- “Check Objectives in Referenced Models” on page 13-15
- “Checking Model and Configuration with Model Advisor” on page 13-16
- “Check Model During Code Generation” on page 13-17
- “Create Custom Objectives” on page 13-18
- “Code Generation Targets” on page 13-24
- “Configuration Variations” on page 13-31
- “Wizard” on page 13-32

Application Objectives

The first step in applying Embedded Coder configuration options to the application development process is to consider how your application objectives, particularly with respect to efficiency, traceability, and safety, map to code generation options in a model configuration set.

Parameters that you set in the **Solver**, **Data Import/Export**, **Diagnostics**, and **Code Generation** panes of the Configuration Parameters dialog box specify the behavior of a model in simulation and the code generated for the model.

Consider questions such as the following:

- What settings might help you debug your application?
- What is the highest objective for your application — efficiency, traceability, safety precaution, debugging, or some other criteria?
- What is the second highest objective?
- Can the objective at the start of the project differ from the objective required for the end result? What tradeoffs can you make?

After you answer these questions, you must:

- Define your objectives in the configuration set. For more information, see “High-Level Code Generation Objectives” on page 13-3.
- Use the Code Generation Advisor to identify parameter values that are not configured for the objectives that you selected. For more information, see “Determine Model Configuration for Specified Objectives” on page 13-5.

High-Level Code Generation Objectives

Depending on the type of application that your model represents, you are likely to have specific code generation objectives. For example, safety and traceability might be more critical than efficient use of memory. If you have specific objectives, you can quickly configure your model to meet those objectives by selecting and prioritizing from these code generation objectives:

- Execution efficiency (all targets) — Configure code generation settings to achieve fast execution time.
- ROM efficiency (ERT-based targets) — Configure code generation settings to reduce ROM usage.
- RAM efficiency (ERT-based targets) — Configure code generation settings to reduce RAM usage.
- Traceability (ERT-based targets) — Configure code generation settings to provide mapping between model elements and code.
- Safety precaution (ERT-based targets) — Configure code generation settings to increase clarity, determinism, robustness, and verifiability of the code.
- Debugging (all targets) — Configure code generation settings to debug the code generation build process.
- MISRA-C:2004 guidelines (ERT-based targets) — Configure code generation settings to increase compliance with MISRA-C:2004 guidelines.
- Polyspace (ERT-based targets) — Configure code generation settings to prepare the code for Polyspace[®] analysis.

Based on your objective selections and prioritization, the Code Generation Advisor checks your model and suggests changes that you can make to achieve your code generation objectives.

Note: If you select the MISRA-C:2004 guidelines code generation objective, the Code Generation Advisor checks:

- The model configuration settings for compliance with the MISRA-C:2004 configuration setting recommendations.
 - For blocks that are not supported or recommended for MISRA-C:2004 compliant code generation.
-

Setting code generation objectives and running the Code Generation Advisor provides information on how to meet code generation objectives for your model. The Code Generation Advisor does not alter the generated code. You can use the Code Generation Advisor to make the suggested changes to your model. The generated code is changed only after you modify your model and regenerate code. If you use the Code Generation Advisor to set code generation objectives and check your model, the generated code includes comments identifying which objectives you specified, the checks the Code Generation Advisor ran on the model, and the results of running the checks.

Determine Model Configuration for Specified Objectives


You can use the Code Generation Advisor to review your model and identify the parameters that are not configured for your objective. The Code Generation Advisor reviews a subset of model configuration parameters. It displays the results in the **Check model configuration settings against code generation objectives** check.

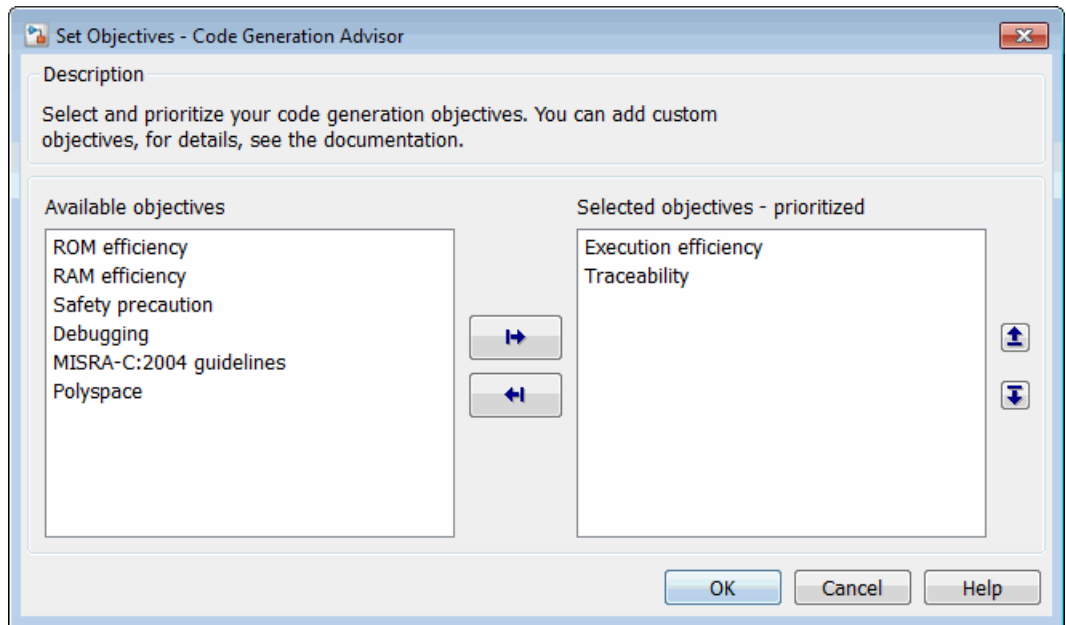
The Code Generation Advisor uses the information presented in “Recommended Settings Summary” to determine the values. When there is a conflict due to multiple objectives, the higher-priority objective takes precedence.

You can use the Code Generation Advisor to review a model before generating code, or as part of the code generation process. When you choose to review a model before generating code, you specify which model, subsystem, or referenced model the Code Generation Advisor reviews. When you choose to review a model as part of the code generation process, the Code Generation Advisor reviews the entire system.

Check and Configure Model for Code Generation Objectives

This example shows how to configure and check your model to meet code generation objectives:

- 1 On the menu bar, select **Code > C/C++ Code > Code Generation Advisor**.
Alternatively:
 - On the toolbar  drop-down list, select **Code Generation Advisor**.
 - Right-click a subsystem, and then select **C/C++ Code > Code Generation Advisor**. Go to step 3.
- 2 In the System Selector window, select the model or subsystem that you want to review, and then click **OK**.
- 3 In the Code Generation Advisor, on the **Code Generation Objectives** pane, select the code generation objectives. As you select objectives, on the left pane, the Code Generation Advisor updates the list of checks it will run on your model. If your model is configured with an ERT-based target, more objectives are available. For this example, the model is configured with an ERT-based target. If your objectives are execution efficiency and traceability, in that priority, do the following:
 - a In **Available objectives**, double-click **Execution efficiency**. **Execution efficiency** is added to **Selected objectives - prioritized**.
 - b In **Available objectives**, double-click **Traceability**. **Traceability** is added to **Selected objectives - prioritized** below **Execution efficiency**.



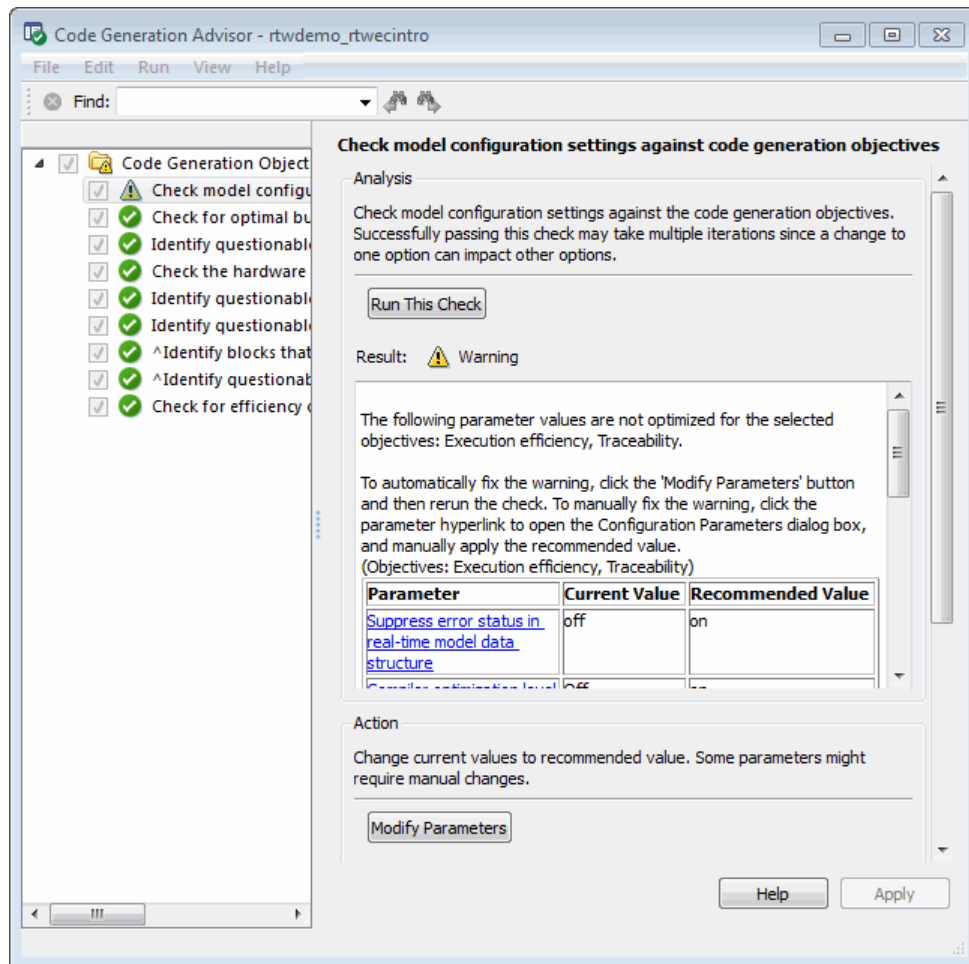
- 4 Click **Run Selected Checks** to run the checks listed in the left pane of the Code Generation Advisor.
- 5 In the Code Generation Advisor window, review the results for **Check model configuration settings against code generation objectives** by selecting it from the left pane. The results for that check are displayed in the right pane.

Check model configuration settings against code generation objectives triggers a warning for either of these reasons:

- Parameters are set to values other than the value recommended for the specified code generation objectives.
- Selected code generation objectives differ from the objectives set in the model.

Click **Modify Parameters** to set:

- Parameter to the value recommended for the specified code generation objectives.
- Code generation objectives in the model to the objectives specified in the Code Generation Advisor.



- 6 In the Code Generation Advisor window, review the results for the remaining checks by selecting them from the left pane. The right pane populates the results for the checks.
- 7 After reviewing the check results, you can choose to fix warnings and failures, as described in “Fix a Model Check Warning or Failure”.

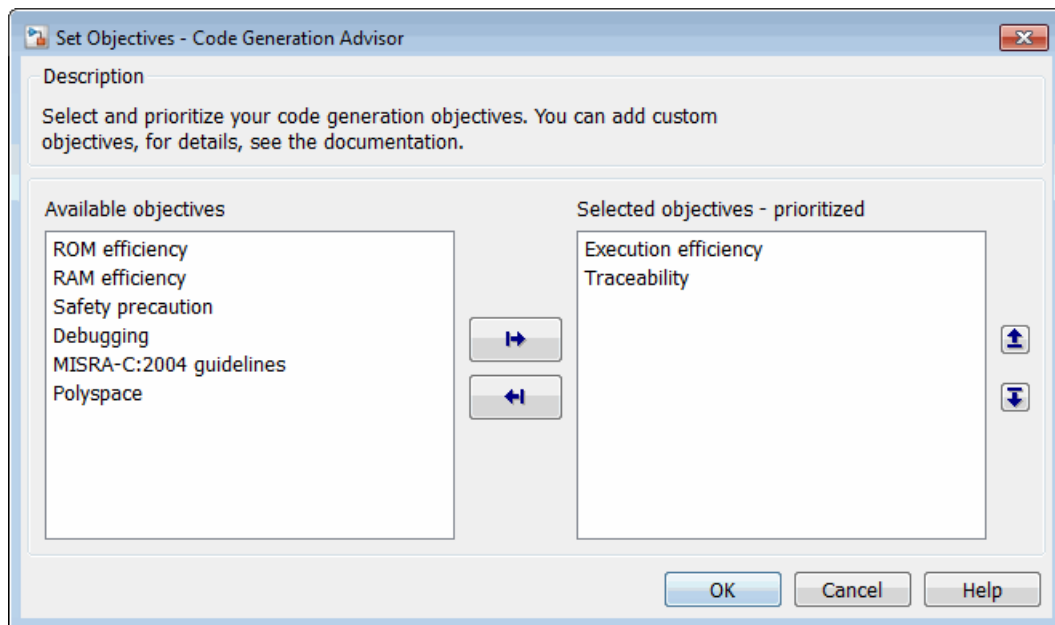
Note: When you specify an efficiency or Safety precaution objective, the Code Generation Advisor includes additional checks.

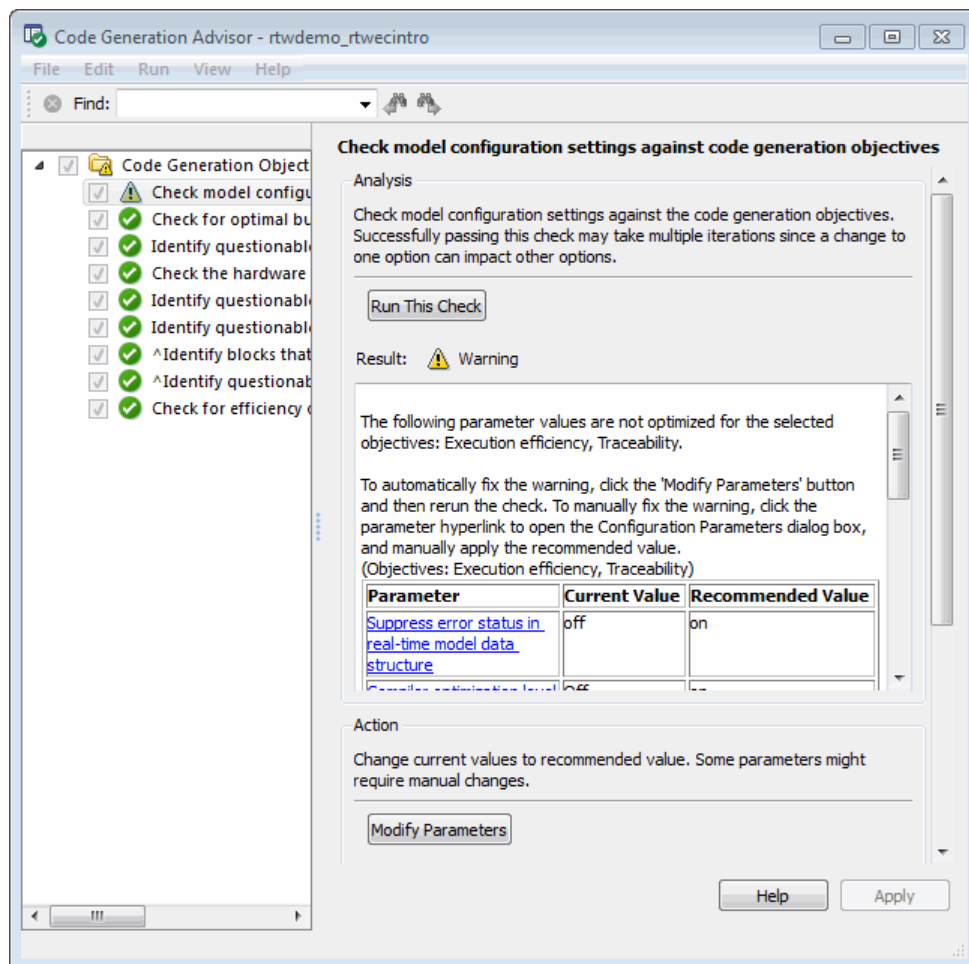
When you make changes to one check, the other check results could become invalid. You must run the checks again.

Check and Configure Model for Code Generation Objectives Using Configuration Parameters Dialog Box

This example shows how to configure and check your model to meet code generation objectives using the Configuration Parameters dialog box:

- 1 Open the Configuration Parameters dialog box. Select **Code Generation**.
- 2 Specify a system target file. If you specify an ERT-based target, more objectives are available. For this example, choose an ERT-based target such as `ert.tlc`.
- 3 Click **Set Objectives**.
- 4 In the “Set Objectives — Code Generation Advisor Dialog Box”, specify your objectives. For example, if your objectives are execution efficiency and traceability, in that priority, do the following:
 - a In **Available objectives**, double-click Execution efficiency. Execution efficiency is added to **Selected objectives - prioritized**.
 - b In **Available objectives**, double-click Traceability. Traceability is added to **Selected objectives - prioritized** below Execution efficiency.





- 8 After reviewing the check results, you can choose to fix warnings and failures, as described in “Fix a Model Check Warning or Failure”.

Note: When you specify an efficiency or Safety precaution objective, the Code Generation Advisor includes additional checks. When you make changes to one check, the other check results could become invalid and you must run the checks again.

For more information, see “Set Objectives — Code Generation Advisor Dialog Box”

Configure Code Generation Objectives Programmatically

This example shows how to configure code generation objectives by writing a MATLAB script or entering commands at the command line.

- 1 Specify a system target file. If you specify an ERT-based target, more objectives are available. For this example, specify `ert.tlc.model_name` is the name or handle to the model.

```
set_param(model_name, 'SystemTargetFile', 'ert.tlc');
```

- 2 Specify your objectives. For example, if your objectives are execution efficiency and traceability, in that priority, enter:

```
set_param(model_name, 'ObjectivePriorities', ...  
{'Execution efficiency', 'Traceability'});
```

Note: When you specify a GRT-based system target file, you can specify an objective at the command line. If you specify ROM efficiency, RAM efficiency, Traceability, MISRA-C:2004 guidelines, Polyspace, or Safety precaution, the build process changes the objective to Unspecified because you have specified a value that is invalid when using a GRT-based target.

Check Objectives in Referenced Models

When you check a model during the code generation process, you must specify the same objectives in the top model and referenced models. If you specify different objectives for the top model and referenced model, the build process generates an error.

To specify different objectives for the top model and each referenced model, review the models separately without generating code.

Checking Model and Configuration with Model Advisor

You can use the Model Advisor checks available with Embedded Coder to assess model readiness for code generation. For information about the Model Advisor, see “Run Model Checks”. For checks available with Embedded Coder, see “Embedded Coder Checks”.

If you want to check and configure your model for code generation objectives such as traceability or safety, see “Application Considerations”.

Check Model During Code Generation

This example shows how to use the Code Generation Advisor to review a model as part of the code generation process.

- 1 Specify your code generation objectives.
- 2 On the **Configuration Parameters > Code Generation > General** pane, select one of the following from **Check model before generating code**:
 - On (proceed with warnings)
 - On (stop for warnings)
- 3 If you only want to generate code, select **Generate code only**; otherwise clear the check box to build an executable.
- 4 Apply your changes, and then click **Generate Code/Build**. The Code Generation Advisor starts and reviews the top model and subsystems.

If the Code Generation Advisor issues failures or warnings, and you specified:

- On (proceed with warnings) — The Code Generation Advisor window opens while the build process proceeds. After the build process is complete, you can review the results.
 - On (stop for warnings) — The build process halts and displays the Diagnostic Viewer. To continue, you must review and resolve the Code Generation Advisor results or change the **Check model before generating code** selection.
- 5 In the Code Generation Advisor window, review the results by selecting a check from the left pane. The right pane populates the results for that check.
 - 6 After reviewing the check results, you can choose to fix warnings and failures as described in “Fix a Model Check Warning or Failure”.

Note: When you specify an efficiency or Safety precaution objective, the Code Generation Advisor includes additional checks. When you make changes to one of these checks, the other check results could become invalid and you must run the check again.

For more information, see “Set Objectives — Code Generation Advisor Dialog Box”

Create Custom Objectives

In this section...

“Specify Parameters in Custom Objectives” on page 13-18

“Specify Checks in Custom Objectives” on page 13-19

“Determine Checks and Parameters in Existing Objectives” on page 13-19

“How to Create Custom Objectives” on page 13-21

The Code Generation Advisor reviews your model based on objectives that you specify. If the predefined efficiency, traceability, Safety precaution, and debugging objectives do not meet your requirements, you can create custom objectives.

You can create custom objectives by:

- Creating a new objective and adding parameters and checks to a new objective.
- Creating a new objective based on an existing objective, then adding, modifying, and removing the parameters and checks within the new objective.

Specify Parameters in Custom Objectives

When you create a custom objective, you specify the values of configuration parameters that the Code Generation Advisor reviews. You can use the following methods:

- `addParam` — Add parameters and specify the values that the Code Generation Advisor reviews in **Check model configuration settings against code generation objectives**. When you add parameters that have dependencies, the software includes the dependencies in the list of parameter values that the Code Generation Advisor reviews.
- `modifyInheritedParam` — Modify inherited parameter values that the Code Generation Advisor reviews in **Check model configuration settings against code generation objectives**.
- `removeInheritedParam` — Remove inherited parameters from a new objective that is based on an existing objective. When a user selects multiple objectives, if another selected objective includes this parameter, the Code Generation Advisor reviews the parameter value in **Check model configuration settings against code generation objectives**.

Specify Checks in Custom Objectives

Objectives include the **Check model configuration settings against code generation objectives** check by default. When you create a custom objective, you specify the list of additional checks that are associated with the custom objective. You can use the following methods:

- **addCheck** — Add checks to the Code Generation Advisor. When a user selects the custom objective, the Code Generation Advisor displays the check, unless the user specifies an additional objective with a higher priority that excludes the check.

For example, you might add a check to the Code Generation Advisor to include a custom check in the automatic model checking process.

- **excludeCheck** — Exclude checks from the Code Generation Advisor. 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.

For example, you might exclude a check from the Code Generation Advisor when a check takes a long time to process.

- **removeInheritedCheck** — Remove inherited checks from a new objective that is based on an existing objective. When a user selects multiple objectives, if another selected objective includes this check, the Code Generation Advisor displays the check.

For example, you might remove an inherited check, rather than exclude the check, when the check takes a long time to process, but the check is important for another objective.

Determine Checks and Parameters in Existing Objectives

When you base a new objective on an existing objective, you can determine what checks and parameters the existing objective contains. The Code Generation Advisor contains the list of checks in each objective.

For example, the **Efficiency** objective includes checks which you can see in the Code Generation Advisor. To see the checks in the Code Generation Advisor:

- 1 Open the `rtwdemo_rtweintro` model.
- 2 Specify an ERT-based target.

- 3 On the model toolbar, select **Code > C/C++ Code > Code Generation Advisor**.
- 4 In the System Selector window, select the model or subsystem that you want to review, and then click **OK**.
- 5 In the Code Generation Advisor, on the **Code Generation Objectives** pane, select the code generation objectives. As you select objectives, on the left pane, the Code Generation Advisor updates the list of checks it will run on your model. For this example, select **Execution efficiency**.
 - In **Available objectives**, double-click **Execution efficiency**. **Execution efficiency** is added to **Selected objectives - prioritized**.

In the left pane, the Code Generation Advisor lists the checks for the **Execution efficiency** objective. The first check, **Check model configuration settings against code generation objectives**, lists parameters and values specified by the objective. For example, the Code Generation Advisor displays the list of parameters and the recommended values in the **Execution efficiency** objective. To see the list of parameters and values:

- 1 Run **Check model configuration settings against code generation objectives**.
- 2 Click **Modify Parameters**.
- 3 Rerun the check.

In the check results, the Code Generation Advisor displays the list of parameters and recommended values for the **Execution efficiency** objective.

Passed

(Objectives: Execution efficiency)

The following parameters have been checked and confirmed with the recommended value

Parameter	Value
non-inlined S-functions	off
Suppress error status in real-time model data structure	on
MAT-file logging	off
Classic call interface	off
continuous time	off
non-finite numbers	off
Single output/update function	on
Minimize algebraic loop occurrences	off

How to Create Custom Objectives

To create a custom objective:

- 1 Create an `sl_customization.m` file.
 - Specify custom objectives in a single `sl_customization.m` file only, or the software generates an error. This issue is true even if you have more than one `sl_customization.m` file on your MATLAB path.
 - Except for the `matlabroot/work` folder, do not place an `sl_customization.m` file in your root MATLAB folder, or its subfolders. Otherwise, the software ignores the customizations that the file specifies.
- 2 Create an `sl_customization` function that takes a single argument. When the software invokes the function, the value of this argument is the Simulink customization manager. In the function:
 - a Create a handle to the code generation objective, using the `ObjectiveCustomizer` constructor.
 - b Register a callback function for the custom objectives, using the `ObjectiveCustomizer.addCallbackObjFcn` method.
 - c Add a call to execute the callback function, using the `ObjectiveCustomizer.callbackFcn` method.

For example:

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

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

end
```

- 3 Create a MATLAB callback function that:
 - Creates code generation objective objects using the `rtw.codegenObjectives.Objective` constructor.
 - Adds, modifies, and removes configuration parameters for each objective using the `addParam`, `modifyInheritedParam`, and `removeInheritedParam` methods.

- Includes and excludes checks for each objective using the `addCheck`, `excludeCheck`, and `removeInheritedCheck` methods.
- Registers objectives using the `register` method.

The following example shows how to create an objective, `Reduce RAM Example`. `Reduce RAM Example` includes five parameters and three checks that the Code Generation Advisor reviews.

```
function addObjectives

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

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

% Add additional checks to the objective
% The Code Generation Advisor automatically includes 'Check model
% configuration settings against code generation objectives' in every
% objective.
addCheck(obj, 'mathworks.design.UnconnectedLinesPorts');
addCheck(obj, 'mathworks.design.Update');

% Register the objective
register(obj);

end
```

The following example shows you how to create an objective, `My Traceability Example`, based on the existing `Traceability` objective. The custom objective modifies, removes, and adds parameters that the Code Generation Advisor reviews. It also adds and removes checks from the Code Generation Advisor.

```
function addObjectives

% Create the custom objective from an existing objective
obj = rtw.codegenObjectives.Objective('ex_my_trace_1', 'Traceability');
setObjectiveName(obj, 'My Traceability Example');

% Modify parameters in the objective
modifyInheritedParam(obj, 'GenerateTraceReportSf', 'Off');
removeInheritedParam(obj, 'ConditionallyExecuteInputs');
addParam(obj, 'MatFileLogging', 'On');

% Modify checks in the objective
```

```
addCheck(obj, 'mathworks.codegen.SWEnvironmentSpec');  
removeInheritedCheck(obj, 'mathworks.codegen.CodeInstrumentation');  
  
%Register the objective  
register(obj);  
  
end
```

- 4** If you previously opened the Code Generation Advisor, close the model from which you opened the Code Generation Advisor.
- 5** Refresh the customization manager. At the MATLAB command line, enter the `sl_refresh_customizations` command.
- 6** Open your model and review the new objectives.

Code Generation Targets

In this section...

“About Target Selection” on page 13-24

“Select an ERT Target” on page 13-25

“Customize an ERT Target” on page 13-26

“Configure Support for Numeric Data” on page 13-26

“Configure Support for Time Values” on page 13-26

“Support for Non-inlined S-Functions” on page 13-27

“Configure Model Function Generation and Argument Passing” on page 13-27

“Set Up Support for Code Reuse” on page 13-29

“Configure a Code Replacement Library” on page 13-30

About Target Selection

The first step to configuring a model for code generation is to choose and configure a code generation target. When you select a target, other model configuration parameters change automatically to best serve requirements of the target. For example:

- Code interface parameters
- Build process parameters, such as the template make file
- Target hardware parameters, such as word size and byte ordering

Use the **Browse** button on the **Code Generation** pane to open the System Target File Browser (see “Select a Target”). The browser lets you select a preset target configuration consisting of a system target file, template makefile, and `make` command.

If you select a target configuration by using the System Target File Browser, your selection appears in the **System target file** field (`target.tlc`).

If you are using a target configuration that does not appear in the System Target File Browser, enter the name of your system target file in the **System target file** field. Click **Apply** or **OK** to configure for that target.

“Targets and Code Formats” describes the use of the browser and includes a complete list of available target configurations.

You also can select a system target file programmatically from MATLAB code, as described in “Select a System Target File Programmatically”.

After selecting a system target, you can modify model configuration parameter settings.

If you want to switch between different targets in a single workflow for different code generation purposes (for example, rapid prototyping versus product code deployment), set up different configuration sets for the same model and switch the active configuration set for the current operation. For more information on how to set up configuration sets and change the active configuration set, see “Manage a Configuration Set”.

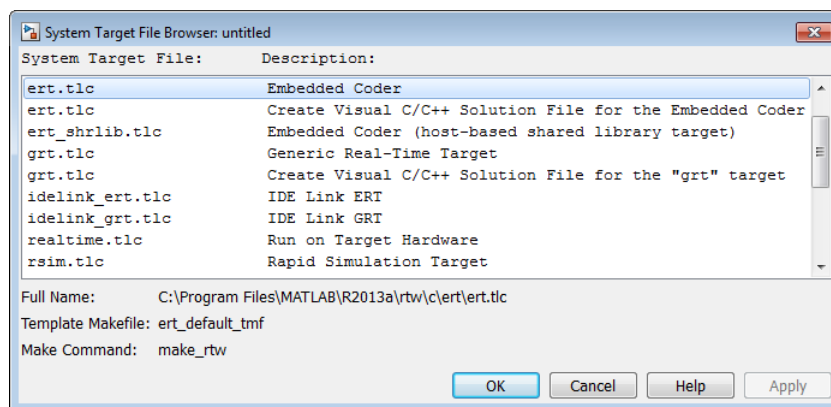
Select an ERT Target

The **Browse** button in the **Target Selection** subpane of the **Code Generation > General** pane lets you select an ERT target with the System Target File Browser. See “Targets and Code Formats” for a general discussion of target selection.

The code generator provides variants of the ERT target including the following:

- Default ERT target
- ERT target for generating and building a Visual C++[®] Solution (.sln) file for the Visual C++ IDE
- ERT target for generating a Windows[®] or UNIX[®] host-based shared library

These targets are based on a common system target file, `ert.tlc`. They are displayed in the System Target File Browser as shown in the figure below.



You can use the `ert_shrlib.tlc` target to generate a host-based shared library from your Simulink model. Selecting this target allows you to generate a shared library version of your model code for your host platform, either a Windows dynamic link library (.dll) file or a UNIX shared object (.so) file. This feature can be used to package your source code securely for easy distribution and shared use.

Customize an ERT Target

For information on customizing ERT targets, see “Target Development”.

Configure Support for Numeric Data

By default, ERT targets support code generation for integer, floating-point, nonfinite, and complex numbers.

To Generate Code that Supports...	Do...
Integer data only	Deselect Support floating-point numbers . If noninteger data or expressions are encountered during code generation, an error message reports the offending blocks and parameters.
Floating-point data	Select Support floating-point numbers .
Nonfinite values (for example, NaN, Inf)	Select Support floating-point numbers and Support nonfinite numbers .
Complex data	Select Support complex numbers .

For more information, see “Code Generation Pane: Interface”.

Configure Support for Time Values

Certain blocks require the value of absolute time (that is, the time from the start of program execution to the present time), elapsed time (for example, the time elapsed between two trigger events), or continuous time. Depending on the blocks used, you might need to adjust the configuration settings for supported time values.

To...	Select...
Generate code that creates and maintains integer counters for blocks that use	Support absolute time . For further information on the allocation and operation of absolute and elapsed timers, see “Timers”. If you do not select this parameter and the model includes block that use

To...	Select...
absolute or elapsed time values (default)	absolute or elapsed time values, the build process generates an error.
Generate code for blocks that rely on continuous time	Support continuous time. If you do not select this parameter and the model includes continuous-time blocks, the build process generates an error.

For more information, see “Code Generation Pane: Interface”.

Support for Non-inlined S-Functions

To generate code for noninlined S-Functions in a model, select **Support noninlined S-functions**. The generation of noninlined S-functions requires floating-point and nonfinite numbers. Thus, when you select **Support non-inlined S-functions**, the ERT target automatically selects **Support floating-point numbers** and **Support non-finite numbers**.

When you select **Support non-finite numbers**, the build process generates an error if the model includes a C MEX S-function that does not have a corresponding TLC implementation (for inlining code generation).

Note that inlining S-functions is highly advantageous in production code generation, for example in implementing device drivers. To enforce the use of inlined S-functions for code generation, clear **Support non-inlined S-functions**.

When generating code for a model that contains non-inlined S-functions with an ERT target and either of the following is true:

- On the **Code Generation** pane, **Generate code only** is cleared.
- On the **Hardware Implementation** pane, **Test hardware is the same as production hardware** is cleared.

There might be a mismatch between the simulation and code generation results. To avoid such a mismatch select **Test hardware is the same as production hardware** or select **Generate code only**.

Configure Model Function Generation and Argument Passing

For ERT targets, you can configure how a model's functions are generated and how arguments are passed to the functions.

To...	Do...
Generate model function calls that are compatible with the main program module of the pre-R2012a GRT target (<code>grt_main.c</code> or <code>.cpp</code>)	Select Classic call interface and MAT-file logging . In addition, deselect Suppress error status in real-time model data structure . Classic call interface provides a quick way to use code generated in R2012a or higher with a pre-R2012a GRT-based custom target by generating wrapper function calls that interface to the generated code.
Reduce overhead and use more local variables by combining the output and update functions in a single <code>model_step</code> function	Select Single output/update function Errors or unexpected behavior can occur if a Model block is part of a cycle and “Single output/update function” is enabled (the default). See “Model Blocks and Direct Feed through” for details.
Generate a <code>model_terminate</code> function for a model not designed to run indefinitely	Select Terminate function required . For more information, see the description of <code>model_terminate</code> .
Generate reusable, reentrant code from a model or subsystem	Select Generate reusable code . See “Set Up Support for Code Reuse” on page 13-29 for details.
Statically allocate model data structures and access them directly in the model code	Deselect Generate reusable code . The generated code is not reusable or reentrant. See “Entry-Point Functions and Scheduling” for information on the calling interface generated for model functions in this case.
Suppress the generation of an error status field in the real-time model data structure, <code>rtModel</code> , for example, if you do not need to log or monitor error messages	Select Suppress error status in real-time model data structure . Selecting this parameter can also cause the <code>rtModel</code> structure to be omitted completely from the generated code. When generating code for multiple integrated models, set this parameter the same for all of the models. Otherwise, the integrated application might exhibit unexpected behavior. For example, if you select the option in one model but not in another, the error status might not be registered by the integrated application. Do not select this parameter if you select the MAT-file logging option. The two options are incompatible.

To...	Do...
Open the Model Step Functions dialog box preview and modify the model's <i>model_step</i> function prototype (see “Entry-Point Functions and Scheduling”)	Click Configure Step Function . Based on the Function specification value you select for your <i>model_step</i> function (supported values include Default model-step function and Model specific C prototype), you can preview and modify the function prototype. Once you validate and apply your changes, you can generate code based on your function prototype modifications. For more information about using the Configure Step Function button and the Model Step Functions dialog box, see “Function Prototype Control”.

For more information, see “Code Generation Pane: Interface”.

Set Up Support for Code Reuse

For ERT targets, you can configure how a model reuses code using the **Generate reusable code** parameter.

Pass root-level I/O as provides options that control how model inputs and outputs at the root level of the model are passed to the *model_step* function.

To...	Select...
Pass each root-level model input and output argument to the <i>model_step</i> function individually (the default)	Generate reusable code and Pass root-level I/O as > Individual arguments.
Pack root-level input arguments and root-level output arguments into separate structures that are then passed to the <i>model_step</i> function	Generate reusable code and Pass root-level I/O as > Structure reference

In some cases, selecting **Generate reusable code** can generate code that compiles but is not reentrant. For example, if a signal, **DWork** structure, or parameter data has a storage class other than **Auto**, global data structures are generated. To handle such cases, use the **Reusable code error diagnostic** parameter to choose the severity levels for diagnostics.

In some cases, the Embedded Coder software is unable to generate valid and compilable code. For example, if the model contains one of the following, the code generated would be invalid.

- An S-function that is not code-reuse compliant
- A subsystem triggered by a wide function call trigger

In these cases, the build terminates after reporting the problem.

For more information, see “Code Generation Pane: Interface”.

Configure a Code Replacement Library

You can configure the code generator to change the code that it generates for functions and operators such that the code meets application requirements. Configure the code generator to apply a code replacement library (CRL) during code generation. If you have an Embedded Coder license, you can develop and apply custom code replacement libraries.

For more information about replacing code, using code replacement libraries that MathWorks provides, see “What Is Code Replacement?” and “Code Replacement Libraries”. For information about developing code replacement libraries, see “What Is Code Replacement Customization?” in the Embedded Coder documentation.

Configuration Variations

Every model contains one or more named configuration sets that specify model parameters such as solver options, code generation options, and other choices. A model can contain multiple configuration sets, but only one configuration set is active at a time. For more information on configuration sets and how to view and edit them in the Configuration Parameters dialog box, see “About Model Configurations”.

A configuration set includes options that specify code generation in general. For more information, see “Configure a Model for Code Generation”. With Embedded Coder and an ERT target more parameters are available for fine-tuning the generated code with respect to customizing the appearance and optimizing the generated code.

Multiple configuration sets can be useful in embedded systems development. By defining multiple configuration sets in a model, you can easily retarget code generation from that model. For example, one configuration set might specify the default ERT target with external mode support enabled for rapid prototyping, while another configuration set might specify the ERT-based target for Visual C++ to generate production code for deployment of the application. Activation of either configuration set fully reconfigures the model for that type of code generation.

Wizard

In this section...

“Configure and Optimize Model with Configuration Wizard Blocks” on page 13-32

“Add a Configuration Wizard Block” on page 13-33

“Use Configuration Wizard Blocks” on page 13-35

“Create a Custom Configuration Wizard Block” on page 13-35

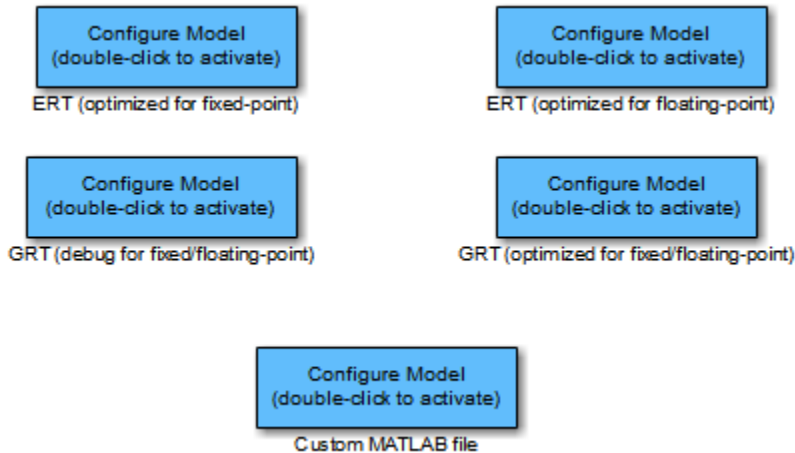
Configure and Optimize Model with Configuration Wizard Blocks

The Embedded Coder software provides a library of *Configuration Wizard* blocks and scripts to help you configure and optimize code generation from your models quickly and easily.

The library provides a Configuration Wizard block you can customize, and four preset Configuration Wizard blocks.

Block	Description
Custom MATLAB file	Automatically update active configuration parameters of parent model using a custom file
ERT (optimized for fixed-point)	Automatically update active configuration parameters of parent model for ERT fixed-point code generation
ERT (optimized for floating-point)	Automatically update active configuration parameters of parent model for ERT floating-point code generation
GRT (debug for fixed/floating-point)	Automatically update active configuration parameters of parent model for GRT fixed- or floating-point code generation with debugging enabled
GRT (optimized for fixed/floating-point)	Automatically update active configuration parameters of parent model for GRT fixed- or floating-point code generation

These are shown in the figure below.



When you add one of the preset Configuration Wizard blocks to your model and double-click it, a predefined MATLAB file script executes and configures parameters of the model's active configuration set without manual intervention. The preset blocks configure the options optimally for one of the following cases:

- Fixed-point code generation with the ERT target
- Floating-point code generation with the ERT target
- Fixed/floating-point code generation with TLC debugging options enabled, with the GRT target.
- Fixed/floating-point code generation with the GRT target

The Custom block is associated with an example MATLAB file script that you can adapt to your requirements.

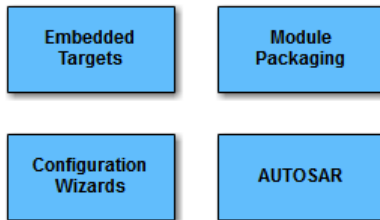
You can also set up the Configuration Wizard blocks to invoke the build process after configuring the model.

Add a Configuration Wizard Block

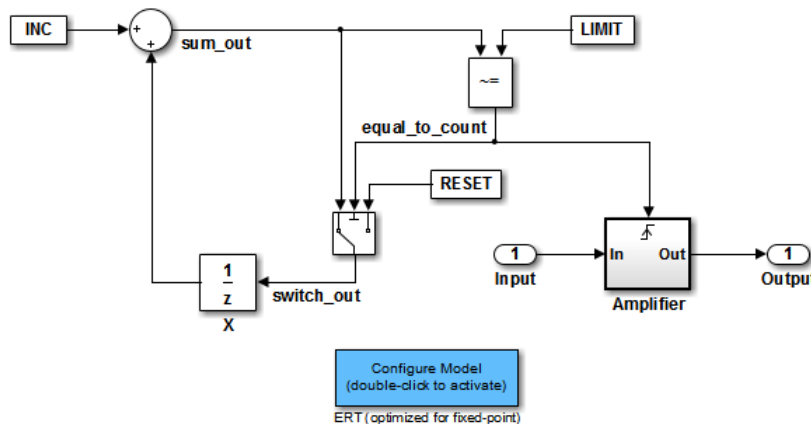
This section describes how to add one of the preset Configuration Wizard blocks to a model.

The Configuration Wizard blocks are available in the Embedded Coder block library. To use a Configuration Wizard block:

- 1 Open the model that you want to configure.
- 2 Open the Embedded Coder library by typing the command `rtweclib`.
- 3 The top level of the library is shown below.



- 4 Double-click the Configuration Wizards icon. The Configuration Wizards sublibrary opens.
- 5 Select the Configuration Wizard block you want to use and drag and drop it into your model. In the figure below, the ERT (optimized for fixed-point) Configuration Wizard block has been added to the model.



- 6 You can set up the Configuration Wizard block to invoke the build process after executing its configuration script. If you do not want to use this feature, skip to the next step.

If you want the Configuration Wizard block to invoke the build process, right-click on the Configuration Wizard block in your model, and select **Mask > Mask Parameters...** from the context menu. Then, select the **Invoke build process after configuration** parameter.

- 7 Click **Apply**, and close the Mask Parameters dialog box.

Note You should not change the **Configure the model for** option, unless you want to create a custom block and script. In that case, see “Create a Custom Configuration Wizard Block” on page 13-35.

- 8 Save the model.
- 9 You can now use the Configuration Wizard block to configure the model, as described in the next section.

Use Configuration Wizard Blocks

Once you have added a Configuration Wizard block to your model, just double-click the block. The script associated with the block automatically sets parameters of the active configuration set that are relevant to code generation (including selection of the target). You can verify that the options have changed by opening the Configuration Parameters dialog box and examining the settings.

If the **Invoke build process after configuration** option for the block was selected, the script also initiates the code generation and build process.

Note: You can add more than one Configuration Wizard block to your model. This provides a quick way to switch between configurations.

Create a Custom Configuration Wizard Block

The Custom Configuration Wizard block is shipped with an associated MATLAB file script, `rtwsampleconfig.m`. The script is located in the folder `matlabroot/toolbox/rtw/rtw`.

Both the block and the script are intended to provide a starting point for customization. This section describes:

- How to create a custom Configuration Wizard block linked to a custom script.
- Operation of the example script, and programming conventions and requirements for a customized script.
- How to run a configuration script from the MATLAB command line (without a block).

Setting Up a Configuration Wizard Block

This section describes how to set up a custom Configuration Wizard block and link it to a script. If you want to use the block in more than one mode, it is advisable to create a Simulink library to contain the block.

To begin, make a copy of the example script for later customization:

- 1 Create a folder to store your custom script. This folder should not be anywhere inside the MATLAB folder structure (that is, it should not be under *matlabroot*).

The discussion below refers to this folder as */my_wizards*.

- 2 Add the folder to the MATLAB path. Save the path for future sessions.
- 3 Copy the example script (*matlabroot/toolbox/rtw/rtw/rtwsampleconfig.m*) to the */my_wizards* folder you created in the previous steps. Then, rename the script as desired. The discussion below uses the name *my_configscript.m*.
- 4 Open the example script into the MATLAB editor. Scroll to the end of the file and enter the following line of code:

```
disp('Custom Configuration Wizard Script completed.');
```

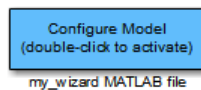
This statement is used later as a test to verify that your custom block has executed the script.

- 5 Save your script and close the MATLAB editor.

The next step is to create a Simulink library and add a custom block to it. Do this as follows:

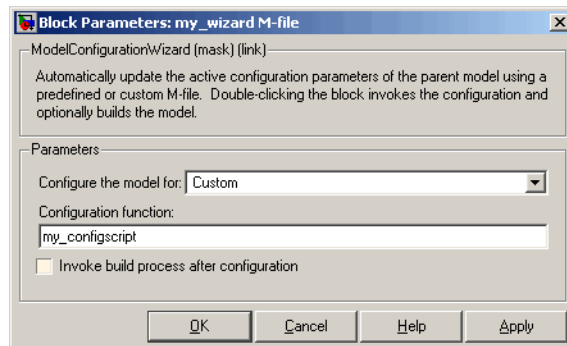
- 1 Open the Embedded Coder library and the Configuration Wizards sublibrary, as described in “Add a Configuration Wizard Block” on page 13-33.
- 2 Select **New > Library** from the **File** menu of the Configuration Wizards sublibrary window. An empty library window opens.
- 3 Select the Custom MATLAB file block from the Configuration Wizards sublibrary and drag and drop it into the empty library window.

- 4 To distinguish your custom block from the original, edit the Custom MATLAB file label under the block as desired.
- 5 Select **Save as** from the **File** menu of the new library window; save the library to the `/my_wizards` folder, under your library name of choice. In the figure below, the library has been saved as `ex_custom_button`, and the block has been labeled `my_wizard MATLAB-file`.



The next step is to link the custom block to the custom script:

- 1 Right-click on the block in your model, and select **Mask > Mask Parameters** from the context menu. Notice that the **Configure the model for** menu is set to **Custom**. When **Custom** is selected, the **Configuration function** edit field is enabled, so that you can enter the name of a custom script.
- 2 Enter the name of your custom script into the **Configuration function** field. (Do not enter the `.m` filename extension, which is implicit.) In the figure below, the script name `my_configscript` has been entered into the **Configuration function** field. This establishes the linkage between the block and script.



- 3 Note that by default, the **Invoke build process after configuration** option is deselected. You can change the default for your custom block by selecting this option. For now, leave this option deselected.
- 4 Click **Apply** and close the Mask Parameters dialog box.
- 5 Save the library.

- 6 Close the Embedded Coder library and the Configuration Wizards sublibrary. Leave your custom library open for use in the next step.

Now, test your block and script in a model. Do this as follows:

- 1 Open the `vdp` model by typing the command:
`vdp`
- 2 Open the Configuration Parameters dialog box and view the options by clicking on **Code Generation** in the list in the left pane of the dialog box.
- 3 Observe that `vdp` is configured, by default, for the GRT target. Close the Configuration Parameters dialog box.
- 4 Select your custom block from your custom library. Drag and drop the block into the `vdp` model.
- 5 In the `vdp` model, double-click your custom block.
- 6 In the MATLAB window, you should see the test message you previously added to your script:

```
Custom Configuration Wizard Script completed.
```

This indicates that the custom block executed the script.

- 7 Reopen the Configuration Parameters dialog box and view the **Code Generation** pane again. You should now see that the model is configured for the ERT target.

Before applying further edits to your custom script, proceed to the next section to learn about the operation and conventions of Configuration Wizard scripts.

Create a Configuration Wizard Script

You should create your custom Configuration Wizard script by copying and modifying the example script, `rtwsampleconfig.m`. This section provides guidelines for modification.

The Configuration Function

The example script implements a single function without a return value. The function takes a single argument `CS`:

```
function rtwsampleconfig(cs)
```

The argument `CS` is a handle to a proprietary object that contains information about the model's active configuration set. The Simulink software obtains this handle and passes

it in to the configuration function when the user double-clicks a Configuration Wizard block.

Your custom script should conform to this prototype. Your code should use `cs` as a “black box” object that transmits information to and from the active configuration set, using the accessor functions described below.

Access Configuration Set Parameters

To set parameters or obtain parameter values, use the Simulink `set_param` and `get_param` functions.

Option names are passed in to `set_param` and `get_param` as strings specifying an *internal option name*. The internal option name may not correspond to the option label on the GUI (for example, the Configuration Parameters dialog box). The example configuration accompanies each `set_param` and `get_param` call with a comment that correlates internal option names to GUI option labels. For example:

```
set_param(cs, 'LifeSpan', '1'); % Application lifespan (days)
```

To obtain the current setting of an option in the active configuration set, call `get_param`. Pass in the `cs` object as the first argument, followed by the internal option name. For example, the following code excerpt tests the setting of the **Create code generation report** option:

```
if strcmp(get_param(cs, 'GenerateReport'), 'on')
    ...
```

To set an option in the active configuration set, call `set_param`. Pass in the `cs` object as the first argument, followed by one or more parameter/value pairs that specify the internal option name and its value. For example, the following code excerpt turns off the **Support absolute time** option:

```
set_param(cs, 'SupportAbsoluteTime', 'off');
```

Select a Target

A Configuration Wizard script must select a target configuration. The example script uses the ERT target as a default. The script first stores string variables that correspond to the required **System target file**, **Template makefile**, and **Make command** settings:

```
stf = 'ert.tlc';
```

```
tmf = 'ert_default_tmf';  
mc  = 'make_rtw';
```

The system target file is selected by passing the `cs` object and the `stf` string to the `switchTarget` function:

```
switchTarget(cs,stf,[]);
```

The template makefile and make command options are set by `set_param` calls:

```
set_param(cs,'TemplateMakefile',tmf);  
set_param(cs,'MakeCommand',mc);
```

To select a target, your custom script needs only to set up the string variables `stf`, `tmf`, and `mc` and pass them to the calls, as above.

Obtain Target and Configuration Set Information

The following utility functions and properties are provided so that your code can obtain information about the current target and configuration set, with the `cs` object:

- `isValidParam(cs, 'option')`: The `option` argument is an internal option name. `isValidParam` returns true if `option` is a valid option in the context of the active configuration set.
- `getPropEnabled(cs, 'option')`: The `option` argument is an internal option name. Returns true if this `option` is enabled (that is, writable).
- `IsERTTarget` property: Your code can detect whether or not the currently selected target is derived from the ERT target is selected by checking the `IsERTTarget` property, as follows:

```
isERT = strcmp(get_param(cs,'IsERTTarget'),'on');
```

This information can be used to determine whether or not the script should configure ERT-specific options, for example:

```
if isERT  
    set_param(cs,'ZeroExternalMemoryAtStartup','off');  
    set_param(cs,'ZeroInternalMemoryAtStartup','off');  
    set_param(cs,'InitFltsAndDbIsToZero','off');  
    set_param(cs,'InlinedParameterPlacement',...  
              'NonHierarchical');  
    set_param(cs,'NoFixptDivByZeroProtection','on')  
end
```


Invoke a Configuration Wizard Script from the MATLAB Command Prompt

Configuration Wizard scripts can be run from the MATLAB command prompt. (The Configuration Wizard blocks are provided as a graphical convenience, but are not essential.)

Before invoking the script, you must open a model and instantiate a `CS` object to pass in as an argument to the script. After running the script, you can invoke the build process with the `rtwbuild` command. The following example opens, configures, and builds a model.

```
open my_model;  
cs = getActiveConfigSet ('my_model');  
rtwsampleconfig(cs);  
rtwbuild('my_model');
```


Code Appearance

- “Add Custom Comments to Generated Code” on page 14-3
- “Add Custom Comments for Signal or Parameter Identifiers” on page 14-5
- “Add Global Comments” on page 14-7
- “Specify Comment Style” on page 14-13
- “Customize Generated Identifier Naming Rules” on page 14-14
- “Identifier Format Control” on page 14-16
- “Control Name Mangling in Generated Identifiers” on page 14-19
- “Avoid Identifier Name Collisions with Referenced Models” on page 14-21
- “Maintain Traceability for Generated Identifiers” on page 14-22
- “Exceptions to Identifier Formatting Conventions” on page 14-23
- “Identifier Format Control Parameters Limitations” on page 14-24
- “Specify Simulink Data Object Naming Rules” on page 14-26
- “Control Code Style” on page 14-27
- “Customize Code Organization and Format” on page 14-35
- “Specify Templates For Code Generation” on page 14-37
- “Code Generation Template (CGT) Files” on page 14-38
- “Custom File Processing (CFP) Templates” on page 14-42
- “Change the Organization of a Generated File” on page 14-44
- “Generate Source and Header Files with a Custom File Processing (CFP) Template” on page 14-46
- “Comparison of a Template and Its Generated File” on page 14-54
- “Code Template API Summary” on page 14-58
- “Generate Custom File and Function Banners” on page 14-61
- “Template Symbols and Rules” on page 14-69
- “Code Annotation for Justifying Polyspace Checks” on page 14-77

- “Manage Placement of Data Definitions and Declarations” on page 14-79
- “Specify Delimiter for #Includes” on page 14-105

Add Custom Comments to Generated Code

You can include auto-generated comments in the generated code as described in “Configure Code Comments”. For ERT targets, include additional custom comments by setting parameters on the **Code Generation > Comments** pane in the Configuration Parameters dialog box. With these parameters, you can enable or suppress generation of descriptive information in comments for blocks and other model elements.

Goal	Specify
Include the text specified in the Description field of a block's Block Properties dialog box as comments in the code generated for each block.	Simulink block descriptions
Add a comment that includes the block name at the start of the code for each block.	Simulink block descriptions
Include the text specified in the Description field of a Simulink data object (such as a signal, parameter, data type, or bus) in the Simulink Model Explorer as comments in the code generated for each object.	Simulink data object descriptions
Include comments just above signals and parameter identifiers in the generated code as specified in the MATLAB or TLC function.	Custom comments (MPT objects only)
Include the text specified in the Description field in the Properties dialog box for a Stateflow object as comments just above the code generated for each object.	Stateflow object descriptions
Include requirements assigned to Simulink blocks in the generated code comments (for more information, see “Generate Code for Models with Requirements Links”).	Requirements in block comments

When you select **Simulink block descriptions**:

- The description text for blocks and Stateflow objects and block names generated as comments can include international (non-US-ASCII) characters. For details on international character support, see “International Character Support”.

- The code generation software automatically inserts comments into the generated code for custom blocks. Therefore, you do not need to include block comments in the associated TLC file for a custom block.

Note: If you have existing TLC files with manually inserted comments for block descriptions, the code generation process emits these comments instead of the automatically generated comments. Consider removing existing block comments from your TLC files. Manually inserted comments might be poorly formatted in the generated code and code-to-model traceability might not work.

- For virtual blocks or blocks that have been removed due to block reduction, comments are not generated.

For more information, see “Code Generation Pane: Comments”.

Add Custom Comments for Signal or Parameter Identifiers

This example shows you how to add a comment just above a signal or parameter identifier in the generated code. Do the following:

- 1 Write a MATLAB or TLC function and save it in a `.m` or `.tlc` file
- 2 In the Configuration Parameters dialog box, on the **Code Generation > Comments** pane, select the **Custom comments (MPT objects only)** check box.
- 3 In the **Custom comments function field**, select the `.m` or `.tlc` file.

You can include some or all of the property values for the data object. Each Simulink signal or parameter data object has properties, as described in “Parameter and Signal Property Values”. This example comment contains some of the property values for the data object MAP as specified on the Model Explorer:

```
/*      DocUnits:          PSI                               */
/*      Owner:              */                               */
/*      DefinitionFile: specialDef                          */
real_T MAP = 0.0;
```

You can type text in the **Description** field in the Model Explorer for a signal or parameter data object. If you select the **Simulink data object descriptions** check box on the **Comments** pane in the Configuration Parameters dialog box, this text appears beside the signal or parameter identifier in the generated code as a comment. For example, typing **Manifold Absolute Pressure** in the **Description** field for the data object MAP results in the following in the generated code:

```
real_T MAP = 0.0;      /* Manifold Absolute Pressure */
```

To add a comment just above a signal or parameter identifier in the generated code:

- 1 The signal or parameter MPT object must use a custom storage class. Open the MPT object properties dialog box and confirm that the **Storage class** is a custom storage class ((**Custom**) suffixed to its name). The default storage class for an MPT object is **Global (Custom)**.
- 2 Write a MATLAB or TLC function that places comments in the generated files. An example `.m` file named `rtwdemo_comments_mptfun.m` is provided in the `matlab/toolbox/rtw/rtwdemos` folder.

The MATLAB function must have three arguments that correspond to `objectName`, `modelName`, and `request`, respectively. The TLC function must have three

arguments that correspond to `objectName`, `modelName`, and `request`, respectively. For the TLC file, you can use the library function `LibGetSLDataObjectInfo` to get every property value of the data object.

- 3 Save the function as a `.m` file or a `.tlc` file and place it in a folder in the MATLAB path.
- 4 Open the model and the Configuration Parameters dialog box.
- 5 On the left pane, under **Code Generation**, click **Comments**.
- 6 In the **Comments** pane, on the right, select the **Custom comments (MPT objects only)** check box.
- 7 In the **Custom comments function** field, type the file name of the `.m` file or `.tlc` file that you created.
- 8 Click **Apply**.
- 9 Click **Generate Code**.
- 10 Open the generated files and inspect their content to verify that the comments are what you want.

Add Global Comments

In this section...

“Use a Simulink DocBlock to Add a Comment” on page 14-7

“Use a Simulink Annotation to Add a Comment” on page 14-10

“Use a Stateflow Note to Add a Comment” on page 14-10

“Use Sorted Notes to Add Comments” on page 14-11

The following examples show how to add a global comment to a Simulink model so that the comment text appears in the generated file or files where you want. Specify a template symbol name with a Simulink DocBlock, a Simulink annotation, or a Stateflow note. You can also use a sorted-notes capability that works with Simulink annotations or Stateflow notes (but not DocBlocks). For more information about template symbols, see “Template Symbols and Rules” on page 14-69.

Note Template symbol names `Description` and `ModifiedHistory` also are fields in the Model Properties dialog box. If you use one of these symbol names for global comment text, and its Model Properties field also has text in it, both names appear in the generated files.

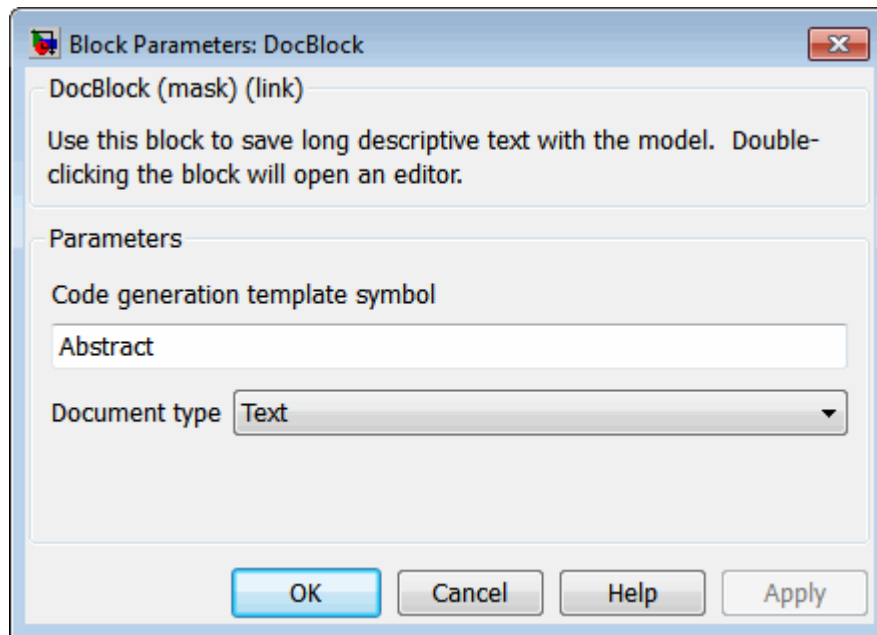
Use a Simulink DocBlock to Add a Comment

- 1 With the model open, from the **View** menu, select **Library Browser**.
- 2 Drag the DocBlock from **Model-Wide Utilities** in the Simulink library into the model.
- 3 Double-click the DocBlock and type the comment that you want in the editor. Save and close the editor.
- 4 Right-click the DocBlock and select **Mask > Mask Parameters**.
- 5 In the **Code generation template symbol** box, type one of the following:
 - `Abstract`
 - `Description`
 - `History`
 - `ModifiedHistory`

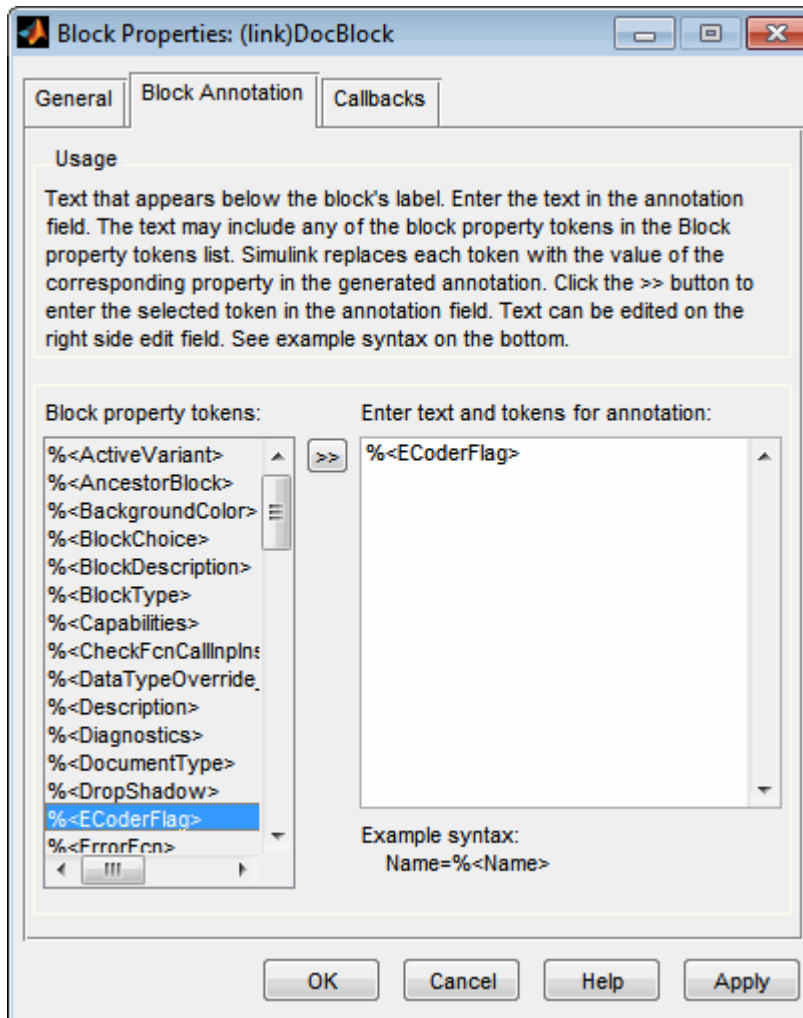
- Notes

Click **OK**. Template symbol names are case sensitive.

If you are using a DocBlock to add comments to your code, set the **Document type** to **Text**. If you set **Document type** to **RTF** or **HTML**, your comments will not appear in the code.



- 6 In the Block Properties dialog box, on the **Block Annotation** tab, select `<ECodeFlag>` and click **OK**. The symbol name that you typed in the previous step now appears under the DocBlock in the model.



- 7 Save the model. After you generate code, the code generator places the comment in each generated file whose template has the symbol name that you typed. The code generator places the comment in the generated file at the location that corresponds to where the symbol name is located in the template file.
- 8 To add more comments to the generated files, repeat steps 1–7.

Use a Simulink Annotation to Add a Comment

- 1 Double-click the unoccupied area on the model where you want to place the comment. See “Annotations”.
- 2 Type `<S:Symbol_name>` followed by the comment. `Symbol_name` is one of the following:
 - Abstract
 - Description
 - History
 - ModifiedHistory
 - Notes

For example, type `<S:Description>This is the description I want.` Template symbol names are case sensitive. (The "S" before the colon indicates "symbol.") If you want the code generator to sort multiple comments for the **Notes** symbol name, replace the next step with “Use Sorted Notes to Add Comments” on page 14-11.

- 3 Click outside the rectangle and save the model. After you generate code, the code generator places the comment in each generated file whose template has the symbol name that you typed. The code generator places the comment in the generated file at the location that corresponds to where the symbol name is located in the template file. If you want the code generator to sort multiple comments for the **Notes** symbol name, replace the next step with “Use Sorted Notes to Add Comments” on page 14-11.
- 4 To add one or more other comments to the generated files, repeat steps 1–3.

Use a Stateflow Note to Add a Comment

- 1 Right-click the unoccupied area on the Stateflow chart where you want to place the comment.
- 2 Select the annotation icon from the palette.
- 3 Type `<S:Symbol_name>` followed by the comment. `Symbol_name` is one of the following:
 - Abstract
 - Description

- History
- ModifiedHistory
- Notes

For example, type `<S:Description>This is the description I want.` Template symbol names are case sensitive. If you want the code generator to sort multiple comments for the `Notes` symbol name, replace the next step with “Use Sorted Notes to Add Comments” on page 14-11.

- 4 Click outside the note and save the model. After you generate code, the code generator places the comment in each generated file whose template has the symbol name that you typed. The code generator places the comment in the generated file at the location that corresponds to where the symbol name is located in the template file.
- 5 To add one or more other comments to the generated files, repeat steps 1–4.

Use Sorted Notes to Add Comments

The sorted-notes capability allows you to add automatically sorted comments to the generated files. The code generator places these comments in each generated file at the location that corresponds to where the `Notes` symbol is located in the template file.

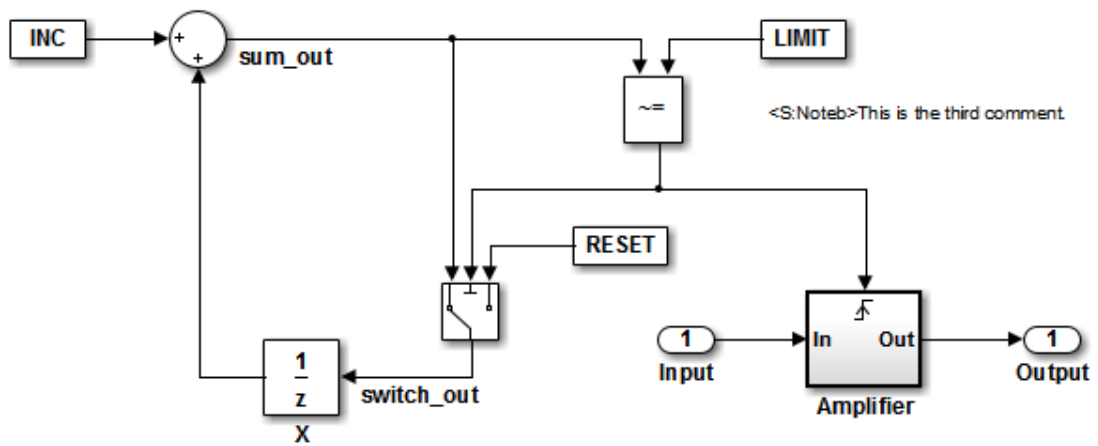
The code generator uses the following sorting order:

- Numbers before letters.
- Among numbers, 0 is first.
- Among letters, uppercase are before lowercase.

You can use sorted notes with a Simulink annotation or a Stateflow note, but not with a DocBlock.

- In the Simulink annotation or the Stateflow note, type `<S:NoteY>` followed by the first comment. Y is a number or a letter.
- Repeat for as many additional comments you want. Replace Y with a subsequent number or letter.

The figure illustrates sorted notes on a model, and where the code generator places each note in a generated file.



<S:Note2>This is the second comment I want under Notes.

<S:Note1>This is the first comment I want associated with the Notes symbol

The relevant fragment from the generated file for this model is:

** NOTES

** Note1: This is the first comment I want associated with the Notes symbol.
 Note2: This is the second comment I want under Notes.
 Noteb: This is the third comment.

**

Specify Comment Style

For ERT-based models, the comment style used in generated code is determined by the programming language selected for the model:

- C code uses `/*...*/` notation for both single-line and multiple-line comments.
- C++ code uses `//...` notation and contains only single-line comments.

If you have an Embedded Coder license, you can modify the comment style for generated code using the command-line parameter `CommentStyle`. The parameter takes the following values:

Value	Description
Auto (default)	For C, generate single or multiple-line comments delimited by <code>/*</code> and <code>*/</code> . For C++, generate single-line comments preceded by <code>//</code> .
Multi-line	Generate single or multiple-line comments delimited by <code>/*</code> and <code>*/</code> .
Single-line	Generate single-line comments preceded by <code>//</code> .

For example, the following command sets the comment style to single-line comments:

```
>> set_param('rtwdemo_counter','CommentStyle','Single-line')
```

Here is an example of code generated using the single-line comment style:

```
// Sum: '<Root>/Sum' incorporates:
//   Constant: '<Root>/INC'
//   UnitDelay: '<Root>/X'

rtb_sum_out = (uint8_T)(1U + rtwdemo_counter_DW.X);
```

Note: For C code generation, select `Single-line` only if your compiler supports it

Customize Generated Identifier Naming Rules

For GRT and RSim targets, the code generator automatically constructs identifiers for variables and functions in the generated code. For ERT targets, you can customize the naming of identifiers in the generated code by specifying parameters on the **Code Generation > Symbols** pane in the Configuration Parameters dialog box.

Goal	Specify
Set the maximum number of characters the code generator uses for function, <code>typedef</code> , and variable names (default 31)	An integer value for the “Maximum identifier length” parameter. For more information, see “Specify Identifier Length to Avoid Naming Collisions”. If you expect your model to generate lengthy identifiers (due to use of long signal or parameter names, for example), or you find that identifiers are mangled more than expected, increase the value of this parameter. For more information, see “Specify Identifier Length to Avoid Naming Collisions”.
Define a macro string that specifies certain substrings that are included within generated identifiers for: <ul style="list-style-type: none"> • Global variables • Global types • Field names of global types • Subsystem methods • Subsystem method arguments • Local temporary variables • Local block output variables • Constant macros • Shared utilities 	A macro string for the Identifier format control parameters. For more information, see “Identifier Format Control” on page 14-16. For exceptions and limitations, see “Exceptions to Identifier Formatting Conventions” on page 14-23 and “Identifier Format Control Parameters Limitations” on page 14-24.
Set the minimum number of characters that the code generator uses for the mangling string	An integer value for the “Minimum mangle length” parameter. For more information, see “Control Name Mangling in Generated Identifiers” on page 14-19.

Goal	Specify
Control whether shortened names are used for system-generated identifiers	<p data-bbox="730 302 1205 361">Shortened for the “System-generated identifiers” parameter. This setting:</p> <ul data-bbox="730 388 1311 614" style="list-style-type: none"><li data-bbox="730 388 1219 420">• Provides more space for user names.<li data-bbox="730 430 1299 489">• Provides a more predictable and consistent naming system that uses camel case.<li data-bbox="730 499 1258 531">• Does not include underscores or plurals<li data-bbox="730 541 1311 614">• Provides consistent abbreviations for both a type and a variable.
Control whether scalar inlined parameter values are expressed in the generated code as literal values or macros	<p data-bbox="730 631 1213 718">The value Literals or Macros for the “Generate scalar inlined parameter as” parameter.</p> <ul data-bbox="730 746 1326 944" style="list-style-type: none"><li data-bbox="730 746 1326 840">• Literals: If you select Inline parameters, parameters are expressed as numeric constants.<li data-bbox="730 850 1252 944">• Macros: Parameters are expressed as variables (with <code>#define</code> macros). This setting makes code more readable.

For more information on these parameters, see “Code Generation Pane: Symbols”.

Identifier Format Control

You can customize generated identifiers by specifying the **Identifier format control** parameters on the **Code Generation > Symbols** pane in the Configuration Parameters dialog box. For each parameter, you can enter a macro string that specifies whether, and in what order, certain substrings are included within generated identifiers. For example, you can specify that the root model name be inserted into each identifier using the **\$R** token.

The macro string can include:

- Valid tokens, which are listed in Identifier Format Tokens. You can use or omit tokens depending on what you want to include in the identifier name. The **Shared utilities** parameter requires you to specify the checksum string token, **\$C**. The other parameters require the mangling string token, **\$M**. For more information, see “Control Name Mangling in Generated Identifiers” on page 14-19. The mangling string token is subject to the use and ordering restrictions noted in Identifier Format Control Parameter Values.
- Valid C or C++ language identifier characters (**a-z**, **A-Z**, **_**, **0-9**).

The build process generates each identifier by expanding tokens and inserting the resultant strings into the identifier. The tokens are expanded in the order listed in Identifier Format Tokens. Character strings are inserted in the positions that you specify around tokens directly into the identifier. Contiguous token expansions are separated by the underscore (**_**) character.

Identifier Format Tokens

Token	Description
\$M	This token is required. If necessary, the code generator inserts a name mangling string to avoid naming collisions. The position of the \$M token in the Identifier format control parameter specification determines the position of the name mangling string in the generated identifier. For example, if you use the specification \$R\$N\$M , the name mangling string is appended (if required) to the end of the identifier. For more information, see “Control Name Mangling in Generated Identifiers” on page 14-19.
\$F	Insert method name (for example, _Update for update method). This token is available only for subsystem methods.

Token	Description
\$N	Insert name of object (block, signal or signal object, state, parameter, shared utility function or parameter object) for which identifier is being generated.
\$R	Insert root model name into identifier, replacing unsupported characters with the underscore (_) character. When you use referenced models, this token is required in addition to \$M (see “Avoid Identifier Name Collisions with Referenced Models” on page 14-21). Note: This token replaces the Prefix model name to global identifiers option in previous releases.
\$H	Insert tag indicating system hierarchy level. For root-level blocks, the tag is the string <code>root_</code> . For blocks at the subsystem level, the tag is of the form <code>sN_</code> . N is a unique system number assigned by the Simulink software. This token is available only for subsystem methods and field names of global types. Note: This token replaces the Include System Hierarchy Number in Identifiers option in previous releases.
\$A	Insert data type acronym (for example, <code>i32</code> for integers) to signal and work vector identifiers. This token is available for local block output variables, local temporary variables, and field names of global types. Note: This token replaces the Include data type acronym in identifier option in previous releases.
\$I	Insert <code>u</code> if the argument is an input or <code>y</code> if the argument is an output. (For example, <code>rtu_</code> for an input argument and <code>rty_</code> for an output argument). This token is available only for subsystem method arguments.
\$C	This token is required for Shared utilities . If the identifier exceeds the Maximum identifier length , the code generator inserts an 8-character checksum to avoid naming collisions. The position of the \$C token in the Identifier format control parameter specification determines the position of the checksum in the generated identifier. For example, if you use the specification <code>\$N\$C</code> , the checksum is appended to the end of the identifier. This token is available only for shared utilities.

Identifier Format Control Parameter Values lists the default macro string, the supported tokens, and the applicable restrictions for each **Identifier format control** parameter.

Identifier Format Control Parameter Values

Parameter	Default Value	Supported Tokens	Restrictions
Global variables	rt\$N\$M	\$R, \$N, \$M	\$F, \$H, \$A, and \$I are not allowed.
Global types	\$N\$R\$M_T	\$N, \$R, \$M	\$F, \$H, \$A, and \$I are not allowed.
Field name of global types	\$N\$M	\$N, \$M, \$H, \$A	\$R, \$F, and \$I are not allowed.
Subsystem methods	\$F\$N\$M	\$R, \$N, \$M, \$F, \$H	\$F and \$H are empty for Stateflow functions; \$A and \$I are not allowed.
Subsystem method arguments	rt\$I\$N\$M	\$N, \$M, \$I	\$R, \$F, \$H, and \$A are not allowed.
Local temporary variables	\$N\$M	\$N, \$M, \$R, \$A	\$F, \$H, and \$I are not allowed.
Local block output variables	rtb_ \$N\$M	\$N, \$M, \$A	\$R, \$F, \$H, and \$I are not allowed.
Constant macros	\$R\$N\$M	\$R, \$N, \$M	\$F, \$H, \$A, and \$I are not allowed.
Shared utilities	\$N\$C	\$N, \$C	\$C is required. \$M, \$R, \$F, \$H, \$A , and \$I are not allowed.

Non-ERT-based targets (such as the GRT target) implicitly use a default `RN$M` specification. This default specification consists of the root model name, followed by the name of the generating object (signal, parameter, state, and so on), followed by a name mangling string.

For limitations that apply to **Identifier format control** parameters, see “Exceptions to Identifier Formatting Conventions” on page 14-23 and “Identifier Format Control Parameters Limitations” on page 14-24.

Control Name Mangling in Generated Identifiers

The position of the **\$M** token in the **Identifier format control** parameter specification determines the position of the name mangling string in the generated identifiers. For example, if you use the specification **\$R\$N\$M**, the name mangling string is appended (if required) to the end of the identifier. For more information, see “Identifier Format Control” on page 14-16.

Name Mangling String Per Object

Object Type	Source of Mangling String
Block diagram	Name of block diagram
Simulink block	“Simulink Identifier” (SID)
Simulink parameter	Full name of parameter owner (model or block) and parameter name
Simulink signal	Signal name, full name of source block, and port number
Stateflow objects	Complete path to Stateflow block and Stateflow computed name (unique within chart)

The length of the name mangling string is specified by the **Minimum mangle length** parameter. The default value is 1, but this automatically increases during code generation as a function of the number of collisions. To minimize disturbance to the generated code during development, specify a larger **Minimum mangle length**. A **Minimum mangle length** of 4 is a conservative value. A value of 4 allows for over 1.5 million collisions for a particular identifier before the mangle length is increased.

Minimize Name Mangling

The length of generated identifiers is limited by the **Maximum identifier length** parameter. When a name collision exists, the **\$M** token is expanded to the minimum number of characters required to avoid the collision. Other tokens and character strings are expanded in the order listed in Identifier Format Tokens. If the **Maximum identifier length** is not large enough to accommodate full expansions of the other tokens, partial expansions are used. To avoid partial expansions, it is good practice to:

- Avoid name collisions. One way to avoid name collisions is to not use default block names (for example, **Gain1**, **Gain2**...) when there are many blocks of the same type in the model.

- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers that you expect to generate.
- Set the **Maximum identifier length** parameter to reserve at least three characters for the name mangling string. The length of the name mangling string increases as the number of name collisions increases.

If changes to the model create more or fewer collisions, an existing name mangling string increases or decreases in length. If the length of the name mangling string increases, additional characters are appended to the existing string. For example, the mangling string 'xyz' can change to 'xyzQ'. For fewer collisions, the name mangling string 'xyz' changes to 'xy'.

Avoid Identifier Name Collisions with Referenced Models

Within a model that uses referenced models, collisions between the names of the models are not allowed. When generating code from a model that uses model referencing:

- You must include the **\$R** token in the **Identifier format control** parameter specifications (in addition to the **\$M** token).
- The **Maximum identifier length** must be large enough to accommodate full expansions of the **\$R** and **\$M** tokens. If **Maximum identifier length** is too small, a code generation error occurs.

When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model, the identifier from the referenced model is preserved. Name mangling is performed on the identifier from the higher-level model.

If your model contains two referenced models with the same input or output port names, and one of the referenced models contains an atomic subsystem with “Function packaging” set to `Nonreusable function`, a name conflict can occur and the build process produces an error.

Maintain Traceability for Generated Identifiers

To verify your model, you can trace back and forth between generated identifiers and corresponding entities within the model. To maintain traceability, it is important that incremental revisions to a model have minimal impact on the identifier names that appear in generated code. There are two ways to minimally impact the identifier names:

- Choose unique names for Simulink objects (blocks, signals, states, and so on) as much as possible.
- Use name mangling when conflicts cannot be avoided.

The position of the name mangling string is specified by the placement of the **\$M** token in the **Identifier format control** parameters. Mangle characters consist of alphanumeric characters that are unique to each object. For more information, see “Control Name Mangling in Generated Identifiers” on page 14-19.

Exceptions to Identifier Formatting Conventions

There are some exceptions to the identifier formatting conventions described in “Identifier Format Control” on page 14-16.

- Type name generation: name mangling conventions do not apply to type names (that is, `typedef` statements) generated for global data types. If the `$R` token is included in the **Identifier format control** parameter specification, the model name is included in the `typedef`. When generating type definitions, the **Maximum identifier length** parameter is not respected.
- Non-`Auto` storage classes: the **Identifier format control** parameters specification does not affect objects (such as signals and parameters) that have a storage class other than `Auto` (such as `ImportedExtern` or `ExportedGlobal`).
- For shared utilities, code generation inserts the checksum specified by `$C` to prevent name collisions in the following situations:
 - `$C` is specified without `$N`.
 - The length of `$N` plus the length of the text that you specify exceeds the **Maximum identifier length**. Code generation truncates `$N` and inserts an 8-character checksum where you specified `$C` in the formatting string.

Identifier Format Control Parameters Limitations

The following limitations apply to the **Identifier format control** parameters:

- The following autogenerated identifiers currently do not fully comply with the setting of the **Maximum identifier length** parameter on the **Code Generation > Symbols** pane of the Configuration Parameters dialog box.
 - Model methods
 - The applicable format string is `$$R$F`, and the longest `$F` is `_derivatives`, which is 12 characters long. The model name can be up to 19 characters without exceeding the default **Maximum identifier length** of 31.
 - Local functions generated by S-functions or by add-on products such as DSP System Toolbox™ that rely on S-functions
 - Local variables generated by S-functions or by add-on products such as DSP System Toolbox that rely on S-functions
 - DW identifiers generated by S-functions in referenced models
 - Fixed-point shared utility macros or shared utility functions
 - Simulink `rtm` macros
 - Most are within the default **Maximum identifier length** of 31, but some exceed the limit. Examples are `RTMSpecAccsGetStopRequestedValStoredAsPtr`, `RTMSpecAccsGetErrorStatusPointer`, and `RTMSpecAccsGetErrorStatusPointerPointer`.
 - Define protection guard macros
 - Header file guards, such as `_RTW_HEADER_$(filename)_h_`, which can exceed the default **Maximum identifier length** of 31 given a filename such as `$_private.h`.
 - Include file guards, such as `$_R_COMMON_INCLUDES_`.
 - `typedef` guards, such as `_CSCI_$_R_CHARTSTRUCT_`.
- In some situations, the following identifiers potentially can conflict with others.
 - Model methods
 - Reentrant model function arguments

- Local functions generated by S-functions or by add-on products such as DSP System Toolbox that rely on S-functions
- Local variables generated by S-functions or by add-on products such as DSP System Toolbox that rely on S-functions
- Fixed-point shared utility macros or shared utility functions
- Include header guard macros
- The following external identifiers that are unknown to the Simulink software might conflict with autogenerated identifiers.
 - Identifiers defined in custom code
 - Identifiers defined in custom header files
 - Identifiers introduced through a non-ANSI C standard library
 - Identifiers defined by custom TLC code
- Identifiers generated for simulation targets might exceed the **Maximum identifier length**. Simulation targets include the model reference simulation target, the accelerated simulation target, the RSim target, and the S-function target.
- Identifiers generated using a model name and bus object data type name, which are both long names, might exceed the **Maximum identifier length**. For example, a ground value variable name is generated as `<model_name>_rtZ<bus_name>`. If the *model_name* and *bus_name* are close to the maximum identifier length, the name exceeds the maximum identifier length.

Specify Simulink Data Object Naming Rules

To Define Rules that Change the Names of...	Specify a Naming Rule with the ...
Signals	Signal naming parameter
Parameters	Parameter naming parameter
Parameters that have a storage class of Define	#define naming parameter

For more information on these parameters, see “Specify Data Object Naming Rules”.

Control Code Style

In this section...

“Control Parentheses in Generated Code” on page 14-27

“Control Indentation Style in Generated Code” on page 14-28

“Control Cast Expressions in Generated Code” on page 14-29

You can change the code style, cast expressions, and indentation of your generated code to conform to certain coding standards. Modify style options by setting parameters on the **Code Generation > Code Style** pane.

In the generated code, you can control the following style aspects:

- Level of parenthesization, see “Control Parentheses in Generated Code” on page 14-27
- Order of operands in expressions, see “Preserve operand order in expression”
- Empty primary condition expressions in `if` statements, see “Preserve condition expression in if statement”
- Whether to generate code for `if-elseif-else` decision logic as `switch-case` statements, see “Convert if-elseif-else patterns to switch-case statements”
- Whether to include the `extern` keyword in function declarations, see “Preserve extern keyword in function declarations”
- Whether to generate default cases for `switch-case` statements in the code for Stateflow charts, see “Suppress generation of default cases for Stateflow switch statements if unreachable”
- Cast expressions, see “Control Cast Expressions in Generated Code” on page 14-29
- Indentation style, see “Control Indentation Style in Generated Code” on page 14-28

Control Parentheses in Generated Code

C code contains some syntactically required parentheses, and can contain additional parentheses that change semantics by overriding default operator precedence. C code can also contain optional parentheses that have no functional significance, but only increase the readability of the code. Optional C parentheses vary between two stylistic extremes:

- Include the minimum parentheses required by C syntax and precedence overrides so that C precedence rules specify all semantics unless overridden by parentheses.
- Include the maximum parentheses that can exist without duplication so that C precedence rules become irrelevant. Parentheses alone completely specify all semantics.

Understanding code with minimum parentheses can require applying nonobvious precedence rules. Maximum parentheses can hinder code reading by belaboring obvious precedence rules. Various parenthesization standards exist that specify one or the other extreme, or define an intermediate style useful to people who read code.

The following example model shows the three levels of parentheses control that you can set before generating code: `rtwdemo_parentheses`. For more information on this parameter, see “Parentheses level”.

Control Indentation Style in Generated Code

For code indentation, you can set the following parameters:

- “Indent style” controls the placement of braces in generated code.
- “Indent size” controls the number of characters per indent level in generated code (2–8 characters).

You can set **Indent style** to **K&R** or **Allman** style.

K&R

K&R stands for Kernighan and Ritchie. Each function has the opening and closing brace on its own line at the same level of indentation as the function header. Code within the function is indented according to the **Indent size**.

For blocks within a function, opening braces are on the same line as the control statement. Closing braces are on a new line at the same level of indentation as the control statement. Code within the block is indented according to the **Indent size**.

For example, here is generated code with the **Indent style** set to **K&R** with an **Indent size** of 2:

```
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

Each function has the opening and closing brace on its own line at the same level of indentation as the function header. Code within the function is indented according to the **Indent size**.

For blocks within a function, opening and closing braces for control statements are on a new line at the same level of indentation as the control statement. This is the key difference between K&R and Allman styles. Code within the block is indented according to the **Indent size**.

For example, here is generated code with the **Indent style** set to Allman with an **Indent size** of 4:

```
void rt_OneStep(void)
{
    static boolean_T OverrunFlag = 0;
    if (OverrunFlag)
    {
        rtmSetErrorStatus(rtwdemo_counter_M, "Overrun");
        return;
    }

    OverrunFlag = TRUE;
    rtwdemo_counter_step();
    OverrunFlag = FALSE;
}
```

Control Cast Expressions in Generated Code

You can choose how the code generator specifies data type casts in the generated code. In the Configuration Parameters dialog box, select **Code Generation > Code Style**.

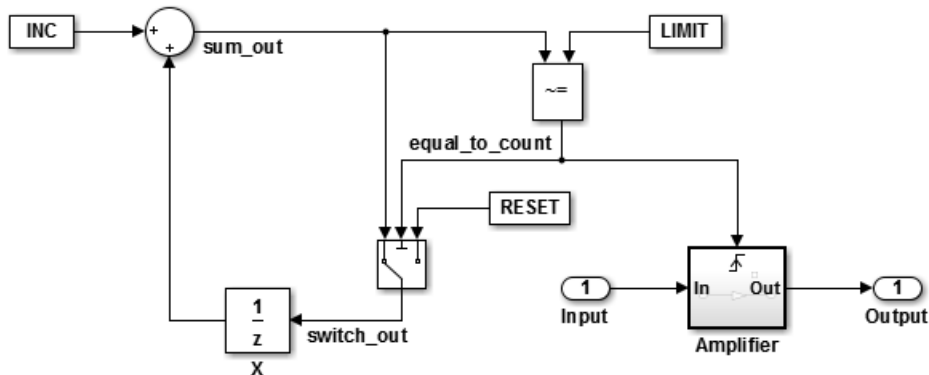
From the **Casting Modes** drop-down list, three parameter options control how the code generator casts data types.

- **Nominal** instructs the code generator to generate code that has minimal data type casting and is compatible with previous code generator versions. When you do not have special data type information requirements, choose **Nominal**.
- **Standards Compliant** instructs the code generator to cast data types to conform to MISRA standards when it generates code. The MISRA data type casting eliminates common MISRA standard violations, including address arithmetic and assignment. It reduces 10.1, 10.2, 10.3, and 10.4 violations.

For more information, see “MISRA C Guidelines”.

- **Explicit** instructs the code generator to cast data type values explicitly when it generates code. You can see how a value is stored, which tells you how much memory space the code uses for the variable. The data type informs you how much precision is possible in calculations involving the variable.

Open the example model `rtwdemo_rtwecintro`.



Enable Nominal Casting Mode and Generate Code

When you choose **Nominal** casting mode, the code generator does not create data type casts for variables in the generated code.

- 1 On the **Code Generation > Code Style** pane, from the **Casting Modes** drop-down list, select **Nominal**.
- 2 On the **Code Generation > Report** pane, select **Create code generation report**.

- 3 On the **Code Generation** pane, select **Generate code only**.
- 4 Click **Apply**.
- 5 Click **Generate Code**.
- 6 In the Code Generation report left pane, click `rtwdemo_rtwecintro.c` to see the code.

```

/* Model step function */
void rtwdemo_rtwecintro_step(void)
{
    boolean_T rtb_equal_to_count;

    /* Sum: 'XRootX/Sum' incorporates:
     * Constant: 'XRootX/INC'
     * UnitDelay: 'XRootX/X'
     */
    rtDWork.X++;

    /* RelationalOperator: 'XRootX/RelOpt' incorporates:
     * Constant: 'XRootX/LIMIT'
     */
    rtb_equal_to_count = (rtDWork.X != 16);

    /* Outputs for Triggered SubSystem: 'XRootX/Amplifier' incorporates:
     * TriggerPort: 'XS1X/Trigger'
     */
    if (rtb_equal_to_count && (rtPrevZCSigState.Amplifier_Trig_ZCE != POS_ZCSIG))
    {
        /* Outport: 'XRootX/Output' incorporates:
         * Gain: 'XS1X/Gain'
         * Inport: 'XRootX/Input'
         */
        rtY.Output = rtU.Input << 1;
    }

    rtPrevZCSigState.Amplifier_Trig_ZCE = (uint8_T)(rtb_equal_to_count ? (int32_T)
        POS_ZCSIG : (int32_T)ZERO_ZCSIG);

    /* End of Outputs for SubSystem: 'XRootX/Amplifier' */

    /* Switch: 'XRootX/Switch' */
    if (!rtb_equal_to_count) {
        /* Update for UnitDelay: 'XRootX/X' incorporates:
         * Constant: 'XRootX/RESET'

```

```

    */
    rtDWork.X = 0U;
}

/* End of Switch: 'XRootX/Switch' */
}

```

Enable Standards Compliant Casting Mode and Generate Code

When you choose **Standards Compliant** casting mode, the code generator creates MISRA standards compliant data type casts for variables in the generated code.

- 1 On the **Code Style** pane, from the **Casting Modes** drop-down list, select **Standards Compliant**.
- 2 On the **Code Generation** pane, click **Apply**.
- 3 Click **Generate Code**.
- 4 In the Code Generation report left pane, click `rtwdemo_rtweccintro.c` to see the code.

```

void rtwdemo_rtweccintro_step(void)
{
    boolean_T rtb_equal_to_count;

    /* Sum: '<Root>/Sum' incorporates:
     * Constant: '<Root>/INC'
     * UnitDelay: '<Root>/X'
     */
    rtDWork.X++;

    /* RelationalOperator: '<Root>/RelOpt' incorporates:
     * Constant: '<Root>/LIMIT'
     */
    rtb_equal_to_count = (boolean_T)(int32_T)((int32_T)rtDWork.X != (int32_T)16);

    /* Outputs for Triggered SubSystem: '<Root>/Amplifier' incorporates:
     * TriggerPort: '<S1>/Trigger'
     */
    if (((int32_T)rtb_equal_to_count) && (rtPrevZCSigState.Amplifier_Trig_ZCE !=
        POS_ZCSIG)) {
        /* Output: '<Root>/Output' incorporates:
         * Gain: '<S1>/Gain'
         * Inport: '<Root>/Input'
         */
    }
}

```

```

    rtY.Output = (int32_T)(uint32_T)((uint32_T)rtU.Input << (uint32_T)(int8_T)1);
}

rtPrevZCSigState.Amplifier_Trig_ZCE = (uint8_T)(int32_T)(rtb_equal_to_count ?
    (int32_T)(uint8_T)POS_ZCSIG : (int32_T)(uint8_T)ZERO_ZCSIG);

/* End of Outputs for SubSystem: '<Root>/Amplifier' */

/* Switch: '<Root>/Switch' */
if (!rtb_equal_to_count) {
    /* Update for UnitDelay: '<Root>/X' incorporates:
     * Constant: '<Root>/RESET'
     */
    rtDWork.X = 0U;
}

/* End of Switch: '<Root>/Switch' */
}

```

Enable Explicit Casting Mode and Generate Code

When you choose **Explicit** casting mode, the code generator creates explicit data type casts for variables in the generated code.

- 1** On the **Code Style** pane, from the **Casting Modes** drop-down list, select **Explicit**.
- 2** On the **Code Generation** pane, click **Apply**.
- 3** Click **Generate Code**.
- 4** In the Code Generation report left pane, click `rtwdemo_rtweccintro.c` to see the code.

```

/* Model step function */
void rtwdemo_rtweccintro_step(void)
{
    boolean_T rtb_equal_to_count;

    /* Sum: '<Root>/Sum' incorporates:
     * Constant: '<Root>/INC'
     * UnitDelay: '<Root>/X'
     */
    rtDWork.X = (uint8_T)(1U + (uint32_T)(int32_T)rtDWork.X);

    /* RelationalOperator: '<Root>/RelOpt' incorporates:
     * Constant: '<Root>/LIMIT'

```

```
*/
rtb_equal_to_count = (boolean_T)((int32_T)rtDWork.X != 16);

/* Outputs for Triggered SubSystem: '<Root>/Amplifier' incorporates:
 * TriggerPort: '<S1>/Trigger'
 */
if (((int32_T)rtb_equal_to_count) && ((int32_T)((int32_T)
    rtPrevZCSigState.Amplifier_Trig_ZCE != (int32_T)POS_ZCSIG))) {
    /* Output: '<Root>/Output' incorporates:
     * Gain: '<S1>/Gain'
     * Inport: '<Root>/Input'
     */
    rtY.Output = rtU.Input << 1;
}

rtPrevZCSigState.Amplifier_Trig_ZCE = (uint8_T)(rtb_equal_to_count ? (int32_T)
    POS_ZCSIG : (int32_T)ZERO_ZCSIG);

/* End of Outputs for SubSystem: '<Root>/Amplifier' */

/* Switch: '<Root>/Switch' */
if (!(int32_T)rtb_equal_to_count) {
    /* Update for UnitDelay: '<Root>/X' incorporates:
     * Constant: '<Root>/RESET'
     */
    rtDWork.X = 0U;
}

/* End of Switch: '<Root>/Switch' */
}
```

More About

- “Code Generation Pane: Code Style”

Customize Code Organization and Format

In this section...

“Custom File Processing Components” on page 14-35

“Custom File Processing Configuration” on page 14-36

Custom file processing (CFP) tools allow you to customize the organization and formatting of your generated code. With these tools, you can:

- Generate a source (.c or .cpp) or header (.h) file. Using a *custom file processing template* (CFP template), you can control how code emits to the standard generated model files (for example, *model.c* or *.cpp*, *model.h*) or generate files that are independent of model code.
- Organize generated code into sections (such as includes, **typedefs**, functions, and more). Your CFP template can emit code (for example, functions), directives (such as **#define** or **#include** statements), or comments into each section.
- Generate custom *file banners* (comment sections) at the start and end of generated code files and custom *function banners* that precede functions in the generated code.
- Generate code to call model functions, such as *model_initialize*, *model_step*, and so on.
- Generate code to read and write model inputs and outputs.
- Generate a main program module.
- Obtain information about the model and the generated files from the model.

Custom File Processing Components

The custom file processing features are based on the following interrelated components:

- *Code generation template* (CGT) files: a CGT file defines the top-level organization and formatting of generated code. See “Code Generation Template (CGT) Files” on page 14-38.
- The *code template API*: a high-level Target Language Compiler (TLC) API that provides functions with which you can organize code into named sections and subsections of generated source and header files. The code template API also provides utilities that return information about generated files, generate standard model calls, and perform other functions. See “Code Template API Summary” on page 14-58.

- *Custom file processing (CFP) templates*: a CFP template is a TLC file that manages the process of custom code generation. A CFP template assembles code to be generated into buffers. A CFP template also calls the code template API to emit the buffered code into specified sections of generated source and header files. A CFP template interacts with a CGT file, which defines the ordering of major sections of the generated code. See “Custom File Processing (CFP) Templates” on page 14-42.

To use CFP templates, you must understand TLC programming, for more information, see “Target Language Compiler”.

Custom File Processing Configuration

Customize generated code by specifying code and data templates on the **Code Generation > Templates** pane:

Goal	Action
Specify a template that defines the top-level organization and formatting of generated source code (.c or .cpp) files	Enter a code generation template (CGT) file for the Source file (*.c) template parameter.
Specify a template that defines the top-level organization and formatting of generated header (.h) files	Enter a CGT file for the Header file (*.h) template parameter. This template file can be the same template file that you specify for Source file (.c) template . If you use the same template file, source and header files contain identical banners. The default template is <i>matlabroot</i> /toolbox/rtw/targets/ecoder/ert_code_template.cgt.
Specify a template that organizes generated code into sections (such as includes, typedefs, functions, and more)	Enter a custom file processing (CFP) template file for the “File customization template” parameter. A CFP template can emit code, directives, or comments into each section. For more information, see “Custom File Processing (CFP) Templates” on page 14-42.
Generate a model-specific example main program module	Select Generate an example main program . For more information, see “Generate a Standalone Program”.

Note: Place the template files that you specify on the MATLAB path.

Specify Templates For Code Generation

To use custom file processing features, create CGT files and CFP templates. These files are based on default templates provided by the code generation software. Once you have created your templates, you must integrate them into the code generation process.

Select and edit CGT files and CFP templates, and specify their use in the code generation process in the **Code Generation > Templates** pane of a model configuration set. The following figure shows options configured for their defaults.

The options related to custom file processing are:

- The **Source file (.c) template** field in the **Code templates** and **Data templates** sections. This field specifies the name of a CGT file to use when generating source (.c or .cpp) files. You must place this file on the MATLAB path.
- The **Header file (.h) template** field in the **Code templates** and **Data templates** sections. This field specifies the name of a CGT file to use when generating header (.h) files. You must place this file on the MATLAB path.

By default, the template for both source and header files is matlabroot/toolbox/rtw/targets/ecoder/ert_code_template.cgt.

- The **File customization template** edit field in the **Custom templates** section. This field specifies the name of a CFP template file to use when generating code files. You must place this file on the MATLAB path. The default CFP template is matlabroot/toolbox/rtw/targets/ecoder/example_file_process.tlc.

In each of these fields, click **Browse** to navigate to and select an existing CFP template or CGT file. Click **Edit** to open the specified file into the MATLAB editor where you can customize it.

Code Generation Template (CGT) Files

Code Generation Template (CGT) files define the top-level organization and formatting of generated source code and header files. CGT files have the following applications:

- Generation of custom banners (comments sections) in code files. See “Generate Custom File and Function Banners” on page 14-61.
- Generation of custom code using a CFP template requires a CGT file. To use CFP templates, you must understand the CGT file structure. In many cases, however, you can use the default CGT file without modifying it.

Default CGT file

The code generation software provides a default CGT file, `matlabroot/toolbox/rtw/targets/ecoder/ert_code_template.cgt`. Base your custom CGT files on the default file.

CGT File Structure

A CGT file consists of one required section and four optional sections:

Code Insertion Section

(Required) This section contains tokens that define an ordered partitioning of the generated code into a number of sections (such as `Includes` and `Defines` sections). Tokens have the form of:

```
%<SectionName>
```

For example,

```
%<Includes>
```

The code generation software defines a minimal set of required tokens. These tokens generate C or C++ source or header code. They are *built-in* tokens (see “Built-In Tokens and Sections” on page 14-39). You can also define custom tokens and custom sections.

Each token functions as a placeholder for a corresponding section of generated code. The ordering of the tokens defines the order in which the corresponding sections appear in the generated code. If you do not include a token, then the corresponding section is not generated. To generate code into a given section, explicitly call the code template API from a CFP template, as described in “Custom File Processing (CFP) Templates” on page 14-42.

The CGT tokens define the high-level organization of generated code. Using the code template API, you can partition each code section into named subsections, as described in “Subsections” on page 14-40.

In the code insertion section, you can also insert C or C++ comments between tokens. Such comments emit directly into the generated code.

File Banner Section

(Optional) This section contains comments and tokens you use in generating a custom file banner.

Function Banner Section

(Optional) This section contains comments and tokens for use in generating a custom function banner.

Shared Utility Function Banner Section

(Optional) This section contains comments and tokens for use in generating a custom shared utility function banner.

File Trailer Section

(Optional) This section contains comments for use in generating a custom trailer banner.

For more information on these sections, see “Generate Custom File and Function Banners” on page 14-61.

Built-In Tokens and Sections

The following code extract shows the required code insertion section of the default CGT file with the required built-in tokens.

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Code insertion section (required)
%%   These are required tokens. You can insert comments and other tokens in
%%   between them, but do not change their order or remove them.
%%
%<Includes>
%<Defines>
%<Types>
%<Enums>
%<Definitions>
%<Declarations>
%<Functions>

```

Note the following requirements for customizing a CGT file:

- Do not remove required built-in tokens.
- Built-in tokens must appear in the order shown because each successive section has dependencies on previous sections.
- Only one token per line.
- Do not repeat tokens.
- You can add custom tokens and comments to the code insertion section as long as you do not violate the previous requirements.

Note: If you modify a CGT file and then rebuild your model, the code generation process does not force a top model build. To regenerate the code, see “Force Regeneration of Top Model Code”.

The following table summarizes the built-in tokens and corresponding section names, and describes the code sections.

Built-In CGT Tokens and Corresponding Code Sections

Token and Section Name	Description
Includes	<code>#include</code> directives section
Defines	<code>#define</code> directives section
Types	<code>typedef</code> section. <code>typedefs</code> can depend on a previously defined type
Enums	Enumerated types section
Definitions	Data definitions (for example, <code>double x = 3.0;</code>)
Declarations	Data declarations (for example, <code>extern double x;</code>)
Functions	C or C++ functions

Subsections

You can define one or more named subsections for any section. Some of the built-in sections have predefined subsections summarized in table “Subsections Defined for Built-In Sections”.

Note: Sections and subsections emit to the source or header file in the order listed in the CGT file.

Using the custom section feature, you can define additional sections. See “Generate a Custom Section”.

Subsections Defined for Built-In Sections

Section	Subsections	Subsection Description
Includes	N/A	
Defines	N/A	
Types	IntrinsicTypes	Intrinsic typedef section. Intrinsic types depend only on intrinsic C or C++ types.
Types	PrimitiveTypedefs	Primitive typedef section. Primitive typedefs depend only on intrinsic C or C++ types and on typedefs previously defined in the IntrinsicTypes section.
Types	UserTop	You can place any type of code in this section, including code that has dependencies on the previous sections.
Types	Typedefs	typedef section. Typedefs can depend on previously defined types
Enums	N/A	
Definitions	N/A	
Declarations	N/A	
Functions		C or C++ functions
Functions	CompilerErrors	#error directives
Functions	CompilerWarnings	#warning directives
Functions	Documentation	Documentation (comment) section
Functions	UserBottom	You can place any code in this section.

Custom File Processing (CFP) Templates

The files provided to support custom file processing are:

- `matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc`: A TLC function library that implements the code template API. `codetemplatelib.tlc` also provides the comprehensive documentation of the API in the comments headers preceding each function.
- `matlabroot/toolbox/rtw/targets/ecoder/example_file_process.tlc`: An example custom file processing (CFP) template, which you should use as the starting point for creating your own CFP templates. Guidelines and examples for creating a CFP template are provided in “Generate Source and Header Files with a Custom File Processing (CFP) Template” on page 14-46.
- TLC files supporting generation of single-rate and multirate main program modules (see “Customizing Main Program Module Generation” on page 14-50).

Once you have created a CFP template, you must integrate it into the code generation process, using the **File customization template** edit field. See “Specify Templates For Code Generation” on page 14-37.

Custom File Processing (CFP) Template Structure

A custom file processing (CFP) template imposes a simple structure on the code generation process. The template, a code generation template (CGT) file, partitions the code generated for each file into a number of sections. These sections are summarized in Built-In CGT Tokens and Corresponding Code Sections and Subsections Defined for Built-In Sections.

Code for each section is assembled in buffers and then emitted, in the order listed, to the file being generated.

To generate a file section, your CFP template must first assemble the code to be generated into a buffer. Then, to emit the section, your template calls the TLC function

```
LibSetSourceFileSection(fileH, section, tmpBuf)
```

where

- `fileH` is a file reference to a file being generated.
- `section` is the code section or subsection to which code is to be emitted. `section` must be one of the section or subsection names listed in Subsections Defined for Built-In Sections.

Determine the `section` argument as follows:

- If Subsections Defined for Built-In Sections does not define subsections for a given section, use the section name as the `section` argument.
- If Subsections Defined for Built-In Sections defines one or more subsections for a given section, you can use either the section name or a subsection name as the `section` argument.
- If you have defined a custom token denoting a custom section, do not call `LibSetSourceFileSection`. Special API calls are provided for custom sections (see “Generate a Custom Section” on page 14-51).
- `tmpBuf` is the buffer containing the code to be emitted.

There is no requirement to generate all of the available sections. Your template need only generate the sections you require in a particular file.

Note that legality or syntax checking is not performed on the custom code within each section.

See “Generate Source and Header Files with a Custom File Processing (CFP) Template” on page 14-46, for typical usage examples.

Change the Organization of a Generated File

The files created during code generation are organized according to the general code generation template. This template has the filename `ert_code_template.cgt`, and is specified by default in **Code Generation > Templates** pane of the Configuration Parameters dialog box.

The following fragment shows the `rtwdemo_mpf.c` file header that is generated using this default template:

```
/*
 * File: rtwdemo_mpf.c
 *
 * Code generated for Simulink model 'rtwdemo_mpf'.
 *
 * Model version           : 1.88
 * Simulink Coder version  : 8.0 (R2011a) 26-Aug-2010
 * TLC version             : 7.6 (Sep 3 2010)
 * C/C++ source code generated on : Thu Sep 09 10:10:14 2010
 *
 * Target selection: ert.tlc
 * Embedded hardware selection: Generic->32-bit Embedded Processor
 * Code generation objectives: Unspecified
 * Validation result: Not run
 */
```

You can change the organization of generated files using code templates and data templates. Code templates organize the files that contain functions, primarily. Data templates organize the files that contain identifiers. In this procedure, you organize the generated files, using the supplied code and data templates:

- 1 Display the active **Templates** configuration parameters.
- 2 In the **Code templates** section of the **Templates** pane, type `code_c_template.cgt` into the **Source file (*.c) templates** text box.
- 3 Type `code_h_template.cgt` into the **Header file (*.h) templates** text box.
- 4 In the **Data templates** section, type `data_c_template.cgt` into the **Source file (*.c) templates** text box.
- 5 Type `data_h_template.cgt` into the **Header file (*.h) templates** text box, and click **Apply**.
- 6 Click **Generate Code**. Now the files are organized using the templates you specified. For example, the `rtwdemo_mpf.c` file header now is organized like this:

```
/**
*****
```

```
** FILE INFORMATION:
** Filename:          rtwdemo_mpf.c
** File Creation Date: 09-Sep-2010
**
** ABSTRACT:
**
**
** NOTES:
**
**
** MODEL INFORMATION:
** Model Name:        rtwdemo_mpf
** Model Description: Data packaging examples
** Model Version:     1.89
** Model Author:      The MathWorks Inc. - Mon Mar 01 11:23:00 2004
**
** MODIFICATION HISTORY:
** Model at Code Generation: ssulliva - Thu Sep 09 10:19:35 2010
**
** Last Saved Modification: ssulliva - Thu Sep 09 10:19:13 2010
**
**
*****
** /
```

Generate Source and Header Files with a Custom File Processing (CFP) Template

In this section...

“Generate Code with a CFP Template” on page 14-46

“Analysis of the Example CFP Template and Generated Code” on page 14-48

“Generate a Custom Section” on page 14-51

“Custom Tokens” on page 14-53

This example shows you the process of generating a simple source (.c or .cpp) and header (.h) file using the example CFP template. Then, it examines the template and the code generated by the template.

The example CFP template, `matlabroot/toolbox/rtw/targets/ecoder/example_file_process.tlc`, demonstrates some of the capabilities of the code template API, including

- Generation of simple source (.c or .cpp) and header (.h) files
- Use of buffers to generate file sections for includes, functions, and so on
- Generation of includes, defines, into the standard generated files (for example, `model.h`)
- Generation of a main program module

Generate Code with a CFP Template

This section sets up a CFP template and configures a model to use the template in code generation. The template generates (in addition to the standard model files) a source file (`timestwo.c` or `.cpp`) and a header file (`timestwo.h`).

Follow the steps below to become acquainted with the use of CFP templates:

- 1 Copy the example CFP template, `matlabroot/toolbox/rtw/targets/ecoder/example_file_process.tlc`, to a folder outside of the MATLAB folder structure (that is, not under `matlabroot`). If the folder is not on the MATLAB path or the TLC path, then add it to the MATLAB path. It is good practice to locate the CFP template in the same folder as your system target file, which is on the TLC path.
- 2 Rename the copied `example_file_process.tlc` to `test_example_file_process.tlc`.

3 Open `test_example_file_process.tlc` into the MATLAB editor.

4 Uncomment the following line:

```
%% %assign ERTCustomFileTest = TLC_TRUE
```

It now reads:

```
%assign ERTCustomFileTest = TLC_TRUE
```

If `ERTCustomFileTest` is not assigned as shown, the CFP template is ignored in code generation.

5 Save your changes to the file. Keep `test_example_file_process.tlc` open, so you can refer to it later.

6 Open the `rtwdemo_udt` model.

7 Open the Simulink Model Explorer. Select the active configuration set of the model, and open the **Code Generation** pane of the active configuration set.

8 Click the **Templates** tab.

9 Specify **File customization template** as `test_example_file_process.tlc`. This is the file you previously edited and is now the specified CFP template for your model.

10 Select the **Generate code only** option.

11 Click **Apply**.

12 Click **Generate Code**. During code generation, notice the following message on the MATLAB command window:

```
Warning: Overriding example ert_main.c!
```

This message is displayed because `test_example_file_process.tlc` generates the main program module, overriding the default action of the ERT target. This is explained in greater detail below.

13 The `rtwdemo_udt` model is configured to generate an HTML code generation report. After code generation completes, view the report. Notice that the **Generated Files** list contains the files `timestwo.c`, `timestwo.h`, and `ert_main.c`. These files were generated by the CFP template. The next section examines the template to learn how this was done.

14 Keep the model, the code generation report, and the `test_example_file_process.tlc` file open so you can refer to them in the next section.

Analysis of the Example CFP Template and Generated Code

This section examines excerpts from `test_example_file_process.tlc` and some of the code it generates. Refer to the comments in `matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc` while reading the following discussion.

Generating Code Files

Source (`.c` or `.cpp`) and header (`.h`) files are created by calling `LibCreateSourceFile`, as in the following excerpts:

```
%assign cFile = LibCreateSourceFile("Source", "Custom", "timestwo")
...
%assign hFile = LibCreateSourceFile("Header", "Custom", "timestwo")
```

Subsequent code refers to the files by the file reference returned from `LibCreateSourceFile`.

File Sections and Buffers

The code template API lets you partition the code generated to each file into sections, tagged as `Definitions`, `Includes`, `Functions`, `Banner`, and so on. You can append code to each section as many times as required. This technique gives you a great deal of flexibility in the formatting of your custom code files.

`Subsections Defined for Built-In Sections` describes the available file sections and their order in the generated file.

For each section of a generated file, use `%openfile` and `%closefile` to store the text for that section in temporary buffers. Then, to write (append) the buffer contents to a file section, call `LibSetSourceFileSection`, passing in the desired section tag and file reference. For example, the following code uses two buffers (`tmwtypesBuf` and `tmpBuf`) to generate two sections (tagged `"Includes"` and `"Functions"`) of the source file `timestwo.c` or `.cpp` (referenced as `cFile`):

```
%openfile tmwtypesBuf

#include "tmwtypes.h"

%closefile tmwtypesBuf

%<LibSetSourceFileSection(cFile, "Includes", tmwtypesBuf)>
```

```

%openfile tmpBuf

/* Times two function */
real_T timestwofcn(real_T input) {
    return (input * 2.0);
}

%closefile tmpBuf

%<LibSetSourceFileSection(cFile, "Functions", tmpBuf)>

```

These two sections generate the entire `timestwo.c` or `.cpp` file:

```

#include "tmwtypes.h"

/* Times two function */
FLOAT64 timestwofcn(FLOAT64 input)
{
    return (input * 2.0);
}

```

Adding Code to Standard Generated Files

The `timestwo.c` or `.cpp` file generated in the previous example was independent of the standard code files generated from a model (for example, `model.c` or `.cpp`, `model.h`, and so on). You can use similar techniques to generate custom code within the model files. The code template API includes functions to obtain the names of the standard models files and other model-related information. The following excerpt calls `LibGetMdlPubHdrBaseName` to obtain the name for the `model.h` file. It then obtains a file reference and generates a definition in the `Defines` section of `model.h`:

```

%% Add a #define to the model's public header file model.h

%assign pubName = LibGetMdlPubHdrBaseName()
%assign modelH = LibCreateSourceFile("Header", "Simulink", pubName)

%openfile tmpBuf

#define ACCELERATION 9.81

%closefile tmpBuf

%<LibSetSourceFileSection(modelH, "Defines", tmpBuf)>

```

Examine the generated `rtwdemo_u dt.h` file to see the generated `#define` directive.

Customizing Main Program Module Generation

Normally, the ERT target determines whether and how to generate an `ert_main.c` or `.cpp` module based on the settings of the **Generate an example main program** and **Target operating system** options on the **Templates** pane of the Configuration Parameters dialog box. You can use a CFP template to override the normal behavior and generate a main program module customized for your target environment.

To support generation of main program modules, two TLC files are provided:

- `bareboard_srmain.tlc`: TLC code to generate an example single-rate main program module for a bareboard target environment. Code is generated by a single TLC function, `FcnSingleTaskingMain`.
- `bareboard_mrmain.tlc`: TLC code to generate a multirate main program module for a bareboard target environment. Code is generated by a single TLC function, `FcnMultiTaskingMain`.

In the example CFP template file `matlabroot/toolbox/rtw/targets/ecoder/example_file_process.tlc`, the following code generates either a single- or multitasking `ert_main.c` or `.cpp` module. The logic depends on information obtained from the code template API calls `LibIsSingleRateModel` and `LibIsSingleTasking`:

```
%% Create a simple main.  Files are located in MATLAB/rtw/c/tlc/mw.

%if LibIsSingleRateModel() || LibIsSingleTasking()
    %include "bareboard_srmain.tlc"
    %<FcnSingleTaskingMain()>
%else
    %include "bareboard_mrmain.tlc"
    %<FcnMultiTaskingMain()>
%endif
```

Note that `bareboard_srmain.tlc` and `bareboard_mrmain.tlc` use the code template API to generate `ert_main.c` or `.cpp`.

When generating your own main program module, you disable the default generation of `ert_main.c` or `.cpp`. The TLC variable `GenerateSampleERTMain` controls generation of `ert_main.c` or `.cpp`. You can directly force this variable to `TLC_FALSE`. The examples `bareboard_mrmain.tlc` and `bareboard_srmain.tlc` use this technique, as shown in the following excerpt from `bareboard_srmain.tlc`.

```
%if GenerateSampleERTMain
    %assign CompiledModel.GenerateSampleERTMain = TLC_FALSE
    %warning Overriding example ert_main.c!
%endif
```

Alternatively, you can implement a `SelectCallback` function for your target. A `SelectCallback` function is a MATLAB function that is triggered during model loading, and also when the user selects a target with the System Target File browser. Your `SelectCallback` function should deselect and disable the **Generate an example main program** option. This prevents the TLC variable `GenerateSampleERTMain` from being set to `TLC_TRUE`.

See the “`rtwgensettings` Structure” section for information on creating a `SelectCallback` function.

The following code illustrates how to deselect and disable the **Generate an example main program** option in the context of a `SelectCallback` function.

```
slConfigUISetVal(hDlg, hSrc, 'GenerateSampleERTMain', 'off');  
slConfigUISetEnabled(hDlg, hSrc, 'GenerateSampleERTMain', 0);
```

Note Creation of a main program for your target environment requires some customization; for example, in a bareboard environment you need to attach `rt_OneStep` to a timer interrupt. It is expected that you will customize either the generated code, the generating TLC code, or both. See “Guidelines for Modifying the Main Program” and “Guidelines for Modifying `rt_OneStep`” for further information.

Generate a Custom Section

You can define custom tokens in a CGT file and direct generated code into an associated built-in section. This feature gives you additional control over the formatting of code within each built-in section. For example, you could add subsections to built-in sections that do not already define subsections. Custom sections must be associated with one of the built-in sections: `Includes`, `Defines`, `Types`, `Enums`, `Definitions`, `Declarations`, or `Functions`. To create custom sections, you must

- Add a custom token to the code insertion section of your CGT file.
- In your CFP file:
 - Assemble code to be generated to the custom section into a buffer.
 - Declare an association between the custom section and a built-in section, with the code template API function `LibAddSourceFileCustomSection`.
 - Emit code to the custom section with the code template API function `LibSetSourceFileCustomSection`.

The following code examples illustrate the addition of a custom token, `Myincludes`, to a CGT file, and the subsequent association of the custom section `Myincludes` with the built-in section `Includes` in a CFP file.

Note: If you have not already created custom CGT and CFP files for your model, copy the default template files `matlabroot/toolbox/rtw/targets/ecoder/ert_code_template.cgt` and `matlabroot/toolbox/rtw/targets/ecoder/example_file_process.tlc` to a work folder that is outside the MATLAB folder structure but on the MATLAB or TLC path, rename them (for example, add the prefix `test_` to each file), and update the **Templates** pane of the Configuration Parameters dialog box to reference them.

First, add the token `Myincludes` to the code insertion section of your CGT file. For example:

```
%<Includes>
%<Myincludes>
%<Defines>
%<Types>
%<Enums>
%<Definitions>
%<Declarations>
%<Functions>
```

Next, in the CFP file, add code to generate `include` directives into a buffer. For example, in your copy of the example CFP file, you could insert the following section between the `Includes` section and the `Create a simple main` section:

```
%% Add a custom section to the model's C file model.c

%openfile tmpBuf
#include "moretables1.h"
#include "moretables2.h"
%closefile tmpBuf

%<LibAddSourceFileCustomSection(modelC,"Includes","Myincludes")>
%<LibSetSourceFileCustomSection(modelC,"Myincludes",tmpBuf)>
```

The `LibAddSourceFileCustomSection` function call declares an association between the built-in section `Includes` and the custom section `Myincludes`. `Myincludes` is a subsection of `Includes`. The `LibSetSourceFileCustomSection` function call directs the code in the `tmpBuf` buffer to the `Myincludes` section of the

generated file. `LibSetSourceFileCustomSection` is syntactically identical to `LibSetSourceFileSection`.

In the generated code, the include directives generated to the custom section appear after other code directed to `Includes`.

```
#include "rtwdemo_udt.h"
#include "rtwdemo_udt_private.h"

/* #include "mytables.h" */
#include "moretables1.h"
#include "moretables2.h"
```

Note: The placement of the custom token in this example CGT file is arbitrary. By locating `%<Myincludes>` after `%<Includes>`, the CGT file specifies only that the `Myincludes` code appears after `Includes` code.

Custom Tokens

Custom tokens are automatically translated to TLC syntax as a part of the build process. To escape a token, that is to prepare it for normal TLC expansion, use the `!` character. For example, the token `%<!TokenName>` is expanded to `%<TokenName>` by the template conversion program. You can specify valid TLC code, including TLC function calls: `%<!MyTLCFcn()>`.

Comparison of a Template and Its Generated File

This figure shows part of a user-modified custom file processing (CFP) template and the resulting generated code. The figure illustrates how you can use a template to:

- Define what code the code generation software should add to the generated file
- Control the location of code in the file
- Optionally insert comments in the generated file

Notice %<Includes>, for example, on the template. The term **Includes** is a symbol name. A percent sign and brackets (%< >) must enclose every symbol name. You can add the desired symbol name (within the %< > delimiter) at a particular location in the template. This is how you control where the code generator places an item in the generated file.

Template and Generated File



Mapping Template Specification to Code Generation

This part of the template...		Generates in the file...		Explanation
		Line	Description	
(1)	<code>/*#INCLUDES*/ %<Includes></code>	26–28	An <code>/*#INCLUDES*/</code> comment, followed by <code>#include</code> statements	The code generator adds the C/C++ comment as a header, and then interprets the <code>%<Includes></code> template symbol to list the required <code>#include</code> statements in the file. This code is first in this section of the file because the template entries are first.
(2)	<code>/*#DEFINES*/ %<Defines></code>	30	A <code>/*#DEFINES*/</code> comment, but no <code>#define</code> statements	Next, the code generator places the comment as a header for <code>#define</code> statements, but the file does not need <code>#define</code> . No code is added.
(3)	<code>#pragma string1</code>	31	<code>#pragma</code> statements	While the code generator requires <code>%<></code> delimiters for template symbols, it can also interpret C/C++ statements in the template without delimiters. In this case, the generator adds the specified statements to the code, following the order in which the statements appear in the template.
(5)	<code>#pragma string2</code>	42		
(4)	<code>/*DEFINITIONS*/ %<Definitions></code>	32–41	<code>/*DEFINITIONS*/</code> comment, followed by definitions	The code generator places the comment and definitions in the file between the <code>#pragma</code> statements, according to the order in the template. It also inserts comments (lines 33 and 36) that are preset in the model's Configuration Parameters dialog box.
(6)	<code>%<Declarations></code>	43	No declarations	The file needs no declarations, so the code generator does not generate declarations for this file. The template does not have

This part of the template...		Generates in the file...		Explanation
		Line	Description	
				a comment to provide a header. Line 43 is left blank.
(7)	%<Functions>	44-74	Functions	Finally, the code generator adds functions from the model, plus comments that are preset in the Configuration Parameters dialog box. But it adds no comments as a header for the functions, because the template does not have one. This code is last because the template entry is last.

For a list of template symbols and the rules for using them, see “Template Symbol Groups” on page 14-69, “Template Symbols” on page 14-72, and “Rules for Modifying or Creating a Template” on page 14-75. To set comment options, from the **Simulation** menu, select **Model Configuration Parameters**. On the Configuration Parameters dialog box, select the **Code Generation > Comments** pane. For details, see “Configure Code Comments”.

Code Template API Summary

Code Template API Functions summarizes the code template API. See the source code in `matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc` for detailed information on the arguments, return values, and operation of these calls.

Code Template API Functions

Function	Description
<code>LibGetNumSourceFiles</code>	Returns the number of created source files (.c or .cpp and .h).
<code>LibGetSourceFileTag</code>	Returns <code><filename>_h</code> and <code><filename>_c</code> for header and source files, respectively, where <code>filename</code> is the name of the model file.
<code>LibCreateSourceFile</code>	Creates a new C or C++ file and returns its reference. If the file already exists, simply returns its reference.
<code>LibGetSourceFileFromIdx</code>	Returns a model file reference based on its index. This is useful for a common operation on all files, such as to set the leading file banner of all files.
<code>LibSetSourceFileSection</code>	Adds to the contents of a specified section within a specified file (see also “Custom File Processing (CFP) Template Structure” on page 14-42).
<code>LibIndentSourceFile</code>	Indents a file (from within the TLC environment).
<code>LibCallModelInitialize</code>	Returns code for calling the model's <code>model_initialize</code> function (valid for ERT only).
<code>LibCallModelStep</code>	Returns code for calling the model's <code>model_step</code> function (valid for ERT only).
<code>LibCallModelTerminate</code>	Returns code for calling the model's <code>model_terminate</code> function (valid for ERT only).

Function	Description
LibCallSetEventForThisBaseStep	Returns code for calling the model's set events function (valid for ERT only).
LibWriteModelData	Returns data for the model (valid for ERT only).
LibSetRTModelErrorStatus	Returns the code to set the model error status.
LibGetRTModelErrorStatus	Returns the code to get the model error status.
LibIsSingleRateModel	Returns true if model is single rate and false otherwise.
LibGetModelName	Returns name of the model (without an extension).
LibGetMdlSrcBaseName	Returns the name of model's main source file (for example, <i>model.c</i> or <i>.cpp</i>).
LibGetMdlPubHdrBaseName	Returns the name of model's public header file (for example, <i>model.h</i>).
LibGetMdlPrvHdrBaseName	Returns the name of the model's private header file (for example, <i>model_private.h</i>).
LibIsSingleTasking	Returns true if the model is configured for single-tasking execution.
LibWriteModelInput	Returns the code to write to a particular root input (that is, a model inport block). (valid for ERT only).
LibWriteModelOutput	Returns the code to write to a particular root output (that is, a model outport block). (valid for ERT only).
LibWriteModelInputs	Returns the code to write to root inputs (that is, all model inport blocks). (valid for ERT only)
LibWriteModelOutputs	Returns the code to write to root outputs (that is, all model outport blocks). (valid for ERT only).

Function	Description
LibNumDiscreteSampleTimes	Returns the number of discrete sample times in the model.
LibSetSourceFileCodeTemplate	Set the code template to be used for generating a specified source file.
LibSetSourceFileOutputDirectory	Set the folder into which a specified source file is to be generated.
LibAddSourceFileCustomSection	Add a custom section to a source file. The custom section must be associated with one of the built-in (required) sections: Includes, Defines, Types, Enums, Definitions, Declarations, or Functions.
LibSetSourceFileCustomSection	Adds to the contents of a specified custom section within a specified file. The custom section must have been previously created with LibAddSourceFileCustomSection .
LibGetSourceFileCustomSection	Returns the contents of a specified custom section within a specified file.
LibSetCodeTemplateComplianceLevel	<p>This function must be called from your CFP template before other code template API functions are called. Pass in 2 as the <code>level</code> argument.</p> <hr/> <p>Note: Some MathWorks TLC files pass in 1 as the <code>level</code> argument. Currently, there is no difference in handling of level 1 versus level 2 by MathWorks software.</p>

Generate Custom File and Function Banners

Using code generation template (CGT) files, you can specify custom file banners and function banners for the generated code files. File banners are comment sections in the header and trailer sections of a generated file. Function banners are comment sections for each function in the generated code. Use these banners to add a company copyright statement, specify a special version symbol for your configuration management system, remove time stamps, and for many other purposes. These banners can contain characters, which propagate to the generated code.

To specify banners, create a custom CGT file with customized banner sections. The build process creates an executable TLC file from the CGT file. The code generation process then invokes the TLC file.

You do not need to be familiar with TLC programming to generate custom banners. You can modify example files that are supplied with the ERT target.

Note Prior releases supported direct use of customized TLC files as banner templates. You specified these with the **Source file (.c) banner template** and **Header file (.h) banner template** options of the ERT target. You can still use a custom TLC file banner templates, however, you can now use CGT files instead.

ERT template options on the **Code Generation > Templates** pane of a configuration set, in the **Code templates** section, support banner generation.

The options for function and file banner generation are:

- “Code templates: Source file (*.c) template”: CGT file to use when generating source (.c or .cpp) files. Place this file on the MATLAB path.
- “Code templates: Header file (*.h) template”: CGT file to use when generating header (.h) files. You must place this file on the MATLAB path. This file can be the same template specified in the **Code templates: Source file (*.c) template** field, in which case identical banners are generated in source and header files.

By default, the template for both source and header files is matlabroot/toolbox/rtw/targets/ecoder/ert_code_template.cgt.

- In each of these fields, click **Browse** to navigate to and select an existing CGT file for use as a template. Click **Edit** to open the specified file into the MATLAB editor, where you can customize it.

Create a Custom File and Function Banner Template

To customize a CGT file for custom banner generation, make a local copy of the default code template and edit it, as follows:

- 1 Activate the configuration set that you want to work with.
- 2 Open the **Code Generation** pane of the active configuration set.
- 3 Click the **Templates** tab.
- 4 By default, the code template specified in the **Code templates: Source file (*.c) template** and **Code templates: Header file (*.h) template** fields is `matlabroot/toolbox/rtw/targets/ecoder/ert_code_template.cgt`.
- 5 If you want to use a different template as your starting point, click **Browse** to locate and select a CGT file.
- 6 Click **Edit** button to open the CGT file into the MATLAB editor.
- 7 Save a local copy of the CGT file. Store the copy in a folder that is outside of the MATLAB folder structure, but on the MATLAB path. If required, add the folder to the MATLAB path.
- 8 If you intend to use the CGT file with a custom target, locate the CGT file in a folder under your target root folder.
- 9 Rename your local copy of the CGT file. When you rename the CGT file, update the associated **Code templates: Source file (*.c) template** or **Code templates: Header file (*.h) template** field to match the new file name.
- 10 Edit and customize the local copy of the CGT file for banner generation, using the information provided in “Customize a Code Generation Template (CGT) File for File and Function Banner Generation” on page 14-63.
- 11 Save your changes to the CGT file.
- 12 Click **Apply** to update the configuration set.
- 13 Save your model.
- 14 Generate code. Examine the generated source and header files to confirm that they contain the banners specified by the template or templates.

Customize a Code Generation Template (CGT) File for File and Function Banner Generation

This section describes how to edit a CGT file for custom file and function banner generation. For a description of CGT files, see “Code Generation Template (CGT) Files” on page 14-38.

Components of the File and Function Banner Sections in the CGT file

In a CGT file, you can modify the following sections: file banner, function banner, shared utility function banner, and file trailer. Each section is defined by open and close tags. The tags specific to each section are shown in the following table.

CGT File Section	Open Tag	Close Tag
File Banner	<FileBanner>	</FileBanner>
Function Banner	<FunctionBanner>	</FunctionBanner>
Shared-utility Banner	<SharedUtilityBanner>	</SharedUtilityBanner>
File Trailer	<FileTrailer>	</FileTrailer>

You can customize your banners by including tokens and comments between the open and close tag for each section. Tokens are typically TLC variables, for example <ModelVersion>, which are replaced with values in the generated code.

Note: Including C comment indicators, `/*` or `*/`, in the contents of your banner might introduce an error in the generated code.

An open tag includes tag attributes. Enclose the value of the attribute in double quotes. The attributes available for an open tag are:

- **width:** specifies the width of the file or function banner comments in the generated code. The default value is 80.
- **style:** specifies the boundary for the file or function banner comments in the generated code.

The open tag syntax is as follows:

```
<OpenTag style = "style_value" width = "num_width">
```

The built-in style options for the `style` attribute are:

- `classic`

```
/* single line comments */  
/*  
 * multiple line comments  
 * second line  
*/
```

- `classic_cpp`

```
// single line comments  
//  
// multiple line comments  
// second line  
//
```

- `box`

```
/*  
 * banner contents  
*/
```

- `box_cpp`

```
////  
// banner contents  
////
```

- `open_box`

```
/*  
 * banner contents  
*/
```

- `open_box_cpp`

```
////  
// banner contents  
////
```

- `doxygen`

```
/** single line comments */  
  
/**  
 * multiple line comments  
 * second line
```

```

    */
• doxygen_cpp

    /// single line comments

    ///
    /// multiple line comments
    /// second line
    ///

• doxygen_qt

    /*! single line comments */

    /*!
    * multiple line comments
    * second line
    */

• doxygen_qt_cpp

    //! single line comments

    //!
    //! multiple line comments
    //! second line
    //!

```

File Banner

This section contains comments and tokens for use in generating a custom file banner. The file banner precedes C or C++ code generated by the model. If you omit the file banner section from the CGT file, then no file banner emits to the generated code. The following section is the file banner section provided with the default CGT file, matlabroot/toolbox/rtw/targets/ecoder/ert_code_template.cgt.

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Custom file banner section (optional)
%%
<FileBanner style="classic">
File: %<FileName>

Code generated for Simulink model %<ModelName>.

Model version           : %<ModelVersion>
Simulink Coder version  : %<RTWFileVersion>
TLC version             : %<TLCVersion>
C/C++ source code generated on : %<SourceGeneratedOn>

```

```
%<CodeGenSettings>
</FileBanner>
```

Summary of Tokens for File Banner Generation

FileName	Name of the generated file (for example, "rtwdemo_udt.c").
FileType	Either "source" or "header". Designates whether generated file is a .c or .cpp file or an .h file.
FileTag	Given file names file.c or .cpp and file.h; the file tags are "file_c" and "file_h", respectively.
ModelName	Name of generating model.
ModelVersion	Version number of model.
RTWFileVersion	Version number of <i>model</i> .rtw file.
RTWFileGeneratedOn	Timestamp of <i>model</i> .rtw file.
TLCVersion	Version of Target Language Compiler.
SourceGeneratedOn	Timestamp of generated file.
CodeGenSettings	Code generation settings for model: target language, target selection, production hardware selection, test hardware selection, code generation objectives (in priority order), and Code Generation Advisor validation result.

Function Banner

This section contains comments and tokens for use in generating a custom function banner. The function banner precedes C or C++ function generated during the build process. If you omit the function banner section from the CGT file, the default function banner emits to the generated code. The following section is the default function banner section provided with the default CGT file, matlabroot/toolbox/rtw/targets/ecoder/ert_code_template.cgt.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Custom function banner section (optional)
%% Customize function banners by using the following predefined tokens:
%% %<ModelName>, %<FunctionName>, %<FunctionDescription>, %<Arguments>,
%% %<ReturnType>, %<GeneratedFor>, %<BlockDescription>.
%%
<FunctionBanner style="classic">
%<FunctionDescription>
%<BlockDescription>
</FunctionBanner>
```

Summary of Tokens for Function Banner Generation

FunctionName	Name of function
Arguments	List of function arguments
ReturnType	Return type of function
ModelName	Name of generating model
FunctionDescription	Short abstract about the function
GeneratedFor	Full block path for the generated function
BlockDescription	User input from the Block Description parameter of the block properties dialog box. BlockDescription contains an optional token attribute, style . The only valid value for style is content_only , which is case-sensitive and enclosed in double quotes. Use the content_only style when you want to include only the block description content that you entered in the block parameter dialog. The syntax for the token attribute style is: %<BlockDescription style = "content_only">

Shared Utility Function Banner

The shared utility function banner section contains comments and tokens for use in generating a custom shared utility function banner. The shared utility function banner precedes C or C++ shared utility function generated during the build process. If you omit the shared utility function banner section from the CGT file, the default shared utility function banner emits to the generated code. The following section is the default shared utility function banner section provided with the default CGT file, matlabroot/toolbox/rtw/targets/ecoder/ert_code_template.cgt.

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Custom shared utility function banner section (optional)
%% Customize banners for functions generated in shared location by using the
%% following predefined tokens: %<FunctionName>, %<FunctionDescription>,
%% %<Arguments>, %<ReturnType>.
%%
<SharedUtilityBanner style="classic">
%<FunctionDescription>
</SharedUtilityBanner>
    
```

Summary of Tokens for Shared Utility Function Banner Generation

FunctionName	Name of function
--------------	------------------

Arguments	List of function arguments
ReturnType	Return type of function
FunctionDescription	Short abstract about function

File Trailer

The file trailer section contains comments for generating a custom file trailer. The file trailer follows C or C++ code generated from the model. If you omit the file trailer section from the CGT file, no file trailer emits to the generated code. The following section is the default file trailer provided in the default CGT file.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
%% Custom file trailer section (optional)  
%%  
<FileTrailer style="classic">  
File trailer for generated code.  
  
[EOF]  
</FileTrailer>
```

Tokens available for the file banner are available for the file trailer. See [Summary of Tokens for File Banner Generation](#).

Template Symbols and Rules

In this section...

- “Introduction” on page 14-69
- “Template Symbol Groups” on page 14-69
- “Template Symbols” on page 14-72
- “Rules for Modifying or Creating a Template” on page 14-75

Introduction

“Template Symbol Groups” on page 14-69 and “Template Symbols” on page 14-72 describe custom file processing (CFP) template symbols and rules for using them. The location of a symbol in one of the supplied template files (`code_c_template.cgt`, `code_h_template.cgt`, `data_c_template.cgt`, or `data_h_template.cgt`) determines where the items associated with that symbol are located in the corresponding generated file. “Template Symbol Groups” on page 14-69 identifies the symbol groups, starting with the parent (“Base”) group, followed by the children of each parent. “Template Symbols” on page 14-72 lists the symbols alphabetically.

Template Symbol Groups

Symbol Group	Symbol Names in This Group
Base (Parents)	Declarations Defines Definitions Documentation Enums Functions Includes Types
Declarations	ExternalCalibrationLookup1D

Symbol Group	Symbol Names in This Group
	ExternalCalibrationLookup2D ExternalCalibrationScalar ExternalVariableScalar
Defines	LocalDefines LocalMacros
Definitions	FilescopeCalibrationLookup1D FilescopeCalibrationLookup2D FilescopeCalibrationScalar FilescopeVariableScalar GlobalCalibrationLookup1D GlobalCalibrationLookup2D GlobalCalibrationScalar GlobalVariableScalar

Symbol Group	Symbol Names in This Group
Documentation	Abstract Banner Created Creator Date Description FileName History LastModifiedDate LastModifiedBy ModelName ModelVersion ModifiedBy ModifiedComment ModifiedHistory
	Notes ToolVersion
Functions	CFunctionCode
Types	This parent has no children.

Template Symbols

Symbol Name*	Symbol Group	Symbol Scope	Symbol Description (What the symbol puts in the generated file)
Abstract	Documentation	N/A	User-supplied description of the model or file. Placed in the generated file based on the Stateflow note, Simulink annotation, or DocBlock on the model.**
Banner	Documentation	N/A	Comments located near top of the file. Contains information that includes model and software versions, and date file was generated.
CFunctionCode	Functions	File	C/C++ functions. Must be at the bottom of the template.
Created	Documentation	N/A	Date when model was created. From Created on field on Model Properties dialog box.
Creator	Documentation	N/A	User who created model. From Created by field on Model Properties dialog box.
Date	Documentation	N/A	Date file was generated. Taken from computer clock.
Declarations	Base		Data declaration of a signal or parameter. For example, <code>extern real_T globalvar;</code>
Defines	Base	File	Required <code>#defines</code> of <code>.h</code> files.
Definitions	Base	File	Data definitions of signals or parameters.
Description	Documentation	N/A	Description of model. From Model description field on Model Properties dialog box.**

Symbol Name*	Symbol Group	Symbol Scope	Symbol Description (What the symbol puts in the generated file)
Documentation	Base	N/A	Comments about how to interpret the generated files.
Enums	Base	File	Enumerated data type definitions.
ExternalCalibrationLookup1D	Declarations	External	***
ExternalCalibrationLookup2D	Declarations	External	***
ExternalCalibrationScalar	Declarations	External	***
ExternalVariableScalar	Declarations	External	***
FileName	Documentation	N/A	Name of the generated file.
FilescopeCalibrationLookup1D	Definitions	File	***
FilescopeCalibrationLookup2D	Definitions	File	***
FilescopeCalibrationScalar	Definitions	File	***
FilescopeVariableScalar	Definitions	File	***
Functions	Base	File	Generated function code.
GlobalCalibrationLookup1D	Definitions	Global	***
GlobalCalibrationLookup2D	Definitions	Global	***
GlobalCalibrationScalar	Definitions	Global	***
GlobalVariableScalar	Definitions	Global	***
History	Documentation	N/A	User-supplied revision history of the generated files. Placed in the generated file based on the Stateflow note, Simulink annotation, or DocBlock on the model.**
Includes	Base	File	<code>#include</code> preprocessor directives.

Symbol Name*	Symbol Group	Symbol Scope	Symbol Description (What the symbol puts in the generated file)
LastModifiedDate	Documentation	N/A	Date when model was last saved. From Last saved on field on Model Properties dialog box.
LastModifiedBy	Documentation	N/A	User who last saved model. From Last saved by field on Model Properties dialog box.
LocalDefines	Defines	File	#define preprocessor directives from code-generation data objects.
LocalMacros	Defines	File	C/C++ macros local to the file.
ModelName	Documentation	N/A	Name of the model.
ModelVersion	Documentation	N/A	Version number of the Simulink model. From Model version field on Model Properties dialog box.
ModifiedBy	Documentation	N/A	Name of user who last modified the model.
ModifiedComment	Documentation	N/A	Comment user enters in the Modified Comment field on the Log Change dialog box. For more information, see “Log Comments History”.
ModifiedHistory	Documentation	N/A	Text from Model history field on Model Properties dialog box.**
Notes	Documentation	N/A	User-supplied miscellaneous notes about the model or generated files. Placed in the generated file based on the Stateflow note, Simulink annotation, or DocBlock on the model.**

Symbol Name*	Symbol Group	Symbol Scope	Symbol Description (What the symbol puts in the generated file)
ToolVersion	Documentation	N/A	A list of the versions of the toolboxes used in generating the code.
Types	Base		Data types of generated code.

* Symbol names must be enclosed between %< >. For example, %<Functions>.

** This symbol can be used to add a comment to the generated files. See “Add Global Comments”. The code generator places the comment in each generated file whose template has this symbol name. The code generator places the comment at the location that corresponds to where the symbol name is located in the template file.

*** The description can be deduced from the symbol name. For example, GlobalCalibrationScalar is a symbol that identifies a scalar. It contains data of global scope that you can calibrate .

Rules for Modifying or Creating a Template

The following are the rules for creating a MPF template. “Comparison of a Template and Its Generated File” on page 14-54 illustrates several of these rules.

- 1 Place a symbol on a template within the %< > delimiter. For example, the symbol named `Includes` should look like this on a template: %<Includes>. *Note that symbol names are case sensitive.*
- 2 Place a symbol on a template where desired. Its location on the template determines where the item associated with this symbol is located in the generated file. If no item is associated with it, the symbol is ignored.
- 3 Place a C/C++ statement outside of the %< > delimiter, and on a different line than a %< > delimiter, for that statement to appear in the generated file. For example, `#pragma message ("my text")` in the template results in `#pragma message ("my text")` at the corresponding location in the generated file. Note that the statement must be compatible with your C/C++ compiler.
- 4 Use the `.cgt` extension for every template filename. ("`cgt`" stands for code generation template.)

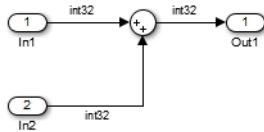
- 5** Note that `%% $Revision: 1.1.4.10.4.1 $` appears at the top of the MathWorks supplied templates. This is for internal MathWorks use only. It does not need to be placed on a user-defined template and does not show in a generated file.
- 6** Place a comment on the template between `/* */` as in standard ANSI C⁴. This results in `/*comment*/` on the generated file.
- 7** Each MPF template must have all of the Base group symbols, in predefined order. They are listed in “Template Symbol Groups” on page 14-69. Each symbol in the Base group is a parent. For example, `Declarations` is a parent symbol.
- 8** Each symbol in a non-Base group is a child. For example, `LocalMacros` is a child.
- 9** Except for Documentation children, children must be placed after their parent, before the next parent, and before the `Functions` symbol.
- 10** Documentation children can be located before or after their parent in any order anywhere in the template.
- 11** If a non-Documentation child is missing from the template, the code generator places the information associated with this child at its parent location in the generated file.
- 12** If a Documentation child is missing from the template, the code generator omits the information associated with that child from the generated file.

4. ANSI is a registered trademark of the American National Standards Institute, Inc.

Code Annotation for Justifying Polyspace Checks

With the Polyspace Code Prover™ product you can apply Polyspace verification to Embedded Coder generated code. The software detects run-time errors in the generated code and helps you to locate and fix model faults.

Polyspace might highlight overflows for certain operations that are legitimate because of the way Embedded Coder implements these operations. Consider the following model and the corresponding generated code.



```

32 /* Sum: '<Root>/Sum' incorporates:
33  * Inport: '<Root>/In1'
34  * Inport: '<Root>/In2'
35  */
36 qY_0 = sat_add_U.In1 + sat_add_U.In2;
37 if ((sat_add_U.In1 < 0) && ((sat_add_U.In2 < 0) && (qY_0 >= 0))) {
38     qY_0 = MIN_int32_T;
39 } else {
40     if ((sat_add_U.In1 > 0) && ((sat_add_U.In2 > 0) && (qY_0 <= 0))) {
41         qY_0 = MAX_int32_T;
42     }
43 }

```

Embedded Coder software recognizes that the largest built-in data type is 32-bit. It is not possible to saturate the results of the additions and subtractions using `MIN_INT32` and `MAX_INT32` and a bigger single-word integer data type. Instead the software detects the results overflow and the direction of the overflow, and saturates the result.

If you do not provide justification for the addition operator on line 36, a Polyspace verification generates an orange check that indicates a potential overflow. The verification does not take into account the saturation function of lines 37 to 43. In addition, the trace-back functionality of Polyspace Code Prover does not identify the reason for the orange check.

To justify overflows from operators that are legitimate, on the **Configuration Parameters > Code Generation > Comments** pane:

- Under **Overall control**, select the **Include comments** check box.
- Under **Auto generate comments**, select the **Operator annotations check box**.

When you generate code, the Embedded Coder software annotates the code with comments for Polyspace. For example:

```
32 /* Sum: '<Root>/Sum' incorporates:
33    * Inport: '<Root>/In1'
34    * Inport: '<Root>/In2'
35    */
36 qY_0 = sat_add_U.In1 +/*MW:0v0k*/ sat_add_U.In2;
```

When you run a verification using Polyspace Code Prover, the Polyspace software uses the annotations to justify the operator-related orange checks and assigns the **Not a defect** classification to the checks.

Manage Placement of Data Definitions and Declarations

In this section...

“Overview of Data Placement” on page 14-79

“Priority and Usage” on page 14-80

“Ownership Settings” on page 14-85

“Memory Section Settings” on page 14-86

“Data Placement Rules” on page 14-86

“Settings for a Data Object” on page 14-86

“Data Placement Rules and Results” on page 14-94

Overview of Data Placement

This chapter focuses on module packaging features (MPF) settings that are interdependent. Their combined values, along with Simulink partitioning, determine the file placement of data definitions and declarations, or *data placement*. This includes

- The number of files generated.
- Whether or not the generated files contain definitions for a model's global identifiers. And, if a definition exists, the settings determine the files in which MPF places them.
- Where MPF places global data declarations (**extern**).

The following six MPF settings are distributed among the main procedures and form an important interdependency:

- The **Data definition** field on the **Code Placement** pane of the Configuration Parameters dialog box.
- The **Data declaration** field on the **Code Placement** pane of the Configuration Parameters dialog box.
- The **Owner** field of the data object in the Model Explorer and the checkbox for **Use owner from data object for data definition placement** on the **Code Placement** pane of the Configuration Parameters dialog box. The term "ownership settings" refers to these fields together.
- The **Definition file** field of the data object on the Model Explorer.
- The **Header file** field of the data object on the Model Explorer.
- The **Memory section** field of the data object on the Model Explorer.

Priority and Usage

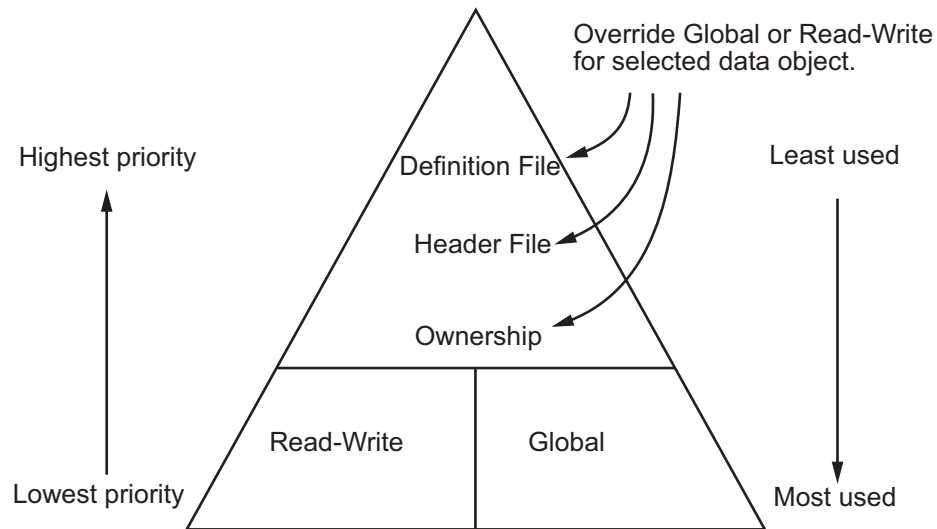
- “Overview” on page 14-80
- “Read-Write Priority” on page 14-81
- “Global Priority” on page 14-84
- “Definition File, Header File, and Ownership Priorities” on page 14-85

Overview

There is a priority order among interdependent MPF settings. From highest to lowest, the priorities are

- Definition File priority
- Header File priority
- Ownership priority
- Read-Write priority or Global priority

Priority order varies inversely with frequency of use, as illustrated below. For example, Definition File is highest priority but least used.



MPF Settings Priority and Usage

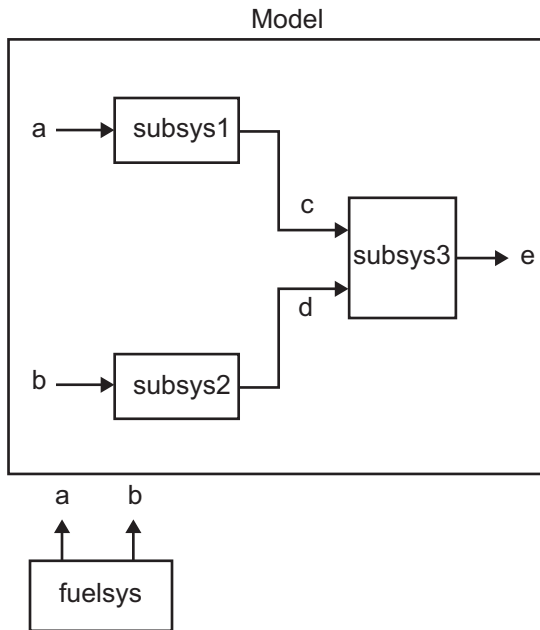
Unless they are overridden, the Read-Write and Global priorities place in the generated files all of the model's MPF-derived data objects that you selected using Data Object Wizard. (See “Create Data Objects with Data Object Wizard” for details.) Before generating the files, you can use the higher priority Definition file, Header file, and Ownership, as desired, to override Read-Write or Global priorities for single data objects. Most users will employ Read-Write or Global, without an override. A few users, however, will want to do an override for certain data objects. We expect that those users whose applications include multiple modules will want to use the Ownership priority.

The priorities are used only for those data objects that are derived from `Simulink.Signal` and `Simulink.Parameter`, and whose custom storage classes are specified using the Custom Storage Class Designer. (For details, see “Design Custom Storage Classes and Memory Sections”.) Otherwise, the build process determines the data placement.

Read-Write Priority

This is the lowest priority. Consider that a model consists of one or more Simulink blocks or Stateflow diagrams. There can be subsystems within these. For the purpose of illustration, think of a model with one top-level block called `fuelsys`. You double-clicked the block and now see three subsystems labeled `subsys1`, `subsys2` and `subsys3`, as shown in the next figure. Signals `a` and `b` are outputs from the top-level block (`fuelsys`). Signal `a` is an input to `subsys1` and `b` is input to `subsys2`. Signal `c` is an output from `subsys1`. Notice the other inputs and outputs (`d` and `e`). Signals `a` through `e` have corresponding data objects.

As explained in “Data Definition and Declaration Management”, MPF provides you with the means of selecting a data object that you want defined as an identifier in the generated code. MPF also allows you to specify property values for each data object.

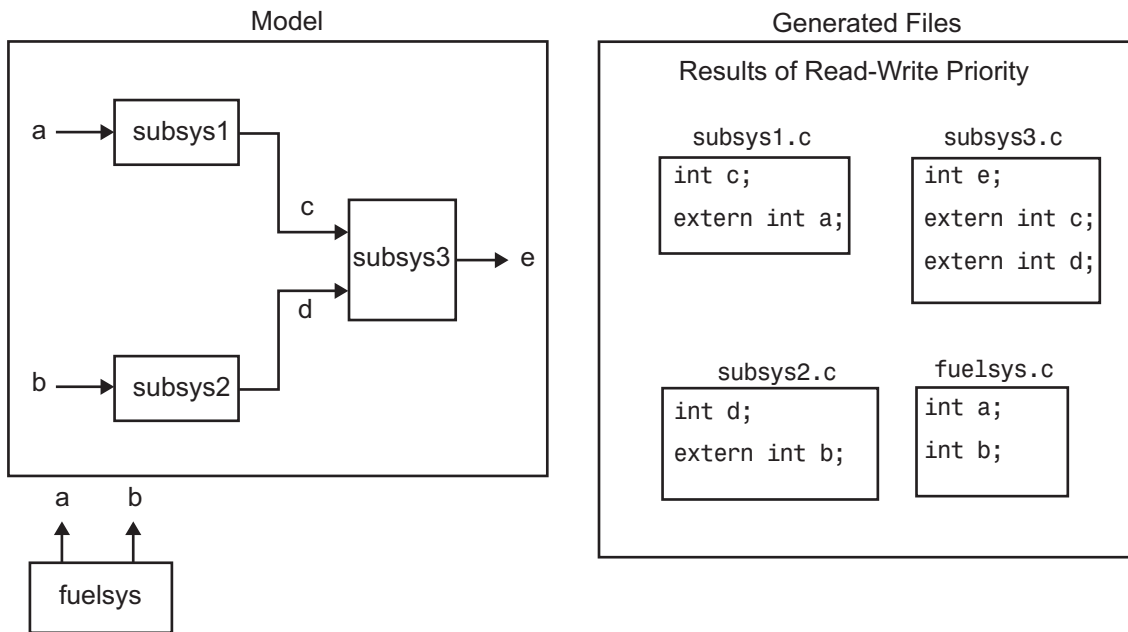


The Generated Files

We generate code for this model. As shown in the figure below, this results in a `.c` source file corresponding to each of the subsystems. (In actual applications, there could be more than one `.c` source file for a subsystem. This is based on the file partitioning previously selected for the model. But for our illustration, we only need to show one for each subsystem.) Data objects `a` through `e` have corresponding identifiers in the generated files.

A `.c` source file has one or more functions in it, depending on the internal operations (functions) of its corresponding subsystem. An identifier in a generated `.c` file has local scope when it is used only in one function of that `.c` file. An identifier has file scope when more than one function in the same `.c` file uses it. An identifier has global scope when more than one of the generated files uses it.

A subsystem's source file contains the definitions for that subsystem's data objects that have local scope or file scope. (These definitions are not shown in the figure.) But where are the definitions and declarations for data objects of global scope? These are shown in the next figure.



For the Read-Write priority, this source file contains the definitions for the subsystem's global data objects, if this is the file that first writes to the data object's address. Other files that read (use) that data object only include a reference to it. This is why this priority is called Read-Write. Since a read and a write of a file are analogous to input and output of a model's block, respectively, there is another way of saying this. The definitions of a block's global data objects are located in the corresponding generated file, if that data object is an output from that block. The declarations (`extern`) of a block's global data objects are located in the corresponding generated file, if that data object is an input to that block.

Settings for Read-Write Priority

The generated files and what they include, as just described, occur when the Read-Write priority is used. For this to be the case, the other priorities are turned off. That is,

- The **Data definition** field on the **Code Placement** pane is set to **Data defined in source file**.
- The **Data declaration** field on the **Code Placement** pane is set to **Data declared in source file**.

- The **Owner** field on the Model Explorer is blank, and the checkbox for the **Use owner from data object for data definition placement** field on the **Code Placement** pane is not checked.
- **Definition file** and **Header file** on the Model Explorer are blank.

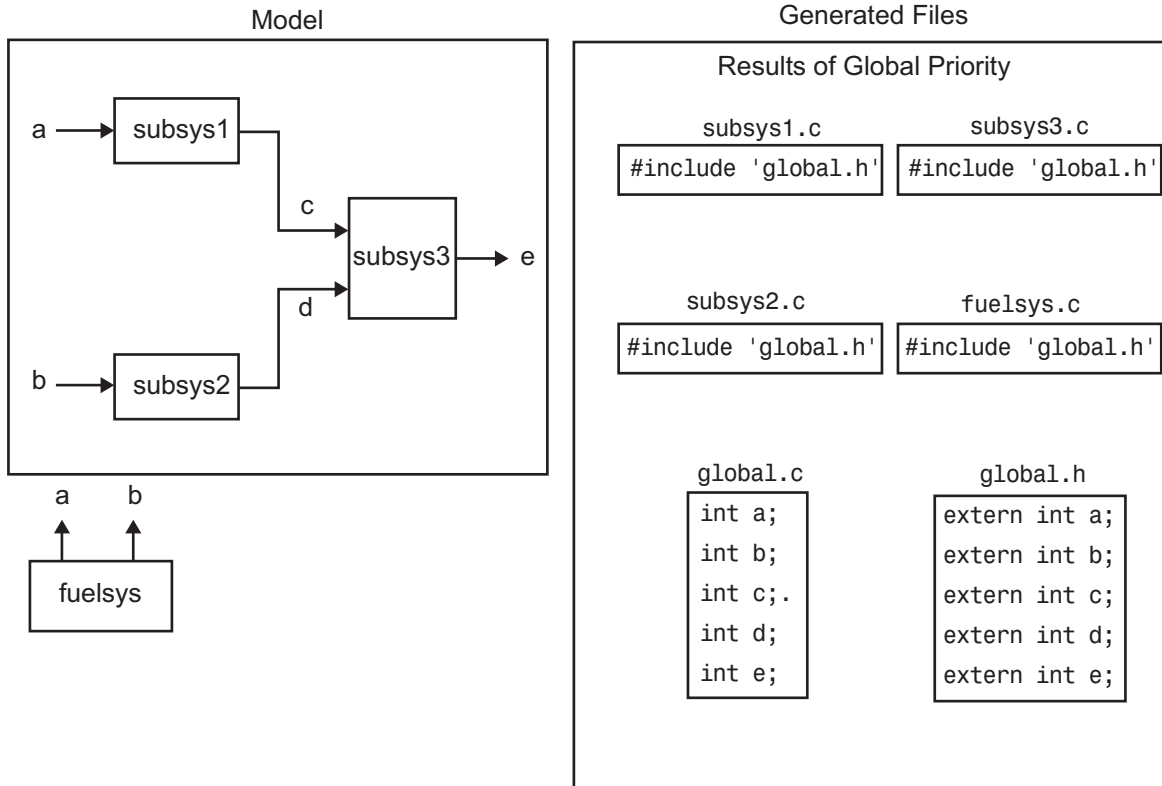
Global Priority

This has the same priority as Read-Write (the lowest) priority. The settings for this are the same as for Read-Write Priority, except

- The **Data definition** field on the **Code Placement** pane is set to `Data defined in single separate source file`.
- The **Data declaration** field on the **Code Placement** pane is set to `Data declared in single separate header file`.

The generated files that result are shown in the next figure. A subsystem's data objects of local or file scope are defined in the `.c` source file where the subsystem's functions are located (not shown). The data objects of global scope are defined in another `.c` file (called `global.c` in the figure). The declarations for the subsystem's data objects of global scope are placed in a `.h` file (called `global.h`).

For example, data objects of local and file scope for `subsys1` are defined in `subsys1.c`. Signal `c` in the model is an output of `subsys1` and an input to `subsys2`. So `c` is used by more than one subsystem and thus is a global data object. Because of the global priority, the definition for `c` (`int c;`) is in `global.c`. The declaration for `c` (`extern int c;`) is in `global.h`. Since `subsys2` uses (reads) `c`, `#include "global.h"` is in `subsys2.c`.



Definition File, Header File, and Ownership Priorities

While the Read-Write and Global priorities operate on all MPF-derived data objects that you want defined in the generated code, the remaining priorities allow you to override the Read-Write or Global priorities for one or more particular data objects. There is a high-to-low priority among these remaining priorities — Definition File, Header File, and Ownership — for a particular data object, as shown in MPF Settings Priority and Usage

Ownership Settings

Ownership settings refers to the **on** or **off** setting specified using the **Use owner from data object for data definition placement** checkbox on the **Code Placement** pane of the Configuration Parameters dialog box, and the **Owner** field of a data object in the Model Explorer. These settings do not control what files are generated. These settings

only specify definitions and `extern` statements. There are four possible configurations, as shown in “Ownership Settings” on page 14-95.

Memory Section Settings

Memory sections allow you to specify storage directives for a data object. As shown in “Parameter and Signal Property Values”, the possible values for the **Memory section** property of a parameter or signal object are `Default`, `MemConst`, `MemVolatile` or `MemConstVolatile`.

If you specify a filename for **Definition file**, and select `Default`, `MemConst`, `MemVolatile` or `MemConstVolatile` for the **Memory section** property, the code generation software generates a `.c` file and an `.h` file. The `.c` file contains the definition for the data object with the `pragma` statement or qualifier associated with the **Memory section** selection. The `.h` file contains the declaration for the data object. The `.h` file can be included, using the preprocessor directive `#include`, in files that need to reference the data object.

You can add more memory sections. For more information, see “Design Custom Storage Classes and Memory Sections” and “Memory Sections”.

Data Placement Rules

For a complete set of data placement rules in convenient tabular form, based on the priorities discussed in this chapter, see “Data Placement Rules and Results” on page 14-94.

Settings for a Data Object

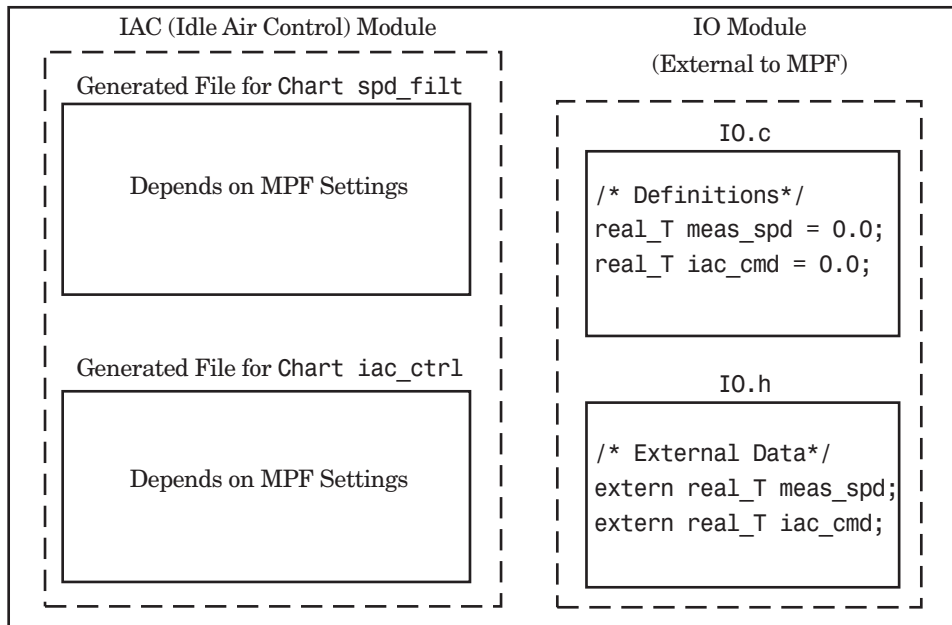
- “Introduction” on page 14-86
- “Read-Write” on page 14-88
- “Ownership” on page 14-89
- “Header File” on page 14-91
- “Definition File” on page 14-93

Introduction

“Settings and Resulting Generated Files” on page 14-95 provides example settings for one data object of a model. Eight examples are listed so that you can see the generated

files that result from a wide variety of settings. Four examples from this table are discussed below in more detail. These discussions provide information for understanding settings you might choose. For illustration purposes, the four examples assume that we are dealing with an overall system that controls engine idle speed.

The next figure shows that the software component of this example system consists of two modules, IAC (Idle Air Control), and IO (Input-Output).

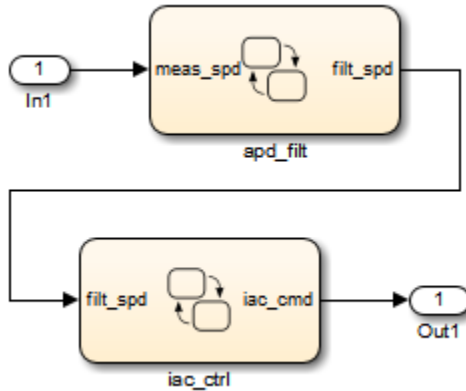


Engine Idle Speed Control System

The code in the IO module controls the system's IO hardware. Code is generated only for the IAC module. (Some other means produced the code for the IO module, such as hand-coding.) So the code in IO is external to MPF, and can illustrate legacy code. To simplify matters, the IO code contains one source file, called `IO.c`, and one header file, called `IO.h`.

The IAC module consists of two Stateflow charts, `spd_filt` and `iac_ctrl`. The `spd_filt` chart has two signals (`meas_spd` and `filt_spd`), and one parameter (`a`). The `iac_ctrl` chart also has two signals (`filt_spd` and `iac_cmd`) and a parameter (`ref_spd`). (The parameters are not visible in the top-level charts.) One file for each

chart is generated. This example system allows us to illustrate referencing from file to file within the MPF module, and model to external module. It also illustrates the case where there is no such referencing.



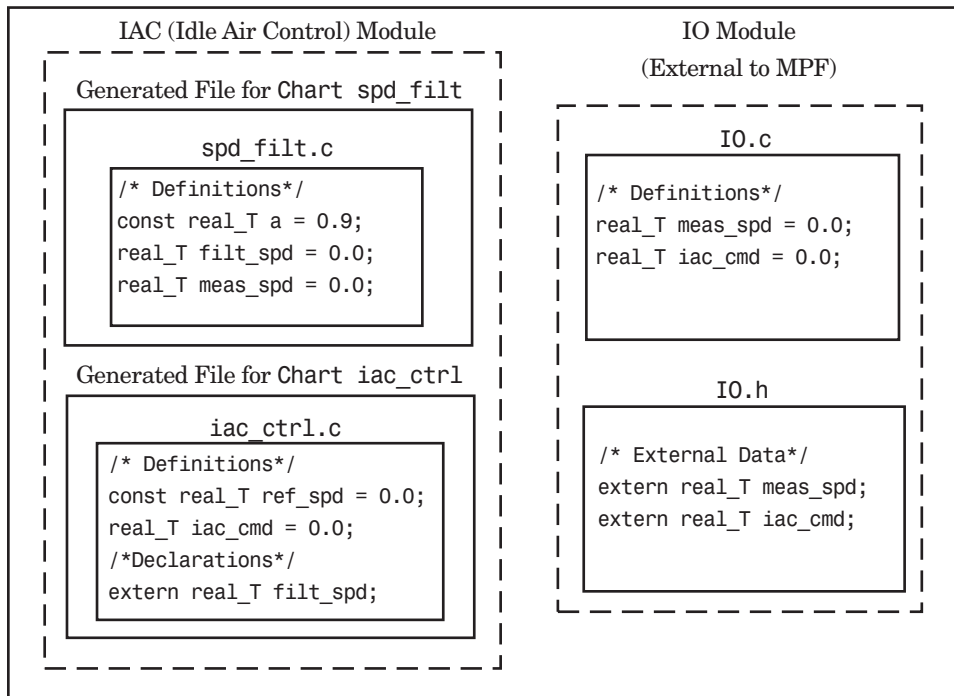
Proceed to the discussion of the desired example settings:

- “Read-Write” on page 14-88
- “Ownership” on page 14-89
- “Header File” on page 14-91
- “Definition File” on page 14-93

Read-Write

These settings and the generated files that result are shown as Example Settings 1 in “Settings and Resulting Generated Files” on page 14-95. As you can see from the table, this example illustrates the case in which only one .c source file (for each chart) is generated.

So, for the IAC model, select the following settings. Accept the **Data defined in source file** in the **Data definition** field and the **Data declared in source file** in the **Data declaration** field on the **Code Placement** pane of the Configuration Parameters dialog box. Accept the default unchecked **Use owner from data object for data definition placement** field. Accept the default blank settings for the **Owner**, **Definition file** and **Header file** fields on the Model Explorer. For **Memory section**, accept **Default**. Now the Read-Write priority is active. Generate code. The next figure shows the results in terms of definition and declaration statements.



Engine Idle Speed Control System (Read-Write Example)

The code generator generated a `spd_filt.c` for the `spd_filt` chart and `iac_ctrl.c` for the `iac_ctrl` chart. As you can see, MPF placed definitions of data objects for the `spd_filt` chart in `spd_filt.c`. It placed definitions of data objects for the `iac_ctrl` chart in `iac_ctrl.c`.

However, notice `real_T filt_spd`. This data object is defined in `spd_filt.c` and declared in `iac_ctrl.c`. That is, since the Read-Write priority is active, `filt_spd` is defined in the file that first writes to its address. And, it is declared in the file that reads (uses) it. Further, `real_T meas_spd` is defined in both `spd_filt.c` and the external `IO.c`. And, `real_T iac_cmd` is defined in both `iac_ctrl.c` and `IO.c`.

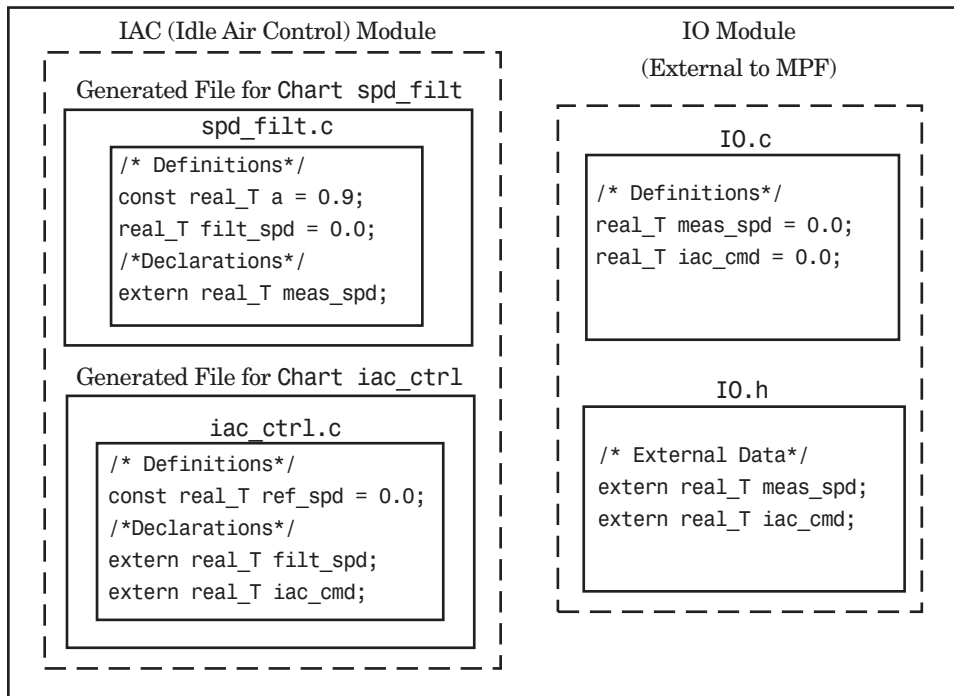
Ownership

See tables “Ownership Settings” on page 14-95 and “Settings and Resulting Generated Files” on page 14-95. In the “Read-Write” on page 14-88, there are several instances where the same data object is defined in more than one `.c` source file,

and there is no declaration (`extern`) statement. This would result in compiler errors during link time. But in this example, we configure MPF Ownership rules so that linking can take place. Notice the Example Settings 2 row in “Settings and Resulting Generated Files” on page 14-95. Except for the ownership settings, assume these are the settings you made for the model in the IAC module. Since this example has no **Definition file** or **Header file** specified, now Ownership takes priority. (If you specified a **Definition file** or **Header file**, MPF ignores the ownership settings.)

On the **Code Placement** pane of the Configuration Parameters dialog box, check the box for the **Use owner from data object for data definition placement** field. Open the Model Explorer (by issuing the MATLAB command `daexplr`) and, for all data objects except `meas_spd` and `iac_cmd`, type IAC in the **Owner** field (case sensitive). Then, only for the `meas_spd` and `iac_cmd` data objects, type IO as their **Owner** (case sensitive). Generate code.

The results are shown in the next figure. Notice the `extern real_T meas_spd` statement in `spd_filt.c`, and `extern real_T iac_cmd` in `iac_ctrl.c`. MPF placed these declaration statements in the files where these data objects are used. This allows the generated source files (`spd_filt.c` and `iac_ctrl.c`) to be compiled and linked with `IO.c`.



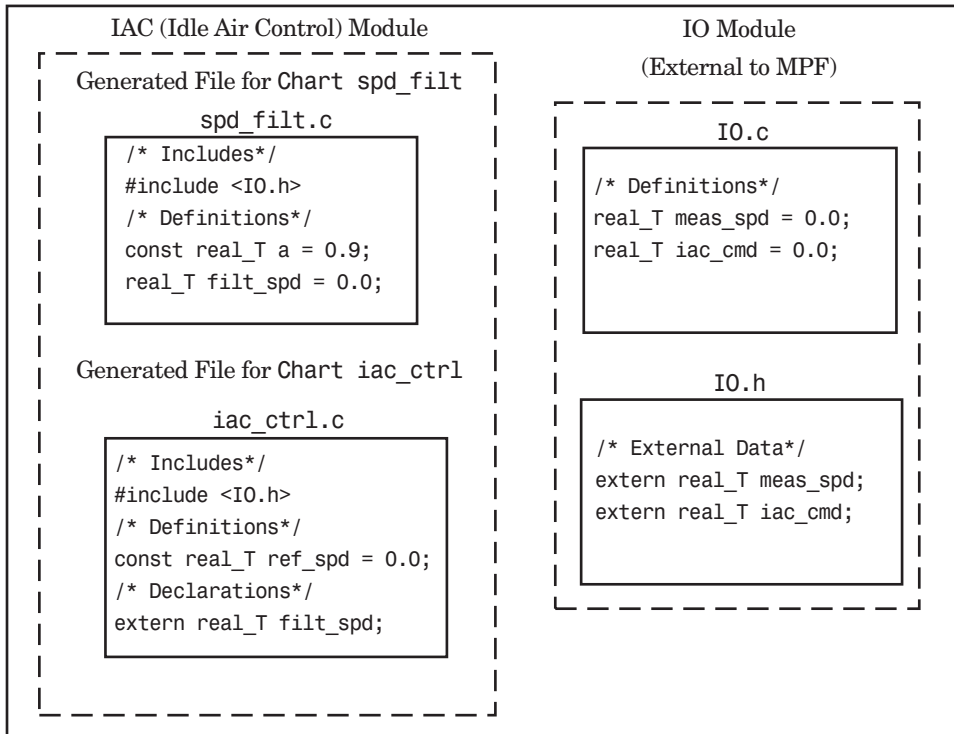
Engine Idle Speed Control System (Ownership Example)

Header File

These settings and the generated files that result are shown as Example Settings 3 in “Settings and Resulting Generated Files” on page 14-95. This example has no **Definition file** specified. If you specified a **Definition file**, MPF ignores the **Header file** setting. The focus of this example is to show how the **Header file** settings result in the linking of the two chart source files to the external IO files, shown in the next figure. (Also, ownership settings will be used to link the two chart files with each other.)

As you can see in the figure, the `meas_spd` and `iac_cmd` identifiers are defined in `IO.c` and declared in `IO.h`. Both of these identifiers are external to the generated `.c` files. You open the Model Explorer and select both the `meas_spd` and `iac_cmd` data objects. For each of these data objects, in the **Header file** field, specify `IO.h`, since this is where these two objects are declared. This setting allows the `spd_filt.c` source file to compile and link with the external `IO.c` file.

Now we configure the ownership settings. In the Model Explorer, select the `filt_spd` data object and set its **Owner** field to **IAC**. Then, on the **Code Placement** pane of the Configuration Parameters dialog box, check the box for the **Use owner from data object for data definition placement** field. Now the `spd_filt` source file links to the `iac_ctrl` source file. Generate code. See the figure below.



Engine Idle Speed Control System (Header File Example)

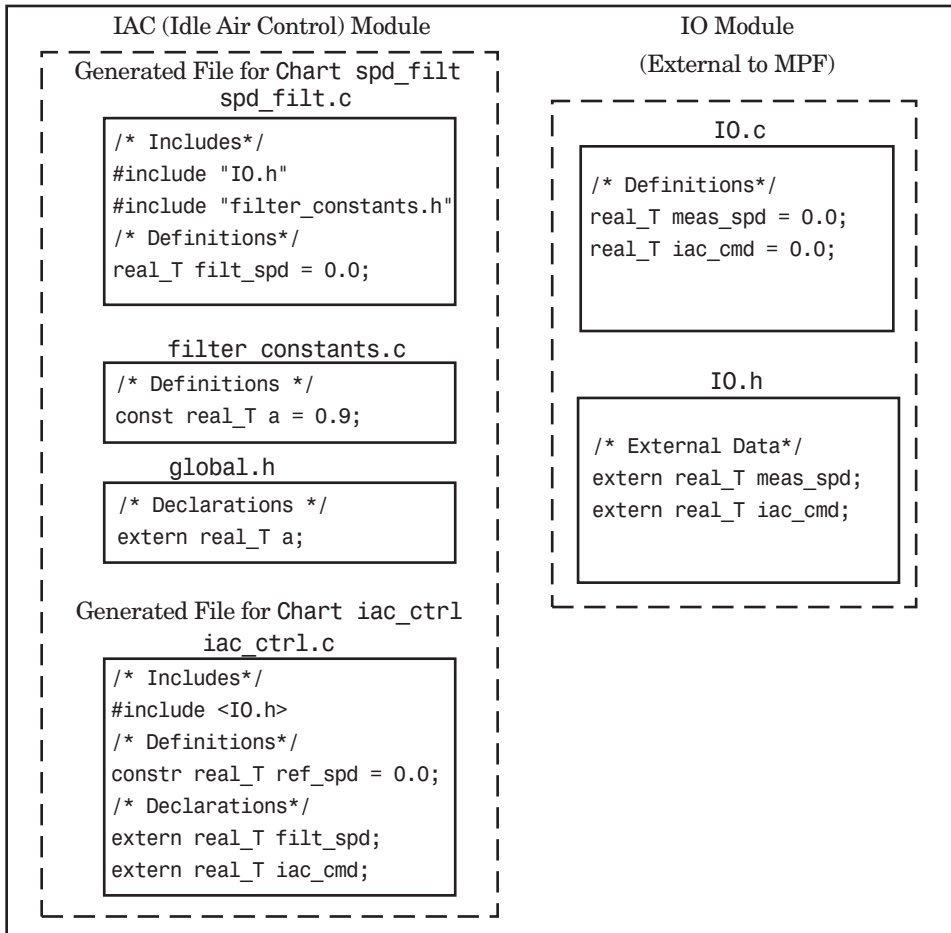
Since you specified the `IO.h` filename for the **Header file** field for the `meas_spd` and `iac_ctrl` objects, the code generator assumed that their declarations are in `IO.h`. So the code generator placed `#include IO.h` in each source file: `spd_filt.c` and `iac_ctrl.c`. So these two files will link with the external IO files. Also, due to the ownership settings that were specified, the code generator places the `real_T filt_spd = 0.0;` definition in `spd_filt.c` and declares the `filt_spd` identifier in `iac_ctrl.c` with `extern real_T iac_cmd;`. Consequently, the two source files will link together.

Definition File

These settings and the generated files that result are shown as Example Settings 4 in “Settings and Resulting Generated Files” on page 14-95. Notice that a definition filename is specified. The settings in the table only apply to the data object called **a**. You have decided that you do not want this object defined in `spd_filt.c`, the generated source file for the `spd_filt` chart. (There are many possible organizational reasons one might want an object declared in another file. It is not important for this example to specify the reason.)

For this example, assume the settings for all data objects are the same as those indicated in “Header File” on page 14-91, except for the data object **a**. The description below identifies only the differences that result from this.

Open the Model Explorer, and select data object **a**. In the **Definition file** field specify a filename. Choose `filter_constants.c`. Generate code. The results are shown in the next figure.



Engine Idle Speed Control System (Definition File Example)

The code generator generates the same files as in the “Header File” on page 14-91, and adds a new file, `filter_constants.c`. Data object `a` now is defined in `filter_constants.c`, rather than in the source file `spd_filt.c`, as it is in the example. This data object is declared with an `extern` statement in `global.h`.

Data Placement Rules and Results

- “Ownership Settings” on page 14-95

- “Settings and Resulting Generated Files” on page 14-95
- “Data Placement Rules” on page 14-97

Ownership Settings

Row Number	Enable Data Ownership Checkbox	Owner Setting	Result*
1	Off**	Blank**	Embedded Coder determines whether the current model defines data.
2	Off**	A name is specified.	Embedded Coder determines whether the current model defines data.
3	On	Blank**	Embedded Coder determines whether the current model defines data.
4	On	A name is specified.	The model specified in the Owner setting defines data.

* See also “Ownership Settings” on page 14-85.

** Default.

Settings and Resulting Generated Files

	Data Defined In...	Data Declared In...	Owner-ship*	Defined File**	Header File	Generated Files
Example Settings 1 (Rd-Write Example)	Source file	Source file	Blank	Blank	Blank	.c / .cpp source file
Example Settings 2 (Owner-ship Example)	Source file	Source file	Name of module specified	Blank	Blank	.c / .cpp source file
Example Settings 3 (Header File Example)	Source file	Source file	Blank	Blank	Desired include filename specified.	.c / .cpp source file .h definition file
Example Settings 4	Source file	Source file	Blank	Desired definition	Desired include	.c / .cpp source file

	Data Defined In...	Data Declared In...	Ownership*	Defined File**	Header File	Generated Files
(Def. File Example)				filename specified.	filename specified.	.c/.cpp definition file* .h definition file*
Example Settings 5	Single separate source file	Source file	Blank	Blank	Blank	.c/.cpp source file global.c/.cpp
Example Settings 6	Single separate source file	Single separate header file	Blank	Blank	Blank	.c/.cpp source file global.c/.cpp global.h
Example Settings 7	Single separate source file	Single separate header file	Name of module specified	Blank	Blank	.c/.cpp source file global.c/.cpp global.h
Example Settings 8	Single separate source file	Single separate header file	Blank	Blank	Desired include filename specified.	.c/.cpp source file global.c/.cpp global.h .h definition file

* "Blank" in ownership setting means that the check box for the **Use owner from data object for data definition placement** field on the **Code Placement** pane is **Off** and the **Owner** field on the Model Explorer is blank. "Name of module specified" can be a variety of ownership settings as defined in "Ownership Settings" on page 14-95.

** The code generator generates a definition .c/.cpp file for every data object for which you specified a definition filename (unless you selected **#DEFINE** for the **Memory section** field). For example, if you specify the same definition filename for all data objects, only one definition .c/.cpp file is generated. The code generator places declarations in *model.h* by default, unless you specify **Data declared in single separate header file** for the **Data declaration** option on the **Code Generation > Code Placement** pane of the Configuration Parameter dialog box. If you select that data placement option, the code generator places declarations in *global.h*. If you specify a definition filename for each data object, the code generator generates one definition .c/.cpp file for each data object and places declarations in *model.h* by default, unless you specify **Data declared in single separate header file** for **Data declaration**. If you select that data placement option, the code generator places declarations in *global.h*.

Note: If you generate C++ rather than C code, the .c files listed in the following table will be .cpp files.

Data Placement Rules

Storage Class Setting	Global Settings:		Override Settings for Specific Data Object:			Results in Generated Files:		
	Data Def.	Data Dec.	Def. File	Owner	Header File	Where Data Def. Is	Where Data Dec. Is	Dec. Inclusion
<i>mpt or Simulink Noncustom Storage Classes:</i>								
auto	N/A	N/A	N/A	N/A	N/A	Note 12	model.h	Note 1
Exported-- Global	N/A	N/A	N/A	N/A	N/A	model.c	model.h	Note 1
Imported-- Extern, Imported-- Extern-- Pointer	N/A	N/A	N/A	N/A	N/A	None. External	model_ private.h	Note 2
Simulink-- Global	N/A	N/A	N/A	N/A	N/A	Note 13	model.h	Note 1
<i>mpt or Simulink Custom Storage Class: Imported Data:</i>								
Imported-- FromFile	D/C	D/C	D/C	N/A	null	None	model_ private.h	Note 3
Imported-- FromFile	D/C	D/C	D/C	N/A	hdr.h	None	model_ private.h	Note 4
<i>Simulink Custom Storage Class: #define Data:</i>								
Define	D/C	D/C	N/A	N/A	N/A	N/A	#define, model.h	Note 5
<i>mpt Custom Storage Class: #define Data:</i>								
Define	D/C	D/C	N/A	N/A	null	N/A	#define, model.h	Note 5

Storage Class Setting	Global Settings:		Override Settings for Specific Data Object:			Results in Generated Files:		
	Data Def.	Data Dec.	Def. File	Owner	Header File	Where Data Def. Is	Where Data Dec. Is	Dec. Inclusion
Define	D/C	D/C	N/A	N/A	hdr.h	N/A	#define, model.h	Note 6
<i>mpt or Simulink Custom Storage Class: GetSet:</i>								
GetSet	D/C	D/C	N/A	N/A	hdr.h	N/A	External hdr.h	Note 4
<i>mpt or Simulink Custom Storage Class: Bitfield, Struct:</i>								
Bitfield, Struct	D/C	D/C	N/A	N/A	N/A	model.c	model.h	Note 7
<i>mpt Custom Storage Class: Global, Const, ConstVolatile, Volatile:</i>								
Global, Const, Const--Volatile, Volatile	auto	auto	null	null or locally owned	null	model.c	model.h	Note 1
Global, Const, Const--Volatile, Volatile	src	auto	null	null or locally owned	null	src.c	model.h	Note 1
Global, Const, Const--Volatile, Volatile	sep	auto	null	null or locally owned	null	gbl.c	model.h	Note 1
Global, Const, Const--Volatile, Volatile	auto	src	null	null or locally owned	null	model.c	src.c	Note 8

Storage Class Setting	Global Settings:		Override Settings for Specific Data Object:			Results in Generated Files:		
	Data Def.	Data Dec.	Def. File	Owner	Header File	Where Data Def. Is	Where Data Dec. Is	Dec. Inclusion
Global, Const, Const--Volatile, Volatile	src	src	null	null or locally owned	null	src.c	src.c	Note 8
Global, Const, Const--Volatile, Volatile	sep	src	null	null or locally owned	null	gbl.c	src.c	Note 8
Global, Const, Const--Volatile, Volatile	auto	sep	null	null or locally owned	null	model.c	gbl.h	Note 9
Global, Const, Const--Volatile, Volatile	src	sep	null	null or locally owned	null	src.c	gbl.h	Note 9
Global, Const, Const--Volatile, Volatile	sep	sep	null	null or locally owned	null	gbl.c	gbl.h	Note 9
Global, Const, Const--Volatile, Volatile	D/C	D/C	data.c	D/C	null	data.c	See Note 10.	Note 10
Global, Const, Const--Volatile, Volatile	D/C	D/C	data.c	D/C	hdr.h	data.c	hdr.h	Note 11

Storage Class Setting	Global Settings:		Override Settings for Specific Data Object:			Results in Generated Files:		
	Data Def.	Data Dec.	Def. File	Owner	Header File	Where Data Def. Is	Where Data Dec. Is	Dec. Inclusion
Global, Const, Const--Volatile, Volatile	auto	D/C	null	null	hdr.h	model.c	hdr.h	Note 11
Global, Const, Const--Volatile, Volatile	src	D/C	null	null	hdr.h	src.c	hdr.h	Note 11
Global, Const, Const--Volatile, Volatile	sep	D/C	null	null	hdr.h	gbl.c	hdr.h	Note 11
Global, Const, Const--Volatile, Volatile	D/C	auto	null	External owner	null	External user--supplied file	model.h	Note 1
Global, Const, Const--Volatile, Volatile	D/C	src	null	External owner	null	External user--supplied file	src.c	Note 8
Global, Const, Const--Volatile, Volatile	D/C	sep	null	External owner	null	External user--supplied file	gbl.h	Note 9
Global, Const, Const--Volatile, Volatile	D/C	D/C	null	External owner	header.h	External user--supplied file	hdr.h	Note 11

Storage Class Setting	Global Settings:		Override Settings for Specific Data Object:			Results in Generated Files:		
	Data Def.	Data Dec.	Def. File	Owner	Header File	Where Data Def. Is	Where Data Dec. Is	Dec. Inclusion
Global, Const, Const--Volatile, Volatile	D/C	D/C	null	External owner	header.h	External user--supplied file	hdr.h	Note 11
<i>mpt Custom Storage Class: Exported Data:</i>								
ExportTo-File	auto	auto	null	null	null	model.c	model.h	Note 1
ExportTo-File	src	auto	null	null	null	src.c	model.h	Note 1
ExportTo-File	sep	auto	null	null	null	gbl.c	model.h	Note 1
ExportTo-File	auto	src	null	null	null	model.c	src.c	Note 8
ExportTo-File	src	src	null	null	null	src.c	src.c	Note 8
ExportTo-File	sep	src	null	null	null	gbl.c	src.c	Note 8
ExportTo-File	auto	sep	null	null	null	model.c	gbl.h	Note 9
ExportTo-File	src	sep	null	null	null	src.c	gbl.h	Note 9
ExportTo-File	sep	sep	null	null	null	gbl.c	gbl.h	Note 9
ExportTo-File	D/C	D/C	data.c	null	null	data.c	See Note 10.	Note 10
ExportTo-File	D/C	D/C	data.c	null	hdr.h	model.c	hdr.h	Note 11
ExportTo-File	auto	D/C	null	null	hdr.h	src.c	hdr.h	Note 11
ExportTo-File	sep	D/C	null	null	hdr.h	gbl.c	hdr.h	Note 11
<i>Simulink Custom Storage Class: Default, Const, ConstVolatile, Volatile:</i>								
Default, Const, Const--Volatile, Volatile	auto	auto	N/A	N/A	N/A	model.c	model.h	Note 1

Storage Class Setting	Global Settings:		Override Settings for Specific Data Object:			Results in Generated Files:		
	Data Def.	Data Dec.	Def. File	Owner	Header File	Where Data Def. Is	Where Data Dec. Is	Dec. Inclusion
Default, Const, Const--Volatile, Volatile	src	auto	N/A	N/A	N/A	src.c	model.h	Note 1
Default, Const, Const--Volatile, Volatile	sep	auto	N/A	N/A	N/A	gbl.c	model.h	Note 1
Default, Const, Const--Volatile, Volatile	auto	src	N/A	N/A	N/A	model.c	src.c	Note 8
Default, Const, Const--Volatile, Volatile	src	src	N/A	N/A	N/A	src.c	src.c	Note 8
Default, Const, Const--Volatile, Volatile	sep	src	N/A	N/A	N/A	gbl.c	src.c	Note 8
Default, Const, Const--Volatile, Volatile	auto	sep	N/A	N/A	N/A	model.c	gbl.h	Note 9
Default, Const, Const--Volatile, Volatile	src	sep	N/A	N/A	N/A	src.c	gbl.h	Note 9

Storage Class Setting	Global Settings:		Override Settings for Specific Data Object:			Results in Generated Files:		
	Data Def.	Data Dec.	Def. File	Owner	Header File	Where Data Def. Is	Where Data Dec. Is	Dec. Inclusion
Default, Const, Const--Volatile, Volatile	sep	sep	N/A	N/A	N/A	gbl.c	gbl.h	Note 9
<i>Simulink Custom Storage Class: Exported Data:</i>								
ExportTo-File	auto	auto	N/A	N/A	null	model.c	model.h	Note 1
ExportTo-File	src	auto	N/A	N/A	null	src.c	model.h	Note 1
ExportTo-File	sep	auto	N/A	N/A	null	gbl.c	model.h	Note 1
ExportTo-File	auto	src	N/A	N/A	null	model.c	src.c	Note 8
ExportTo-File	src	src	N/A	N/A	null	src.c	src.c	Note 8
ExportTo-File	sep	src	N/A	N/A	null	gbl.c	src.c	Note 8
ExportTo-File	auto	sep	N/A	N/A	null	model.c	gbl.h	Note 9
ExportTo-File	src	sep	N/A	N/A	null	src.c	gbl.h	Note 9
ExportTo-File	sep	sep	N/A	N/A	null	gbl.c	gbl.h	Note 9
ExportTo-File	auto	D/C	N/A	N/A	hdr.h	model.c	hdr.h	Note 11
ExportTo-File	src	D/C	N/A	N/A	hdr.h	src.c	hdr.h	Note 11
ExportTo-File	sep	D/C	N/A	N/A	hdr.h	gbl.c	hdr.h	Note 11

Notes

In the previous table:

- A Declaration Inclusion Approach is a file in which the header file that contains the data declarations is included.
- D/C stands for don't care.
- Dec stands for declaration.
- Def stands for definition.
- gbl stands for global.

- `hdr` stands for header.
- `N/A` stands for not applicable.
- `null` stands for field is blank.
- `sep` stands for separate.

Note 1: `model.h` is included directly in all source files.

Note 2: `model_private.h` is included directly in all source files.

Note 3: `extern` is included in `model_private.h`, which is in `source.c`.

Note 4: `header.h` is included in `model_private.h`, which is in `source.c`.

Note 5: `model.h` is included directly in all source files that use `#define`.

Note 6: `header.h` is included in `model.h`, which is in source files that use `#define`.

Note 7: `model.h` is included in all `source.c` files.

Note 8: `extern` is inlined in source files where data is used.

Note 9: `global.h` is included in `model.h`, which is in all source files.

Note 10: When you specify a definition filename for a data object, a header file is not generated for that data object. The code generator declares the data object according to the data placement priorities.

Note 11: `header.h` is included in `model.h`, which is in all source files.

Note 12: Signal: Either not defined because it is expression folded, or local data, or defined in a structure in `model.c`, all depending on model's code generation settings. Parameter: Either inlined in the code, or defined in `model_data.c`.

Note 13: Signal: In a structure that is defined in `model.c`. Parameter: In a structure that is defined in `model_data.c`.

Specify Delimiter for #Includes

Understanding the purpose of this procedure requires understanding the `Header file` property of a data object, described in “Parameter and Signal Property Values”, and applied in “Create mpt Data Objects with Data Object Wizard”. For a particular data object, you can specify as the `Header file` property value a `.h` filename where that data object will be declared. Then, in the `IncludeFile` section of the generated file, this `.h` file is indicated in a `#include` preprocessor directive.

Further, when specifying the filename as the `Header file` property value, you may or may not place it within the double-quote or angle-bracket delimiter. That is, you can specify it as `filename.h`, `"filename.h"`, or `<filename.h>`. The code generator finds every data object for which you specified a filename as its `Header file` property value *without* a delimiter. By default, it assigns to each of these the double-quote delimiter.

This procedure allows you to specify the angle-bracket delimiter for these instead of the default double-quote delimiter. See the figure below.

- 1 In the **#include file delimiter** field on the **Code Placement** pane of the Configuration Parameters dialog box, select `#include <header.h>` instead of the default `#include "header.h"`.
- 2 Click **Apply**.

Source Code Generation

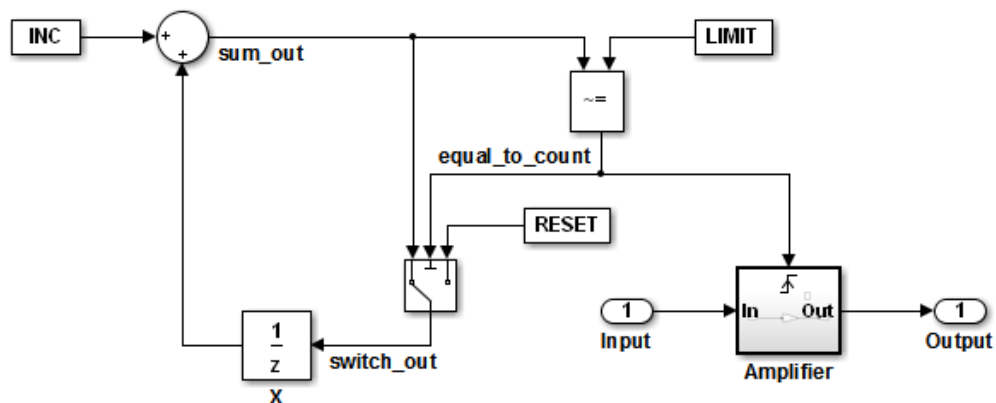
- “Generating Code Using Embedded Coder®” on page 15-2
- “Generate Code Modules” on page 15-9
- “Generate Reentrant Code from MATLAB Code” on page 15-15
- “Generate Reentrant Code from Top-Level Models” on page 15-33

Generating Code Using Embedded Coder®

This example shows how to select a target for a Simulink® model, configure options, generate C code for embedded systems, and view generated files.

1. Open the model.

```
model='rtwdemo_rtwecintro';
open_system(model)
```



Algorithm Description

An 8-bit counter feeds a triggered subsystem parameterized by constants INC, LIMIT, and RESET. The I/O for the model is Input and Output. The Amplifier subsystem amplifies the input signal by gain factor K, which is updated whenever signal equal_to_count is true.



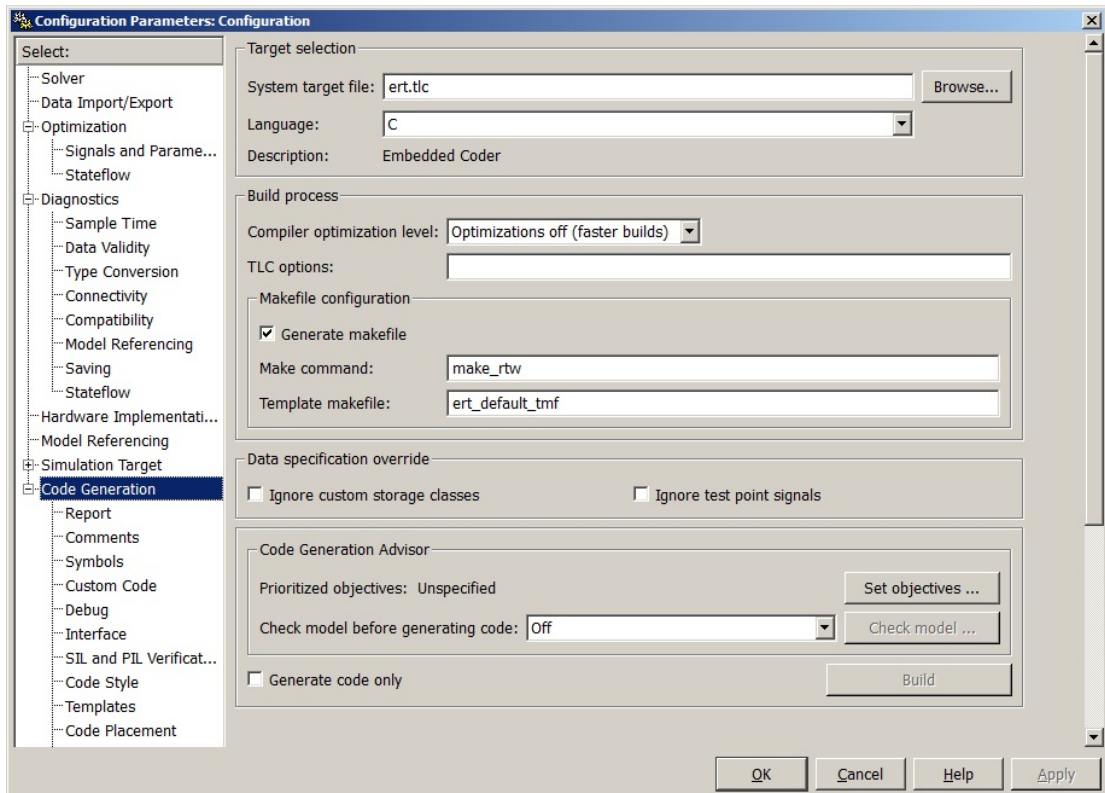
Copyright 1994-2012 The MathWorks, Inc.

2. Open the **Configuration Parameters** dialog box from the model editor by clicking **Simulation > Model Configuration Parameters**.

Alternately, type the following commands at the MATLAB® command prompt.

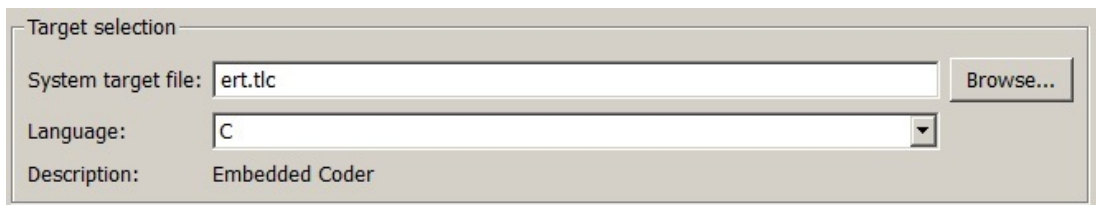
```
cs = getActiveConfigSet(model);
openDialog(cs);
```

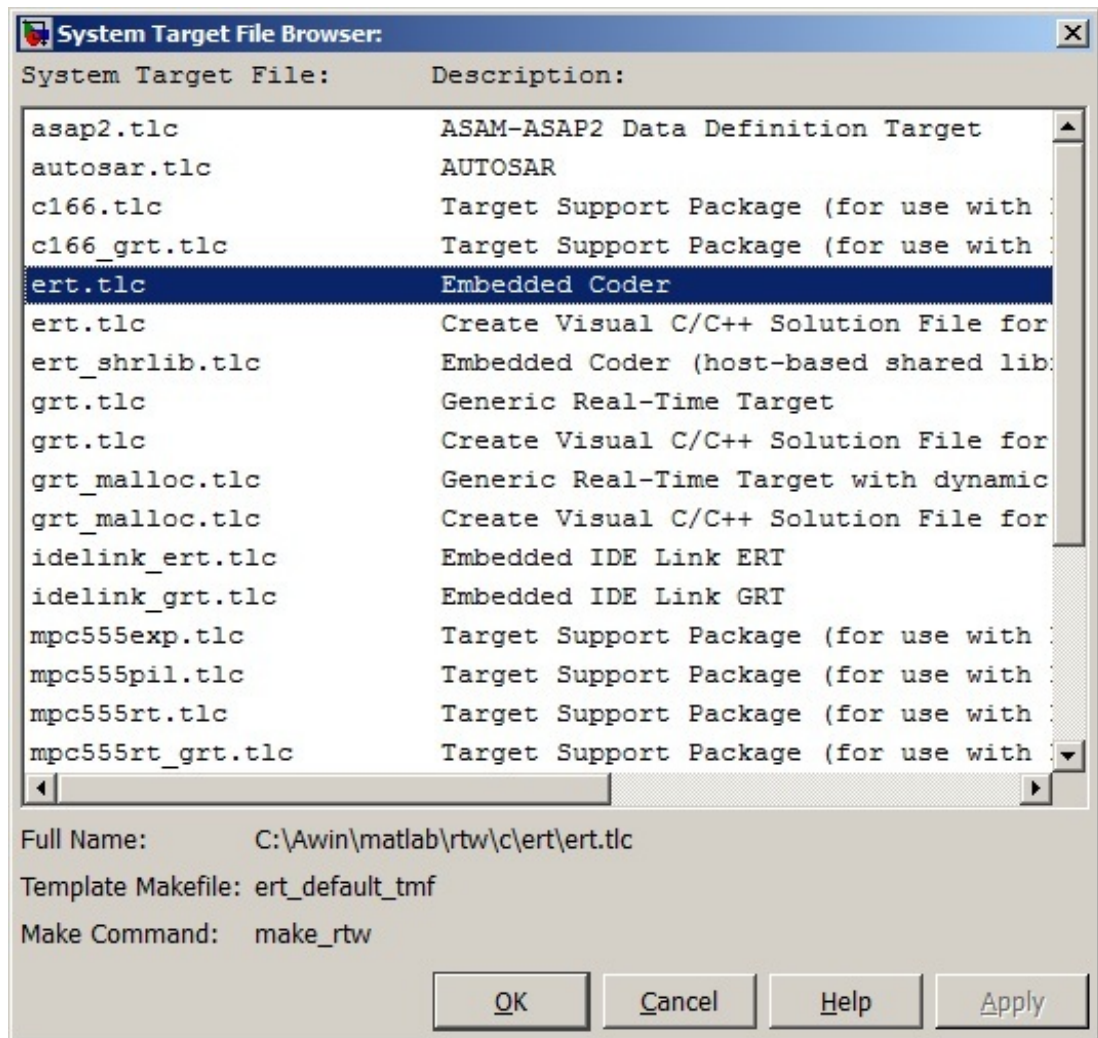
3. Select the **Code Generation** node.



4. In the **Target Selection** pane, click **Browse** to select a target.

You can generate code for a particular target environment or purpose. Some built-in targeting options are provided using system target files, which control the code generation process for a target.



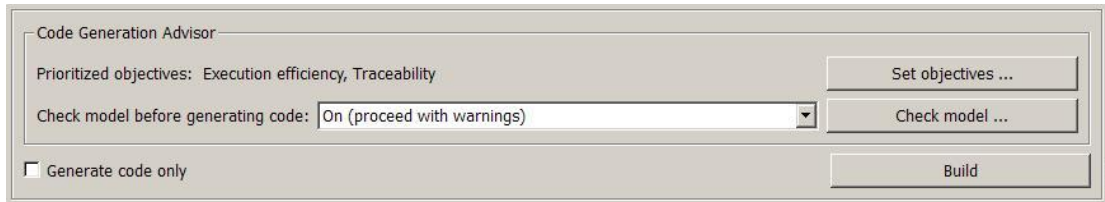


5. Select the **Embedded Real-Time (ERT)** target and click **Apply**.

The ERT target includes a utility to specify and prioritize code generation settings based on your application objectives.

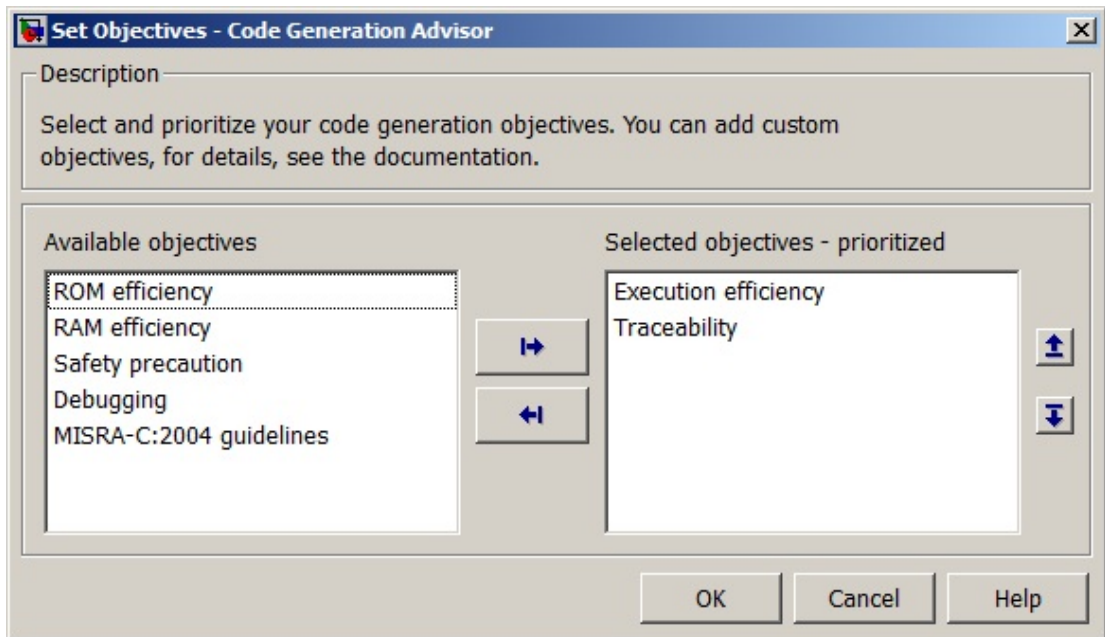
6. In the **Code Generation Advisor** pane, click **Set Objectives**.

You can set and prioritize objectives for the generated code. For example, while code traceability might be a very important criterion for your application, you might not want to prioritize it at the cost of code execution efficiency.



7. In the **Set Objectives** pane, select **Execution efficiency** and **Traceability**. Click **OK**.

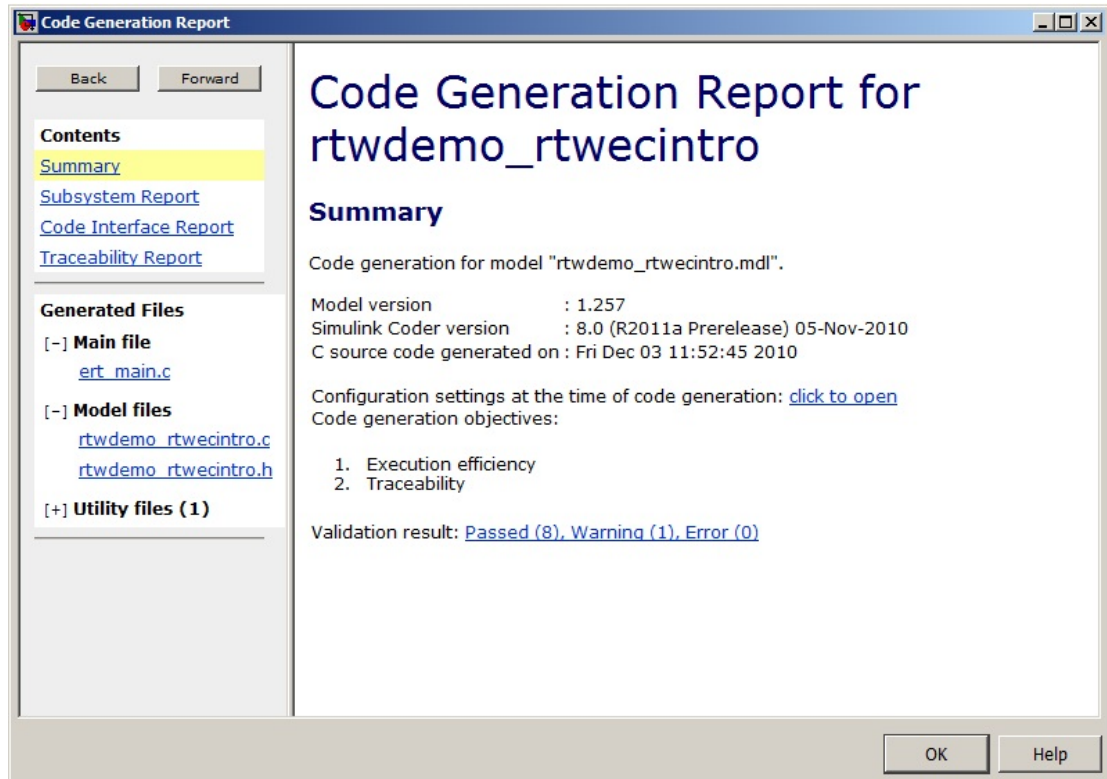
You can select and prioritize a combination of objectives before generating code.



8. In the **Code Generation** pane, click **Build** to generate code.

9. View the code generation report that appears.

The report includes `rtwdemo_rtweintro.c`, associated utility and header files, and traceability and validation reports.



The figure below contains a portion of `rtwdemo_rtweintro.c`

```

1  /* Model step function */
2  void rtwdemo_rtweintro_step(void)
3  {
4      boolean_T rtb_equal_to_count;
5
6      /* Sum: '<Root>/Sum' incorporates:
7       * Constant: '<Root>/INC'
8       * UnitDelay: '<Root>/X'
9       */
10     rtDWork.X = (uint8_T)(1U + (uint32_T)rtDWork.X);
11
12     /* RelationalOperator: '<Root>/RelOpt' incorporates:
13      * Constant: '<Root>/LIMIT'
14      */
15     rtb_equal_to_count = (rtDWork.X != 16);
16
17     /* Outputs for Triggered SubSystem: '<Root>/Amplifier' incorporates:
18      * TriggerPort: '<S1>/Trigger'
19      */
20     if (rtb_equal_to_count && (rtPrevZCSigState.Amplifier_Trig_ZCE != POS_ZCSIG))
21     {
22         /* Output: '<Root>/Output' incorporates:
23          * Gain: '<S1>/Gain'
24          * Inport: '<Root>/Input'
25          */
26         rtY.Output = rtU.Input << 1;
27     }
28
29     rtPrevZCSigState.Amplifier_Trig_ZCE = (uint8_T)(rtb_equal_to_count ? (int32_T)
30     POS_ZCSIG : (int32_T)ZERO_ZCSIG);
31
32     /* End of Outputs for SubSystem: '<Root>/Amplifier' */
33
34     /* Switch: '<Root>/Switch' */
35     if (!rtb_equal_to_count) {
36         /* Update for UnitDelay: '<Root>/X' incorporates:
37          * Constant: '<Root>/RESET'
38          */
39         rtDWork.X = 0U;
40     }
41
42     /* End of Switch: '<Root>/Switch' */
43 }

```

10. Close the model.

```
bdclose(model)
rtwdemoclean;
```

Generate Code Modules

In this section...

“Introduction” on page 15-9

“Generated Code Modules” on page 15-9

“User-Written Code Modules” on page 15-12

“Customize Generated Code Modules” on page 15-12

Introduction

This section summarizes the code modules and header files that make up a Embedded Coder program and describes where to find the code modules and header files.

The easiest way to locate and examine the generated code files is to use the HTML code generation report. The code generation report provides a table of hyperlinks that you click to view the generated code in the MATLAB Help browser. For more information, see “Traceability in Code Generation Report”.

Generated Code Modules

The Embedded Coder software creates a build folder in your working folder to store generated source code. The build folder also contains object files, a makefile, and other files created during the code generation process. The default name of the build folder is *model_ert_rtw*.

Embedded Coder File Packaging summarizes the structure of source code generated by the Embedded Coder software.

Embedded Coder File Packaging

File	Description
<i>model.c</i> or <i>.cpp</i>	Contains entry points for code implementing the model algorithm (for example, <i>model_step</i> , <i>model_initialize</i> , and <i>model_terminate</i>).
<i>model_private.h</i>	Contains local macros and local data that are required by the model and subsystems. This file is included in the <i>model.c</i> file as a <code>#include</code> statement. You do not need to include <i>model_private.h</i> when interfacing handwritten code to the generated code of a model.

File	Description
<i>model.h</i>	Declares model data structures and a public interface to the model entry points and data structures. Also provides an interface to the real-time model data structure (<i>model_M</i>) with accessor macros. <i>model.h</i> is included in the subsystem <i>.c</i> or <i>.cpp</i> files of the model. If you are interfacing your handwritten code to generated code for one or more models, include <i>model.h</i> for each of those models.
<i>model_data.c</i> or <i>.cpp</i> (conditional)	<i>model_data.c</i> or <i>.cpp</i> is conditionally generated. It contains the declarations for the parameters data structure, the constant block I/O data structure, and any zero representations for the model structure data types. If these data structures and zero representations are not used in the model, <i>model_data.c</i> or <i>.cpp</i> is not generated. These structures and zero representations are declared <code>extern</code> in <i>model.h</i> .
<i>model_types.h</i>	Provides forward declarations for the real-time model data structure and the parameters data structure. Function declarations of reusable functions might need these declarations. Also provides type definitions for user-defined types used by the model.
<i>rtwtypes.h</i>	Defines data types, structures, and macros required by Embedded Coder generated code. Most other generated code modules also require these definitions. For more information, see “ <i>rtwtypes.h</i> and Shared Utility Code”.
<i>multiword_types.h</i>	Contains type definitions for wide data types and their chunks. File is generated when multiword data types are used or when you select one or more of the following in the Configuration Parameters dialog box on the Code Generation > Interface pane: <ul style="list-style-type: none"> • MAT-file logging • <code>External mode</code> from the Interface list
<i>model_reference_types.h</i>	Contains type definitions for timing bridges. File is generated for a model reference target or a model containing model reference blocks.
<i>builtin_typeid_types.h</i>	Defines an enumerated type corresponding to built-in data types. File is generated when you select one or more of the following in the Configuration Parameters dialog box on the Code Generation > Interface pane:

File	Description
	<ul style="list-style-type: none"> • MAT-file logging • C API from the Interface list
ert_main.c or .cpp (optional)	If the Generate an example main program option is on, this file is generated. (This option is on by default.) See “Generate an example main program”.
rtmodel.h (optional)	<p>If the Generate an example main program option is off, this file is generated. (See “Generate an example main program”.)</p> <p>rtmodel.h contains <code>#include</code> directives required by the <code>rt_main.c</code> or <code>rt_cppclass_main.cpp</code> static main program module. Because the static main program module is not created at code generation time, it includes <code>rtmodel.h</code> to access model-specific data structures and entry points.</p> <p>For more information, see “Static Main Program Module”.</p>
model_capi.c or .cpp model_capi.h (optional)	Provides data structures that enable a running program to access model signals, states, and parameters without external mode. To learn how to generate and use the <code>model_capi.c</code> or <code>.cpp</code> and <code>.h</code> files, see “Data Interchange Using the C API” in the Simulink Coder documentation.

You can customize the generated set of files in several ways:

- **File packaging formats:** Specify the number of source files generated for your model. In the Configuration Parameter dialog box, on the **Code Generation > Code Placement** pane, specify the **File packaging format** parameter. For more information, see “Customize Generated Code Modules” on page 15-12.
- **Nonvirtual subsystem code generation:** Instruct the code generation software to generate separate functions, within separate code files, for nonvirtual subsystems. You can control the names of the functions and of the code files. For further information, see “Code Generation of Subsystems”.
- **Custom storage classes:** Use custom storage classes to partition generated data structures into different files based on file names that you specify. For further information, see “Introduction to Custom Storage Classes”.

- **Module Packaging Features (MPF):** Direct the generated code into a required set of `.c` or `.cpp` and `.h` files, and control the internal organization of the generated files. For details, see “Data, Function, and File Definition”.

User-Written Code Modules

Code that you write to interface with generated model code usually includes a customized main module (based on a main program provided by the code generation software), and may also include interrupt handlers, device driver blocks and other S-functions, and other supervisory or supporting code.

Establish a working folder for your own code modules. Put your working folder on the MATLAB path. Minimally, you must also modify the ERT template makefile and system target file so that the build process can find your source and object files. If you want to generate code for a particular microprocessor or development board and deploy the code on target hardware with a cross-development system, make more extensive modifications to the ERT target files.

For information on how to customize the ERT target for your production requirements, see “Target Development”.

Customize Generated Code Modules

Embedded Coder software provides a configuration parameter to specify how the generated source code is packaged into files. The configuration parameter “File packaging format” drop-down list options are located in the Configuration Parameter dialog box, on the **Code Generation > Code Placement** pane, in the Code Packaging section. The options are: **Modular**, **Compact (with separate data file)**, and **Compact**. Generated Files According to File Packaging Format shows the files generated for each file packaging format and the files that have been removed.

Generated Files According to File Packaging Format

File Packaging Format	Generated Files	Removed Files
Modular (default)	<i>model.c</i> subsystem files (optional) <i>model.h</i> <i>model_types.h</i>	None

File Packaging Format	Generated Files	Removed Files
	<i>model_private.h</i> <i>model_data.c</i> (conditional)	
Compact (with separate data file)	<i>model.c</i> <i>model.h</i> <i>model_data.c</i> (conditional)	<i>model_private.h</i> <i>model_types.h</i> (conditional, see below)
Compact	<i>model.c</i> <i>model.h</i>	<i>model_data.c</i> <i>model_private.h</i> <i>model_types.h</i> (conditional, see below)

The code generation process places the content of the removed files as follows:

Removed File	Generated Content In File
<i>model_private.h</i>	<i>model.c</i> and <i>model.h</i>
<i>model_types.h</i>	<i>model.h</i>
<i>model_data.c</i>	<i>model.c</i>

You can specify a different file packaging format for each referenced model.

If you specify **Shared code placement** as **Shared location** on the **Code Generation > Interface** pane of the Configuration Parameter dialog box, the code generation process generates separate files for utility code in a shared location, regardless of the file packaging format. If you specify the **Shared code placement** as **Auto**, the generated code for utilities is dependent on the file packaging format as follows:

- **Modular:** Some shared utility files are in the build directory
- **Compact (with separate data file):** Utility code is generated in *model.c*
- **Compact:** Utility code is generated in *model.c*

File packaging formats **Compact** and **Compact (with separate data file)** generate *model_types.h* for models containing:

- A Model Variants block or a Variant Subsystem block. The `model_types.h` file includes preprocessor directives defining the variant objects associated with a variant block.
- Custom storage classes specifying a separate header file. The `model_types.h` file includes the `#include` call to the external header file.

File packaging formats `Compact` and `Compact (with separate data file)` are not compatible with the following:

- A model containing a subsystem, which is configured to generate separate source files
- A model containing a noninlined S-function
- A model for which **Shared code placement** is set to `Auto`, which uses data objects for which **Data scope** is set to `Exported`

Generate Reentrant Code from MATLAB Code

In this section...

“What Is Reentrant Code?” on page 15-15

“When to Generate Reentrant Code” on page 15-15

“Generate Reentrant Code” on page 15-16

“Generated Code API” on page 15-16

“Call Reentrant Code in a Single-Thread Environment” on page 15-17

“Call Reentrant Code in a Multithreaded Environment” on page 15-18

“Call Reentrant Code with No Persistent or Global Data (UNIX Only)” on page 15-19

“Call Reentrant Code — Multithreaded with Persistent Data (Windows Only)” on page 15-23

“Call Reentrant Code — Multithreaded with Persistent Data (UNIX Only)” on page 15-27

What Is Reentrant Code?

Reentrant code is a reusable programming routine that multiple programs can use simultaneously. Operating systems and other system software that uses multithreading to handle concurrent events use reentrant code. Sharing code with persistent or static data in a concurrent environment is difficult because multiple threads or processes might attempt to simultaneously read and write the static data. Reentrant code does not contain static data. Calling programs maintain their state variables and pass them into the function. Therefore, any number of threads or processes can share one copy of a reentrant routine.

With an Embedded Coder license, you can use `codegen` to generate reusable code. For more information, see “Generate Reentrant Code” on page 15-16.

When to Generate Reentrant Code

Generate reentrant code when you want to:

- Deploy your code in multi-threaded environments.
- Share the same algorithm with different persistent data.
- Compile code that uses function variables that are too large to fit on the stack.

If you do not choose to generate reentrant code, `codegen` generates code that uses statically allocated memory for function variables that are too large to fit on the stack, and for global and persistent variables. The use of static memory allocation for these variables means that you cannot deploy the generated code in environments that require code to be reentrant. In addition, the generated code can result in static memory size overflow if you cannot adjust the static memory allocation size to accommodate the static memory requirements of the program.

When you generate reentrant code, `codegen` creates input data structures for function variables that are too large to fit on the stack, and for persistent and global variables. You can then dynamically allocate memory for these input structures. The use of dynamic memory allocation means that you can deploy the code in reentrant environments.

Generate Reentrant Code

Prerequisites

This option requires an Embedded Coder license.

Procedure

Use the `MultiInstanceCode` option of the `coder.EmbeddedCodeConfig` code generation configuration object. For example, to compile the file `foo.m` and generate reusable code:

- 1 Create a code generation configuration object and enable the `MultiInstanceCode` option.

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

- 2 Pass the configuration object to `codegen` using the `-config` option.

```
codegen -config cfg foo
```

Alternatively, you can set this parameter using the MATLAB Coder Project Settings dialog box. On the **Interface** pane, select **Reusable function** code interface packaging.

Generated Code API

When you generate reusable code, `codegen` supports dynamic allocation of function variables that are too large for the stack, as well as persistent and global variables. It

generates a header file, *primary_function_name_types.h*, which you must include when using the generated code. This header file contains the following structures:

- *primary_function_nameStackData*

This structure contains the user allocated memory. You must pass a pointer to this structure as the first parameter to all functions that use it either directly, because the function uses a field in the structure, or indirectly, because the function passes the structure to a called function.

The *primary_function_nameStackData* structure also contains a pointer to the *primary_function_namePersistentData* structure if the algorithm uses persistent or global data. Including this pointer means that you have to pass only one parameter to each calling function.

- *primary_function_namePersistentData*

If your algorithm uses persistent or global variables, `codegen` provides a separate structure for them and adds a pointer to this structure to the memory allocation structure. Having a separate structure for persistent and global variables allows you to allocate memory for these variables once and share them with all threads. However, if there is no communication between threads, you can choose to allocate memory for these variables per thread or per application.

For more information on using these global structures, see “Multithreaded Examples” on page 15-18.

Call Reentrant Code in a Single-Thread Environment

To call reentrant code in a single-thread environment, create a `main` function that:

- Includes the header file *primary_function_name.h*.
- Allocates memory for the global memory allocation structure *primary_function_nameStackData*.
- If the algorithm uses persistent or global data, allocates memory for the global structure *primary_function_namePersistentData*, .
- Calls these functions:
 - *primary_function_name_initialize*.
 - *primary_function_name*.

- *primary_function_name_terminate*.

When you convert a MATLAB function to a C/C++ library function or a C/C++ executable, `codegen` automatically generates two housekeeping functions that you must call with the C/C++ function. For more information, see “Calling Initialize and Terminate Functions”.

- Frees the memory used for global structures.

Call Reentrant Code in a Multithreaded Environment

To call reentrant code, create a `main` function that:

- Includes the header file *primary_function_name.h*.
- For each thread, allocates memory for the global memory allocation structure *primary_function_nameStackData*.
- If the algorithm uses persistent or global data, allocates memory for the global structure *primary_function_namePersistentData*. If there is communication between threads, you must allocate this memory once for the application. Otherwise, you can choose to allocate memory per thread or per application.
- Contains a thread function that calls these functions:
 - *primary_function_name_initialize*.
 - *primary_function_name*.
 - *primary_function_name_terminate*.

When you convert a MATLAB function to a C/C++ library function or a C/C++ executable, `codegen` automatically generates two housekeeping functions that you must call with the C/C++ function. For more information, see “Calling Initialize and Terminate Functions” in the MATLAB Coder documentation.

- Initializes each thread and passes in a pointer to the memory allocation structure as the first parameter to the thread function.
- Frees the memory used for global structures.

Multithreaded Examples

Type of Reentrant Code	Platform	Reference
Multithreaded without persistent or global data	Windows	“Generating Reentrant C Code from MATLAB Code”

Type of Reentrant Code	Platform	Reference
	UNIX	“Call Reentrant Code with No Persistent or Global Data (UNIX Only)” on page 15-19
Multithreaded with persistent or global data	Windows	“Call Reentrant Code — Multithreaded with Persistent Data (Windows Only)” on page 15-23
	UNIX	“Call Reentrant Code — Multithreaded with Persistent Data (UNIX Only)” on page 15-27

Call Reentrant Code with No Persistent or Global Data (UNIX Only)

This example requires POSIX thread (pthread) libraries and, therefore, runs only on UNIX platforms. It is a simple multithreaded example that uses no persistent or global data. Two threads call the MATLAB function `matrix_exp` with different sets of input data.

MATLAB Code Used for This Example

```
function Y = matrix_exp(X) %#codegen
%
% The function matrix_exp computes matrix exponential
% of the input matrix using Taylor series and returns
% the computed output.
%
E = zeros(size(X));
F = eye(size(X));
k = 1;
while norm(E+F-E,1) > 0
    E = E + F;
    F = X*F/k;
    k = k+1;
end
Y = E;
```

Provide a Main Function

To call the reentrant code, you must provide a `main` function that:

- Includes the header file `matrix_exp.h`.
- For each thread, allocates memory for stack data.
- Calls the `matrix_exp_initialize` housekeeping function. For more information, see “Calling Initialize and Terminate Functions”.
- Calls `matrix_exp`.
- Calls `matrix_exp_terminate`.

- Frees the memory used for stack data.

For this example, `main.c` contains:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "matrix_exp.h"
#include "matrix_exp_initialize.h"
#include "matrix_exp_terminate.h"
#include "rtwtypes.h"
#define NUMELEMENTS (160*160)

typedef struct {
    real_T in[NUMELEMENTS];
    real_T out[NUMELEMENTS];
    matrix_expStackData* spillData;
} IODATA;

/* The thread_function calls the matrix_exp function written in MATLAB */
void *thread_function(void *dummyPtr) {
    IODATA *myIOData = (IODATA*)dummyPtr;
    matrix_exp_initialize();
    matrix_exp(myIOData->spillData, myIOData->in, myIOData->out);
    matrix_exp_terminate();
}

int main() {
    pthread_t thread1, thread2;
    int iret1, iret2;
    IODATA data1;
    IODATA data2;
    int32_T i;

    /*Initializing data for passing to the 2 threads*/
    matrix_expStackData* sd1=(matrix_expStackData*)calloc(1,sizeof(matrix_expStackData));
    matrix_expStackData* sd2=(matrix_expStackData*)calloc(1,sizeof(matrix_expStackData));

    data1.spillData = sd1;
    data2.spillData = sd2;

    for (i=0;i<NUMELEMENTS;i++) {
        data1.in[i] = 1;
        data1.out[i] = 0;
        data2.in[i] = 1.1;
        data2.out[i] = 0;
    }

    /*Initializing the 2 threads and passing required data to the thread functions*/
    printf("Starting thread 1...\n");
    iret1 = pthread_create(&thread1, NULL, thread_function, (void*) &data1);
    if (iret1 != 0){
        perror( "Thread 1 creation failed.");
        exit(EXIT_FAILURE);
    }
}
```



```

}

printf("Starting thread 2...\n");
iret2 = pthread_create(&thread2, NULL, thread_function, (void*) &data2);
if (iret2 != 0){
    perror( "Thread 2 creation failed.");
    exit(EXIT_FAILURE);
}

/*Wait for both the threads to finish execution*/
iret1 = pthread_join(thread1, NULL);
if (iret1 != 0){
    perror( "Thread 1 join failed.");
    exit(EXIT_FAILURE);
}

iret2 = pthread_join(thread2, NULL);
if (iret2 != 0){
    perror( "Thread 2 join failed.");
    exit(EXIT_FAILURE);
}

free(sd1);
free(sd2);

printf("Finished Execution!\n");
exit(EXIT_SUCCESS);
}

```

Generate Reentrant C Code

Run the following script at the MATLAB command line to generate code.

```

% This example can only be run on Unix platforms
if ~isunix
    error('This example requires pthread libraries and can only be run on Unix.');
```

```

end

% Setting the options for the Config object

% Create a code gen configuration object
e = coder.config('exe','ecoder', true);

% Enable reentrant code generation
e.MultiInstanceCode = true;

% Set the post code generation command to be the 'setbuildargs' function
e.PostCodeGenCommand = 'setbuildargs(buildInfo)';

% Compiling
codegen -config e main.c matrix_exp.m -report -args ones(160,160)

```

These commands:

- Check that the example is running on UNIX platforms and generates an error message if not.
- Create a Simulink Coder configuration object for an ERT target.
- Enable the `MultiInstanceCode` option to generate reusable, reentrant code.
- Use the `PostCodeGenCommand` option to set the post-code-generation command to be the `setbuildargs` function. This function sets the `-lpthread` flag to specify that the build include the pthread library.

```
function setbuildargs(buildInfo)
% The example being compiled requires pthread support.
% The -lpthread flag requests that the pthread library
% be included in the build
    linkFlags = {'-lpthread'};
    addLinkFlags(buildInfo, linkFlags);
```

For more information about the `PostCodeGenCommand` option, see “Customize the Post-Code-Generation Build Process”.

- Invoke `codegen` with the following options:
 - `-config` to pass in the code generation configuration object `e`.
 - `main.c` to include this file in the compilation.
 - `-report` to create a code generation report.
 - `-args` to specify an example input with class, size, and complexity.

For more information on these options, see `codegen`.

Examine the Generated Code

`codegen` generates a header file `matrix_exp_types.h`, which defines the `matrix_expStackData` global structure. This structure contains local variables that are too large to fit on the stack.

```
/*
 * matrix_exp_types.h
 *
 * MATLAB Coder code generation for function 'matrix_exp'
 */

#ifndef __MATRIX_EXP_TYPES_H__
#define __MATRIX_EXP_TYPES_H__

/* Type Definitions */
typedef struct {
    struct {
        real_T F[25600];
        real_T Y[25600];
    };
};
```

```

    } f0;
} matrix_expStackData;
#endif
/* End of MATLAB Coder code generation (matrix_exp_types.h) */

```

Run the Code

Finally, call the code using the command:

```
system('./matrix_exp')
```

The executable runs and reports completion.

Call Reentrant Code — Multithreaded with Persistent Data (Windows Only)

This example requires libraries that are specific to the Microsoft Windows operating system and, therefore, runs only on Windows platforms. It is a multithreaded example that uses persistent data. Two threads call the MATLAB function `matrix_exp` with different sets of input data.

MATLAB Code Used for This Example

```

function [Y,numTimes] = matrix_exp(X) %#codegen
%
% The function matrix_exp computes matrix exponential
% of the input matrix using Taylor series and returns
% the computed output. It also returns the number of
% times this function has been called.
%
persistent count;
if isempty(count)
    count = 0;
end
count = count+1;

E = zeros(size(X));
F = eye(size(X));
k = 1;
while norm(E+F-E,1) > 0
    E = E + F;
    F = X*F/k;
    k = k+1;
end
Y = E ;

numTimes = count;

```

Provide a Main Function

To call reentrant code that uses persistent data, you must provide a `main` function that:

- Includes the header file `matrix_exp.h`.
- For each thread, allocates memory for stack data.
- Allocates memory for persistent data, once per application if threads share data, and once per thread otherwise.
- Calls the `matrix_exp_initialize` housekeeping function. For more information, see “Calling Initialize and Terminate Functions”.
- Calls `matrix_exp`.
- Calls `matrix_exp_terminate`.
- Frees the memory used for stack and persistent data.

For this example, `main.c` contains:

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include "matrix_exp.h"
#include "matrix_exp_initialize.h"
#include "matrix_exp_terminate.h"
#include "rtwtypes.h"
#define NUMELEMENTS (160*160)

typedef struct {
    real_T in[NUMELEMENTS];
    real_T out[NUMELEMENTS];
    real_T numTimes;
    matrix_expStackData* spillData;
} IODATA;

/*The thread_function calls the matrix_exp function written in MATLAB*/
DWORD WINAPI thread_function(PVOID dummyPtr) {
    IODATA *myIOData = (IODATA*)dummyPtr;
    matrix_exp_initialize(myIOData->spillData);
    matrix_exp(myIOData->spillData, myIOData->in, myIOData->out, &myIOData->numTimes);
    printf("Number of times function matrix_exp is called is %g\n",myIOData->numTimes);
    matrix_exp_terminate();
    return 0;
}

void main() {
    HANDLE thread1, thread2;
    IODATA data1;
    IODATA data2;
    int32_T i;

    /*Initializing data for passing to the 2 threads*/
    matrix_expPersistentData* pd1 = (matrix_expPersistentData*)calloc(1,sizeof(matrix_expPersistentData));
    matrix_expPersistentData* pd2 = (matrix_expPersistentData*)calloc(1,sizeof(matrix_expPersistentData));
    matrix_expStackData* sd1 = (matrix_expStackData*)calloc(1,sizeof(matrix_expStackData));
    matrix_expStackData* sd2 = (matrix_expStackData*)calloc(1,sizeof(matrix_expStackData));
```

```

sd1->pd = pd1;
sd2->pd = pd2;
data1.spillData = sd1;
data2.spillData = sd2;

for (i=0;i<NUMELEMENTS;i++) {
    data1.in[i] = 1;
    data1.out[i] = 0;
    data2.in[i] = 1.1;
    data2.out[i] = 0;
}

data1.numTimes = 0;
data2.numTimes = 0;

/*Initializing the 2 threads and passing required data to the thread functions*/
printf("Starting thread 1...\n");
thread1 = CreateThread(NULL, 0, thread_function, (PVOID) &data1, 0, NULL);
if (thread1 == NULL){
perror( "Thread 1 creation failed.");
exit(EXIT_FAILURE);
}

printf("Starting thread 2...\n");
thread2 = CreateThread(NULL, 0, thread_function, (PVOID) &data2, 0, NULL);
if (thread2 == NULL){
    perror( "Thread 2 creation failed.");
    exit(EXIT_FAILURE);
}

/*Wait for both the threads to finish execution*/
if (WaitForSingleObject(thread1, INFINITE) != WAIT_OBJECT_0){
perror( "Thread 1 join failed.");
exit(EXIT_FAILURE);
}

if (WaitForSingleObject(thread2, INFINITE) != WAIT_OBJECT_0){
perror( "Thread 2 join failed.");
exit(EXIT_FAILURE);
}

free(sd1);
free(sd2);
free(pd1);
free(pd2);

printf("Finished Execution!\n");
exit(EXIT_SUCCESS);
}

```

Generate Reentrant C Code

Run the following script at the MATLAB command line to generate code.

```
% This example can only be run on Windows platforms
if -ispc
    error...
    ('This example requires Windows-specific libraries and can only be run on Windows.');
```

```
end

% Setting the options for the Config object
% Create a code gen configuration object
e = coder.config('exe', 'ecoder', true);

% Enable reentrant code generation
e.MultiInstanceCode = true;

% Compiling
codegen -config e main.c -report matrix_exp.m -args ones(160,160)
```

These commands:

- Check that the example is running on Windows platforms and generates an error message if not.
- Create a code generation configuration object for an ERT target.
- Enable the `MultiInstanceCode` option to generate reusable, reentrant code.
- Invoke `codegen` with the following options:
 - `-config` to pass in the code generation configuration object `e`.
 - `main.c` to include this file in the compilation.
 - `-report` to create a code generation report.
 - `-args` to specify an example input with class, size, and complexity.

For more information on these options, see `codegen`.

Examine the Generated Code

`codegen` generates a header file `matrix_exp_types.h`, which defines:

- The `matrix_expStackData` global structure that contains local variables that are too large to fit on the stack and a pointer to the `matrix_expPersistentData` global structure.
- The `matrix_expPersistentData` global structure that contains persistent data.

```
/*
 * matrix_exp_types.h
 *
 * MATLAB Coder code generation for function 'matrix_exp'
 */
```

```

*/
#ifdef __MATRIX_EXP_TYPES_H__
#define __MATRIX_EXP_TYPES_H__

/* Type Definitions */
typedef struct {
    real_T count;
} matrix_expPersistentData;
typedef struct {
    struct {
        real_T F[25600];
        real_T Y[25600];
    } f0;
    matrix_expPersistentData *pd;
} matrix_expStackData;
#endif
/* End of code generation (matrix_exp_types.h) */

```

Run the Code

Finally, call the code using the command:

```
system('matrix_exp.exe')
```

The executable runs and reports completion.

Call Reentrant Code — Multithreaded with Persistent Data (UNIX Only)

This example requires POSIX thread (pthread) libraries and, therefore, runs only on UNIX platforms. It is a multithreaded example that uses persistent data. Two threads call the MATLAB function `matrix_exp` with different sets of input data.

MATLAB Code Used for This Example

```

function [Y,numTimes] = matrix_exp(X) %#codegen
%
% The function matrix_exp computes matrix exponential
% of the input matrix using Taylor series and returns
% the computed output. It also returns the number of
% times this function has been called.
%

persistent count;
if isempty(count)
    count = 0;
end
count = count+1;

E = zeros(size(X));
F = eye(size(X));
k = 1;
while norm(E+F-E,1) > 0

```

```
E = E + F;
F = X*F/k;
k = k+1;
end
Y = E ;

numTimes = count;
```

Provide a Main Function

To call reentrant code that uses persistent data, you must provide a `main` function that:

- Includes the header file `matrix_exp.h`.
- For each thread, allocates memory for stack data.
- Allocates memory for persistent data, once per application if threads share data, and once per thread otherwise.
- Calls the `matrix_exp_initialize` housekeeping function. For more information, see “Calling Initialize and Terminate Functions”.
- Calls `matrix_exp`.
- Calls `matrix_exp_terminate`.
- Frees the memory used for stack and persistent data.

For this example, `main.c` contains:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "matrix_exp.h"
#include "matrix_exp_initialize.h"
#include "matrix_exp_terminate.h"
#include "rtwtypes.h"
#define NUMELEMENTS (160*160)

typedef struct {
    real_T in[NUMELEMENTS];
    real_T out[NUMELEMENTS];
    real_T numTimes;
    matrix_expStackData* spillData;
} IODATA;

/*The thread_function calls the matrix_exp function written in MATLAB*/
void *thread_function(void *dummyPtr) {
    IODATA *myIOData = (IODATA*)dummyPtr;
    matrix_exp_initialize(myIOData->spillData);
    matrix_exp(myIOData->spillData, myIOData->in, myIOData->out, &myIOData->numTimes);
    printf("Number of times function matrix_exp is called is %g\n",myIOData->numTimes);
    matrix_exp_terminate();
}
```



```
int main() {
    pthread_t thread1, thread2;
    int iret1, iret2;
    IODATA data1;
    IODATA data2;
    int32_T i;

    /*Initializing data for passing to the 2 threads*/
    matrix_expPersistentData* pd1 =
        (matrix_expPersistentData*)calloc(1,sizeof(matrix_expPersistentData));
    matrix_expPersistentData* pd2 =
        (matrix_expPersistentData*)calloc(1,sizeof(matrix_expPersistentData));
    matrix_expStackData* sd1 = (matrix_expStackData*)calloc(1,sizeof(matrix_expStackData));
    matrix_expStackData* sd2 = (matrix_expStackData*)calloc(1,sizeof(matrix_expStackData));

    sd1->pd = pd1;
    sd2->pd = pd2;
    data1.spillData = sd1;
    data2.spillData = sd2;

    for (i=0;i<NUMELEMENTS;i++) {
        data1.in[i] = 1;
        data1.out[i] = 0;
        data2.in[i] = 1.1;
        data2.out[i] = 0;
    }

    data1.numTimes = 0;
    data2.numTimes = 0;

    /*Initializing the 2 threads and passing required data to the thread functions*/
    printf("Starting thread 1...\n");
    iret1 = pthread_create(&thread1, NULL, thread_function, (void*) &data1);
    if (iret1 != 0){
        perror("Thread 1 creation failed.");
        exit(EXIT_FAILURE);
    }

    printf("Starting thread 2...\n");
    iret2 = pthread_create(&thread2, NULL, thread_function, (void*) &data2);
    if (iret2 != 0){
        perror("Thread 2 creation failed.");
        exit(EXIT_FAILURE);
    }

    /*Wait for both the threads to finish execution*/
    iret1 = pthread_join(thread1, NULL);
    if (iret1 != 0){
        perror("Thread 1 join failed.");
        exit(EXIT_FAILURE);
    }

    iret2 = pthread_join(thread2, NULL);
    if (iret2 != 0){
```

```
    perror( "Thread 2 join failed.");
    exit(EXIT_FAILURE);
}

free(sd1);
free(sd2);
free(pd1);
free(pd2);

printf("Finished Execution!\n");
return(0);
}
```

Generate Reentrant C Code

Run the following script at the MATLAB command line to generate code.

```
% This example can only be run on Unix platforms
if ~isunix
    error('This example requires pthread libraries and can only be run on Unix.');
```

end

```
% Setting the options for the Config object

% Specify an ERT target
e = coder.config('exe','ecoder', true);

% Enable reentrant code generation
e.MultiInstanceCode = true;

% Set the post code generation command to be the 'setbuildargs' function
e.PostCodeGenCommand = 'setbuildargs(buildInfo)';

% Compiling
codegen -config e main.c -report matrix_exp.m -args ones(160,160)
```

These commands:

- Check that the example is running on UNIX platforms and generates an error message if not.
- Create a code generation configuration object.
- Enable the `MultiInstanceCode` option to generate reusable, reentrant code.
- Use the `PostCodeGenCommand` option to set the post-code-generation command to be the `setbuildargs` function. This function sets the `-lpthread` flag to specify that the build include the pthread library.

```
function setbuildargs(buildInfo)
% The example being compiled requires pthread support.
```

```
% The -lpthread flag requests that the pthread library
% be included in the build
    linkFlags = {'-lpthread'};
    addLinkFlags(buildInfo, linkFlags);
```

For more information about the `PostCodeGenCommand` option, see “Customize the Post-Code-Generation Build Process”.

- Invokes `codegen` with the following options:
 - `-config` to pass in the code generation configuration object `e`.
 - `main.c` to include this file in the compilation.
 - `-report` to create a code generation report.
 - `-args` to specify an example input with class, size, and complexity.

For more information on these options, see `codegen`.

Examine the Generated Code

`codegen` generates a header file `matrix_exp_types.h`, which defines:

- The `matrix_expStackData` global structure that contains local variables that are too large to fit on the stack and a pointer to the `matrix_expPersistentData` global structure.
- The `matrix_expPersistentData` global structure that contains persistent data.

```
/*
 * matrix_exp_types.h
 *
 * MATLAB Coder code generation for function 'matrix_exp'
 *
 */

#ifndef __MATRIX_EXP_TYPES_H__
#define __MATRIX_EXP_TYPES_H__

/* Type Definitions */
typedef struct {
    real_T count;
} matrix_expPersistentData;
typedef struct {
    struct {
        real_T F[25600];
        real_T Y[25600];
    } f0;
    matrix_expPersistentData *pd;
} matrix_expStackData;
#endif
/* End of code generation (matrix_exp_types.h) */
```

Run the Code

Finally, call the code using the command:

```
system('./matrix_exp')
```

The executable runs and reports completion.

Generate Reentrant Code from Top-Level Models

To generate reentrant multi-instance code from a model, select **Reusable function** code interface packaging. When you select the **Reusable function** code interface for an ERT-based model:

- By default, the generated *model.c* source file does not contain an allocation function that dynamically allocates model data for each instance of the model. Use the **Use dynamic memory allocation for model initialization** option to control whether an allocation function is generated.
- The generated code passes the real-time model data structure in, by reference, as an argument to *model_step* and the other model entry point functions.
- The real-time model data structure is exported with the *model.h* header file.
- By default, root-level input and output arguments are passed to the reusable model entry-point functions as individual arguments. Use the **Pass root-level I/O as** parameter to control whether root-level input and output arguments are included in the real-time model data structure that is passed to the functions, passed as individual arguments, or passed as references to an input structure and an output structure.

To configure an ERT-based model to generate reusable, reentrant code:

- 1 In the **Code Generation > Interface** pane of the Configuration Parameters dialog box, set “**Code interface packaging**” to the value **Reusable function**. This action enables the parameters **Multi-instance code error diagnostic**, **Pass root-level I/O as**, and **Use dynamic memory allocation for model initialization**.
- 2 Examine the setting of “**Multi-instance code error diagnostic**”. Leave the parameter at its default value **ERROR** unless you have a specific need to alter the severity level for diagnostics displayed when a model violates requirements for generating multi-instance code.
- 3 Configure “**Pass root-level I/O as**” to control how root-level model input and output are passed to *model_step* and the other generated model entry-point functions.

When you set **Code interface packaging** to **Reusable function**, model data (such as block I/O, DWork, and parameters) is packaged into the real-time model data structure, and the model structure is passed to the model entry-point functions. If you set **Pass root-level I/O as** to **Part of model data structure**, the root-level model input and output also are packaged into the real-time model data structure.

- 4 If you want the generated model code to contain a function that dynamically allocates memory for model instance data, select the option “**Use dynamic memory allocation for model initialization**”. If you do not select this option, the generated code statically allocates memory for model data structures.
- 5 Generate model code.
- 6 Examine the model entry-point function interfaces in the generated files and the HTML code generation report. For more information about generating and calling model entry-point functions, see “Entry-Point Functions and Scheduling”.

For an example of a model configured to generate reusable, reentrant code, open the example model `rtwdemo_reusable`. Click the button **View Interface Configuration** and examine the **Code interface** parameters on the **Code Generation > Interface** pane.

Code interface packaging:	<input type="text" value="Reusable function"/>	Multi-instance code error diagnostic:	<input type="text" value="Error"/>
		Pass root-level I/O as:	<input type="text" value="Part of model data structure"/>
<input type="checkbox"/> Classic call interface		<input type="checkbox"/> Use dynamic memory allocation for model initialization	
<input checked="" type="checkbox"/> Single output/update function		<input type="checkbox"/> Terminate function required	

Report Generation

- “Reports for Code Generation” on page 16-2
- “HTML Code Generation Report Extensions” on page 16-3
- “HTML Code Generation Report Location” on page 16-5
- “HTML Code Generation Report for Referenced Models” on page 16-6
- “Search Code Generation Report” on page 16-7
- “Generate a Code Generation Report” on page 16-8
- “Generate Code Generation Report After Build Process” on page 16-9
- “Open Code Generation Report” on page 16-11
- “Generate Code Generation Report Programmatically” on page 16-13
- “View Code Generation Report in Model Explorer” on page 16-14
- “Package and Share the Code Generation Report” on page 16-16
- “Traceability in Code Generation Report” on page 16-18
- “View Code Metrics and Definitions in the Generated Code” on page 16-19
- “Web View of Model in Code Generation Report” on page 16-20
- “Analyze the Generated Code Interface” on page 16-24
- “Static Code Metrics” on page 16-36
- “Generate Static Code Metrics Report for Simulink Model” on page 16-39
- “Generate a Static Code Metrics Report for MATLAB Code” on page 16-45
- “Analyze Code Replacements in the Generated Code” on page 16-51
- “Document Generated Code with Simulink Report Generator” on page 16-53

Reports for Code Generation

Simulink Coder software provides an HTML code generation report so that you can view and analyze the generated code. When your model is built, the code generation process produces an HTML file that is displayed in an HTML browser or in the Model Explorer. The code generation report includes:

- The **Summary** section lists version, date, and code generation objectives information. The **Configuration settings at the time of code generation** link opens a noneditable view of the Configuration Parameters dialog box. The dialog box shows the Simulink model settings at the time of code generation, including TLC options.
- The **Subsystem Report** section contains information on nonvirtual subsystems in the model.
- In the **Generated Files** section on the **Contents** pane, you can click the names of source code files generated from your model to view their contents in a MATLAB Web browser window. In the displayed source code, global variables are hypertext that links to their definitions.
- A **Find** box at the top of the window. For more information, see “Search Code Generation Report” on page 16-7.

For an example, see “Generate a Code Generation Report” on page 16-8 and “View Code Generation Report in Model Explorer” on page 16-14.

The contents of HTML reports varies depending on different target types. You can generate individual HTML reports for a subsystem or referenced model. For more information, see “HTML Code Generation Report for Referenced Models” on page 16-6 and “Generate Code for Referenced Models”.

If you have a Simulink Report Generator license, you can document your code generation project in multiple formats, including HTML, PDF, RTF, Microsoft Word, and XML. For an example of how to create a Microsoft Word report, see “Document Generated Code with Simulink Report Generator” on page 16-53.

HTML Code Generation Report Extensions

The Embedded Coder code generation report is an enhanced version of the HTML code generation report. The Simulink Coder build process generates the HTML report. With the Embedded Coder software, you can configure your model to include the following sections in the report:

- The **Code Interface Report** section provides information about the generated code interface, including model entry point functions and input/output data. For more information, see “Analyze the Generated Code Interface” on page 16-24.
- The **Traceability Report** section allows you to account for **Eliminated / Virtual Blocks** that are untraceable versus the listed **Traceable Simulink Blocks / Stateflow Objects / MATLAB Scripts**. This provides a complete mapping between model elements and code. For more information, see “Customize Traceability Reports”.
- The **Static Code Metrics Report** section provides statistics of the generated code. Metrics are estimated from static analysis of the generated code. For more information, see “Static Code Metrics” on page 16-36.
- The **Code Replacements Report** section allows you to account for code replacement library (CRL) functions that were used during code generation, providing a mapping between each replacement instance and the Simulink block that triggered the replacement. For more information, see “Analyze Code Replacements in the Generated Code” on page 16-51.
- The model Web view displays an interactive model diagram within the code generation report and supports traceability between the source code and the model. Therefore, you can share your model and generated code outside of the MATLAB environment. For more information, see “Generate HTML Code Generation Report with Model Web View” on page 16-20.

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

- If you enable code-to-model traceability, hyperlinks within the displayed source code navigate to the blocks or subsystems from which the code is generated. For more information, see “Traceability in Code Generation Report” on page 16-18 and “Trace Code to Model Objects Using Hyperlinks”.

- If you enable model-to-code traceability, you can navigate to the generated code for a block in the model. For more information, see “Trace Model Objects to Generated Code”.
- If you set the **Code coverage tool** parameter on the **Code Generation > Verification** pane, you can view the code coverage data and annotations. For more information, see “Configure SIL and PIL Code Coverage”.
- If you select the **Static code metrics** check box on the **Code Generation > Report** pane, you can view code metrics information and navigate to code definitions and declarations in the generated code. For more information, see “View Code Metrics and Definitions in the Generated Code”.

HTML Code Generation Report Location

The default location for the code generation report files is in the `html` subfolder of the build folder, `model_target_rtw/html/`. *target* is the name of the **System target file** specified on the **Code Generation** pane. The default name for the top-level HTML report file is `model_codegen_rpt.html` or `subsystem_codegen_rpt.html`. For more information on the location of the build folder, see “Control the Location for Generated Files”.

HTML Code Generation Report for Referenced Models

To generate a code generation report for a top model and code generation reports for each referenced model, you need to specify the **Create code generation report** on the **Code Generation > Report** pane for the top model and each referenced model. You can open the code generation report of a referenced model in one of two ways:

- From the top-model code generation report, you can access the referenced model code generation report by clicking a link under **Referenced Models** in the left navigation pane. Clicking a link opens the code generation report for the referenced model in the browser. To navigate back to the top model code generation report, use the **Back** button at the top of the left navigation pane.
- From the referenced model diagram window, select **Code > C/C++ Code > Code Generation Report > Open Model Report**.

To generate a code generation report for a referenced model individually, follow the instructions in “Generate a Code Generation Report” on page 16-8 and “Open Code Generation Report” on page 16-11 for the referenced model.

Search Code Generation Report

When the code generation report is displayed in the MATLAB Web browser window, you can search the report using the **Find** box at the top of the window. The search is not case sensitive.

Pressing **Ctrl-F** sets focus to the **Find** box. Type text into the **Find** box and hit **Enter** to start the search. The search highlights the found terms in the displayed page and scrolls to the first instance found. Press **Enter** to scroll through the subsequent search hits. If no terms are found, the background of the search box is highlighted red.

Generate a Code Generation Report

To generate a code generation report when the model is built:

- 1 In the Simulink Editor, select **Code > C/C++ Code > Code Generation Report > Options**. The Configuration Parameters dialog box opens with the **Code Generation > Report** pane visible.
- 2 Select the **Create code generation report** parameter.
- 3 If you want the code generation report to automatically open after generating code, select the **Open report automatically** parameter (which is enabled by selecting **Create code generation report**).
- 4 Generate code.

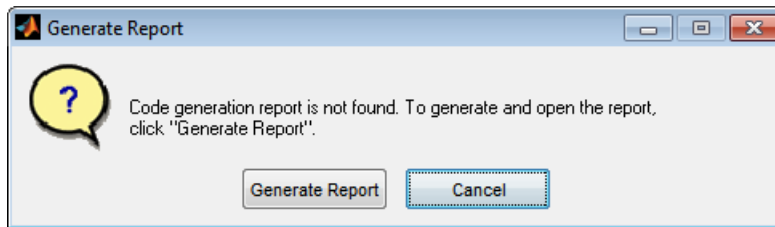
The build process writes the code generation report files to the `html` subfolder of the build folder (see “HTML Code Generation Report Location” on page 16-5). Next, the build process automatically opens a MATLAB Web browser window and displays the code generation report.

To open an HTML code generation report at any time after a build, see “Open Code Generation Report” on page 16-11 and “Generate Code Generation Report After Build Process” on page 16-9.

Generate Code Generation Report After Build Process

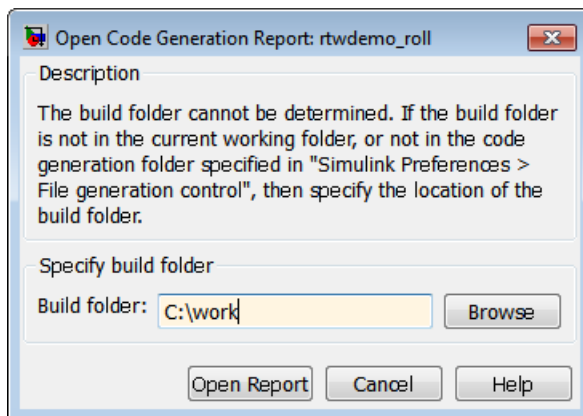
After generating code, if you did not configure your model to create a code generation report, you can generate a code generation report without rebuilding your model.

- 1 In the model diagram window, select **Code > C/C++ Code > Code Generation Report > Open Model Report**.
- 2 If your current working folder contains the code generation files the following dialog opens.



Click **Generate Report**.

- 3 If the code generation files are not in your current working directory, the following dialog opens.



Enter the full path of the build folder for your model, `../model_target_rtw` and click **Open Report**.

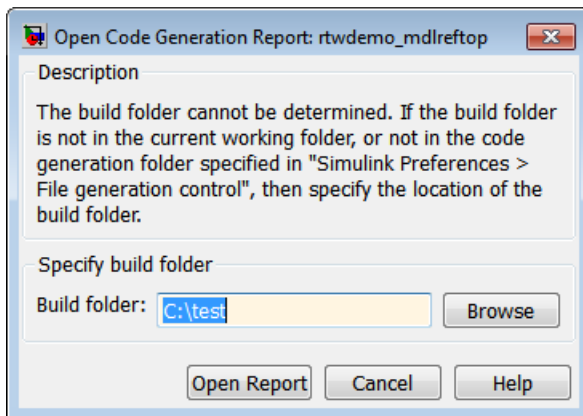
The software generates a report, *model_codgen_rpt.html*, from the code generation files in the build folder you specified.

Note: An alternative method for generating the report after the build process is complete is to configure your model to generate a report and build your model. In this case, the software generates the report without regenerating the code.

Open Code Generation Report

You can refer to existing code generation reports at any time. If you generated a code generation report, you can open the report by selecting **Code > C/C++ Code > Code Generation Report > Open Model Report**. If you are opening a report for a subsystem, select **Open Subsystem Report**. A Simulink Coder license is required to view the code generation report. An Embedded Coder license is required to view a code generation report enhanced with Embedded Coder features.

If your current working folder does not contain the code generation files and the code generation report, the following dialog box opens:



Enter the full path of the build folder for your model, `../model_target_rtw` and click **Open Report**.

Alternatively, you can open the code generation report (`model_codegen_rpt.html` or `subsystem_codegen_rpt.html`) manually into a MATLAB Web browser window, or in another Web browser. For the location of the generated report files, see “HTML Code Generation Report Location” on page 16-5.

Limitation

After building your model or generating the code generation report, if you modify legacy or custom code, you must rebuild your model or regenerate the report for the code generation report to include the updated legacy source files. For example, if you modify your legacy code, and then use the **Code > C/C++ Code > Code Generation**

Report > Open Model Report menu to open an existing report, the software does not check if the legacy source file is out of date compared to the generated code. Therefore, the code generation report is not regenerated and the report includes the out-of-date legacy code. This issue also occurs if you open a code generation report using the `coder.report.open` function.

To regenerate the code generation report, do one of the following:

- Rebuild your model.
- Generate the report using the `coder.report.generate` function.

Generate Code Generation Report Programmatically

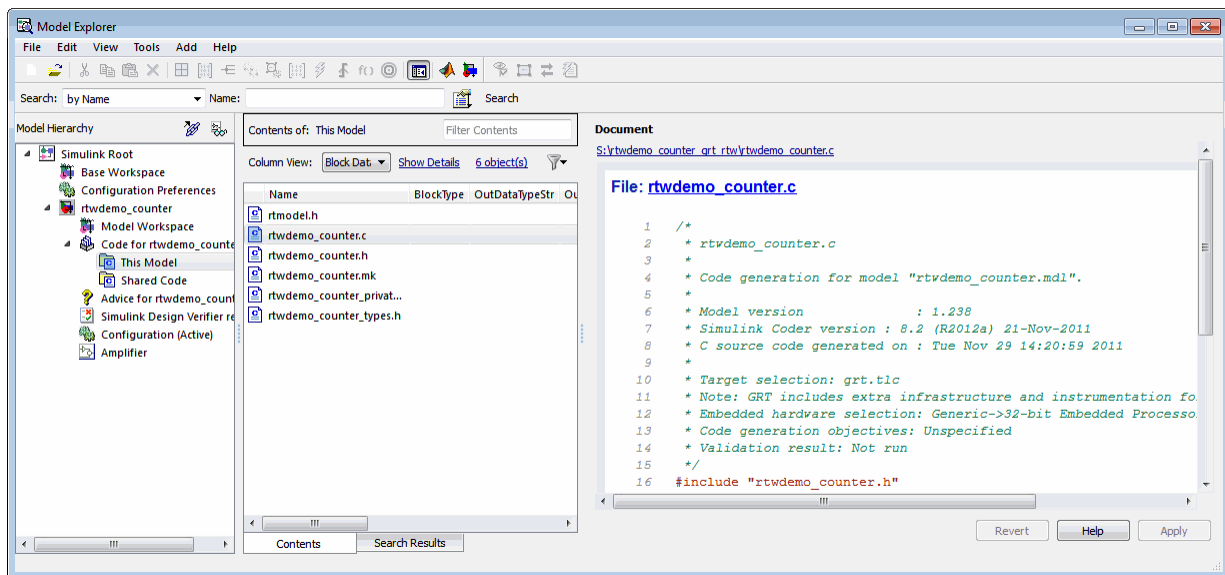
At the MATLAB command line, you can generate, open, and close an HTML Code Generation Report with the following functions:

- `coder.report.generate` generates the code generation report for the specified model.
- `coder.report.open` opens an existing code generation report.
- `coder.report.close` closes the code generation report.

View Code Generation Report in Model Explorer

After generating an HTML code generation report, you can view the report in the right pane of the Model Explorer. You can also browse the generated files directly in the Model Explorer.

When you generate code, or open a model that has generated code for its current target configuration in your working folder, the **Hierarchy** (left) pane of Model Explorer contains a node named **Code for *model1***. Under that node are other nodes, typically called **This Model** and **Shared Code**. Clicking **This Model** displays in the **Contents** (middle) pane a list of generated source code files in the build folder of that model. The next figure shows code for the `rtwdemo_counter` model.



In this example, the file `S:/rtwdemo_counter_grt_rtw/rtwdemo_counter.c` is being displayed. To view a file in the **Contents** pane, click it once.

The views in the **Document** (right) pane are read only. The code listings there contain hyperlinks to functions and macros in the generated code. Clicking the file hyperlink opens that source file in a text editing window where you can modify its contents.

If an open model contains Model blocks, and if generated code for these models exists in the current `slprj` folder, nodes for the referenced models appear in the **Hierarchy** pane

one level below the node for the top model. Such referenced models do not need to be open for you to browse and read their generated source files.

If the Simulink Coder software generates shared utility code for a model, a node named **Shared Code** appears directly under the **This Model** node. It collects source files that exist in the `./slprj/target/_sharedutils` subfolder.

Note You cannot use the **Search** tool built into Model Explorer toolbar to search generated code displayed in the Code Viewer. On PCs, typing **Ctrl+F** when focused on the **Document** pane opens a Find dialog box that you can use to search for strings in the currently displayed file. You can also search for text in the HTML report window, and you can open the files in the editor.

Package and Share the Code Generation Report

In this section...
“Package the Code Generation Report” on page 16-16
“View the Code Generation Report” on page 16-17

Package the Code Generation Report

To share the code generation report, you can package the code generation report files and supporting files into a zip file for transfer. The default location for the code generation report files is in two folders:

- /slprj
- html subfolder of the build folder, *model_target_rtw*, for example *rtwdemo_counter_grt_rtw/html*

To create a zip file from the MATLAB command window:

1 In the Current Folder browser, select the two folders:

- /slprj
- Build folder: *model_target_rtw*

2 Right-click to open the context menu.

3 In the context menu, select **Create Zip File**. A file appears in the Current Folder browser.

4 Name the zip file.

Alternatively, you can use the MATLAB `zip` command to zip the code generation report files:

```
zip('myzip',{'slprj','rtwdemo_counter_grt_rtw'})
```

Note: If you need to relocate the static and generated code files for a model to another development environment, such as a system or an integrated development environment (IDE) that does not include MATLAB and Simulink products, use the Simulink Coder pack-and-go utility. For more information, see “Relocate Code to Another Development Environment”.

View the Code Generation Report

To view the code generation report after transfer, unzip the file and save the two folders at the same folder level in the hierarchy. Navigate to the *model_target_rtw/html/* folder and open the top-level HTML report file named *model_codgen_rpt.html* or *subsystem_codegen_rpt.html* in a Web browser.

Traceability in Code Generation Report

This example shows how to create an HTML code generation report which includes links to trace between the source code and the Simulink model window.

- 1 With your ERT-based model open, open the Configuration Parameters dialog box or Model Explorer and navigate to the **Code Generation > Report** pane.
- 2 Select **Create code generation report** if it is not already selected. By default, **Open report automatically** and **Code-to-model** are selected. **Model-to-code** is not selected.
- 3 Select the **Model-to-code** parameter.
- 4 If your model contains referenced models and you want to enable traceability for the referenced model's code generation report, repeat steps 2–3 for each referenced model.
- 5 Generate code for your model by clicking **Build** on the **Code Generation** pane of the Configuration Parameters dialog box. The build process opens the code generation report in a MATLAB Web browser.
- 6 In the left navigation pane, select a source code file. In the source code in the right pane, there are hyperlinks to blocks in the model.
- 7 Click a hyperlink in the code. The model diagram window displays and highlights the corresponding block in the model.
- 8 To highlight the generated code for a block in your Simulink model, right-click the block and select **C/C++ Code > Navigate to C/C++ Code**. This selection highlights the generated code for the block in the HTML code generation report.
- 9 If you have a referenced model in your model, in the left navigation pane, below **Reference Models**, click the link to a referenced model. The code generation report for the referenced model is now displayed in the window.
- 10 In the left navigation pane, click the **Back** button to go back to the previous code generation report.

Related Examples

- “Trace Model Objects to Generated Code”
- “Trace Code to Model Objects Using Hyperlinks”

More About

- “Traceability Limitations”

View Code Metrics and Definitions in the Generated Code

When you view code in the code generation report, to get access to code metrics and definitions, you can use the following tools:

- On the **Code Generation > Report** pane, if you select the **Static code metrics** check box you can hover your cursor over global variables and functions in the code window to see code metrics information.

```

141 * UnitDelay: '<Root>/X'
142 */
143 /* Gateway: Chart */
144 if (rtDWork.temporalCounter_i1 < 7U) {
145     rtDWork: Global Variable: rtDWork (19 byte)
146 }

```

- In the code window, if you click linked variables or functions, the code inspect window is displayed. The window provides links to definitions for the variables or functions. On the **Code Generation > Report** pane, if you selected the **Static code metrics** check box, you can also see code metrics information for the variable or function.

```

29 /* Block states (auto storage) */
30 D_Work rtDWork;
31
32 /* External outputs (root outputs fed by signals with auto storage) */
33 ExternalOutputs rtY;

```

Global Variable: *rtDWork* (19 byte)
rtDWork defined at [rtwdemo_hyperlinks.c line 30](#)

Web View of Model in Code Generation Report

In this section...

“About Model Web View” on page 16-20

“Generate HTML Code Generation Report with Model Web View” on page 16-20

“Model Web View Limitations” on page 16-23

About Model Web View

To review and analyze the generated code, it is helpful to navigate between the code and model. You can include a Web view of the model within the HTML code generation report. You can then share your model and generated code outside of the MATLAB environment. When you generate the report, the Web view includes the block diagram attributes displayed in the Simulink Editor, such as, block sorted execution order, signal properties, and port data types.

A Simulink Report Generator license is required to include a “Web view” of the model in the code generation report.

Browser Requirements for Web View

Web view requires a Web browser that supports Scalable Vector Graphics (SVG). Web view uses SVG to render and navigate models.

You can use the following Web browsers:

- Mozilla Firefox Version 1.5 or later, which has native support for SVG. To download the Firefox browser, go to www.mozilla.com/.
- The Microsoft Internet Explorer® Web browser with the Adobe® SVG Viewer plug-in. To download the Adobe SVG Viewer plug-in, go to www.adobe.com/svg/.
- Apple Safari Web browser

Generate HTML Code Generation Report with Model Web View

This example shows how to create an HTML code generation report which includes a Web view of the model diagram.

- 1 Open the `rtwdemo_mdleftop` model.

- 2 Open the Configuration Parameters dialog box or Model Explorer and navigate to the **Code Generation** pane.
- 3 Specify `ert.tlc` for the **System target file** parameter.
- 4 Open the **Code Generation > Report** pane.
- 5 Select the following parameters:
 - **Create code generation report**
 - **Open report automatically**
 - **Code-to-model**
 - **Model-to-code**
 - **Generate model Web view**

Note: These settings specify only the top model, not referenced models.

- 6 Open the Configuration Parameters for the referenced model, `rtwdemo_mdhrefbot` and perform steps 3–5.
- 7 Save the models, `rtwdemo_mdhreftop` and `rtwdemo_mdhrefbot`.
- 8 From the top model diagram, press Ctrl+B. After building the model and generating code, the code generation report for the top model opens in a MATLAB Web browser.
- 9 In the left navigation pane, select a source code file. The corresponding source code is displayed in the right pane and includes hyperlinks.

The screenshot shows the 'Code Generation Report' window. The top pane displays C code for a model step function. The bottom pane shows a Simulink model diagram with three blocks: 'rtwdemo_mdirefbot', 'rtwdemo_mdirefbot_upper', and 'rtwdemo_mdirefbot_lower'. A text box below the diagram explains Model Reference. The left navigation pane lists various reports and code files, with 'rtwdemo_mdirefbot.c' highlighted under 'Model files'.

```

49  /* Model step function */
50  void rtwdemo_mdirefbot_step(void)
51  {
52      /* local block i/o variables */
53      real_T rtb_output;
54      real_T rtb_PulseTs01;
55      real_T rtb_output_o;
56      real_T rtb_PulseTs05;
57      real_T rtb_output_oI;
58      real_T rtb_PulseTs10;
59
60      /* DiscretePulseGenerator: '<Root>/Pulse (Ts=0.1)' */
61      rtb_PulseTs01 = (rtwdemo_mdirefbot_DW.clockTickCounter < 1) &&
62      (rtwdemo_mdirefbot_DW.clockTickCounter >= 0) ? 1.0 : 0.0;
63      if (rtwdemo_mdirefbot_DW.clockTickCounter >= 1) {
64          rtwdemo_mdirefbot_DW.clockTickCounter = 0;
65      } else {
66          rtwdemo_mdirefbot_DW.clockTickCounter++;
67      }
68  }

```

Simulink Root
rtwdemo_mdirefbot

This example shows Model Reference, which provides system interface encapsulation and incremental code generation. With Model Reference, you reference one model from another model (one or more times), whereby all aspects of the referenced model are fixed: input/output signal types, parameter types, and sample times. This allows you to modularize your design, and provides incremental code generation for Simulink and Simulink Coder.

The data and functions of a Model Reference system is partitioned into its own set of files, independent of its parent model. In this example, the model `rtwdemo_mdirefbot.mdl` is referenced three times. For simulation and code generation, the model is incrementally generated. That is, `rtwdemo_mdirefbot.mdl` will build the first time, but not on subsequent builds (until `rtwdemo_mdirefbot.mdl` is changed).

Since Model Reference shares and manages code for models, each code generation target must be built into its own directory. The blue buttons below generate code for Simulink Coder and Embedded Coder.

Generate Code Using Simulink Coder Generate Code Using Embedded Coder

OK Help

- 10 Click a link in the code. The model Web view displays and highlights the corresponding block in the model.
- 11 To highlight the generated code for a referenced model block in your model, click CounterB. The corresponding code is highlighted in the source code pane.

Note: You cannot open the referenced model diagram in the Web view by double-clicking the referenced model block in the top model.

- 12 To open the code generation report for a referenced model, in the left navigation pane, below **Referenced Models**, click the link, `rtwdemo_mdirefbot`. The

source files for the referenced model are displayed along with the Web view of the referenced model.

- 13** To go back to the code generation report for the top model, at the top of the left navigation pane, click the **Back** button until the top model's report is displayed.

For more information about exploring a model in a Web view, see “Navigate the Web View” in the Simulink Report Generator documentation.

For more information about navigating between the generated code and the model diagram, see :

- “Trace Model Objects to Generated Code”
- “Trace Code to Model Objects Using Hyperlinks”

Model Web View Limitations

The HTML code generation report includes the following limitations when using the model Web view:

- Code is not generated for virtual blocks. In the model Web view of the code generation report, when tracing between the model and the code, when you click a virtual block, it is highlighted yellow.
- In the model Web view, you cannot open a referenced model diagram by double-clicking the referenced model block in the top model. Instead, open the code generation report for the referenced model by clicking a link under **Referenced Models** in the left navigation pane.
- Stateflow truth tables, events, and links to library charts are not supported in the model Web view.
- Searching in the code generation report does not find or highlight text in the model Web view.
- If you navigate from the actual model diagram (not the model Web view in the report), to the source code in the HTML code generation report, the model Web view is disabled and not visible. To enable the model Web view, open the report again, see “Open Code Generation Report”.
- For a subsystem build, the traceability hyperlinks of the root level inport and outport blocks are disabled.
- “Traceability Limitations” that apply to tracing between the code and the actual model diagram.

Analyze the Generated Code Interface

In this section...

“Code Interface Report Overview” on page 16-24

“Generating a Code Interface Report” on page 16-25

“Navigating Code Interface Report Subsections” on page 16-27

“Interpreting the Entry Point Functions Subsection” on page 16-28

“Interpreting the Inports and Outports Subsections” on page 16-31

“Interpreting the Interface Parameters Subsection” on page 16-32

“Interpreting the Data Stores Subsection” on page 16-34

“Code Interface Report Limitations” on page 16-35

Code Interface Report Overview

When you select the **Create code generation report** option for an ERT-based model, a **Code Interface Report** section is automatically included in the generated HTML report. The **Code Interface Report** section provides documentation of the generated code interface, including model entry point functions and interface data, for consumers of the generated code. The information in the report can help facilitate code review and code integration.

The code interface report includes the following subsections:

- **Entry Point Functions** — interface information about each model entry point function, including `model_initialize`, `model_step`, and (if applicable) `model_terminate`.
- **Inports and Outports** — interface information about each model inport and outport.
- **Interface Parameters** — interface information about tunable parameters that are associated with the model.
- **Data Stores** — interface information about global data stores and data stores with non-auto storage that are associated with the model.

For limitations that apply to code interface reports, see “Code Interface Report Limitations” on page 16-35.

For illustration purposes, this section uses the following models:

- `rtwdemo_basicsc` (with the **ExportedGlobal Storage Class** button selected in the model window) for examples of report subsections
- `rtwdemo_mrmmtbb` for examples of timing information
- `rtwdemo_fcnpctctrl` for examples of function argument and return value information

Generating a Code Interface Report

To generate a code interface report for your model:

- 1 Open your model, go to the **Code Generation** pane of the Configuration Parameters dialog box, and select `ert.tlc` or an ERT-based **System target file**, if one is not already selected.
- 2 Go to the **Code Generation > Report** pane of the Configuration Parameters dialog box and select the option **Create code generation report**, if it is not already selected. The `rtwdemo_basicsc`, `rtwdemo_mrmmtbb`, and `rtwdemo_fcnpctctrl` models used in this section select multiple **Report** pane options by default. But selecting only **Create code generation report**, generates a **Code Interface Report** section in the HTML report.

Alternatively, you can programmatically select the option by issuing the following MATLAB command:

```
set_param(bdroot, 'GenerateReport', 'on')
```

If the **Report** pane option **Code-to-model** is selected, the generated report contains hyperlinks to the model. Leave this value selected unless you plan to use the report outside the MATLAB environment.

- 3 Build the model. If you selected the **Report** pane option **Open report automatically**, the code generation report opens automatically after the build process is complete. (Otherwise, you can open it manually from within the model build folder.)
- 4 To display the code interface report for your model, go to the **Contents** pane of the HTML report and click the **Code Interface Report** link. For example, here is the generated code interface report for the model `rtwdemo_basicsc` (with the **ExportedGlobal Storage Class** button selected in the model window).

Code Interface Report for rtwdemo_basicsc

Table of Contents

- [Entry Point Functions](#)
- [Inports](#)
- [Outports](#)
- [Interface Parameters](#)
- [Data Stores](#)

Entry Point Functions

Function: [rtwdemo_basicsc_initialize](#)

Prototype	void rtwdemo_basicsc_initialize(void)
Description	Initialization entry point of generated code
Timing	Must be called exactly once
Arguments	None
Return value	None
Header file	rtwdemo_basicsc.h

Function: [rtwdemo_basicsc_step](#)

Prototype	void rtwdemo_basicsc_step(void)
Description	Output entry point of generated code
Timing	Must be called periodically, every 1 second
Arguments	None
Return value	None
Header file	rtwdemo_basicsc.h

Inports

Block Name	Code Identifier	Data Type	Dimension
<Root>/In1	input1	real32_T	1
<Root>/In2	input2	real32_T	1
<Root>/In3	input3	real32_T	1
<Root>/In4	input4	real32_T	1

Outports

Block Name	Code Identifier	Data Type	Dimension
<Root>/Out1	output	real32_T	1

Interface Parameters

Parameter Source	Code Identifier	Data Type	Dimension
K2	K2	real_T	1
LOWER	LOWER	real32_T	1
T1Break	T1Break	real32_T	[1 11]
T1Data	T1Data	real32_T	[1 11]
T2Break	T2Break	real32_T	[1 3]
T2Data	T2Data	real32_T	[3 3]
UPPER	UPPER	real32_T	1
K1	K1	int8_T	1

Data Stores

Data Store Source	Code Identifier	Data Type	Dimension
<Root>/Data Store Memory	mode	boolean_T	1

For help navigating the content of the code interface report subsections, see “Navigating Code Interface Report Subsections” on page 16-27. For help interpreting the content of the code interface report subsections, see the sections beginning with “Interpreting the Entry Point Functions Subsection” on page 16-28.

Navigating Code Interface Report Subsections

To help you navigate code interface descriptions, the code interface report provides collapse/expand tokens and hyperlinks, as follows:

- For a large subsection, the report provides [-] and [+] symbols that allow you to collapse or expand that section. In the example in the previous section, the symbols are provided for the **Inports** and **Interface Parameters** sections.
- Several forms of hyperlink navigation are provided in the code interface report. For example:
 - The **Table of Contents** located at the top of the code interface report provides links to each subsection.
 - You can click each function name to go to its definition in *model.c*.
 - You can click each function's header file name to go to the header file source listing.
 - If you selected the **Report** pane option **Code-to-model** for your model, to go to the corresponding location in the model display, you can click hyperlinks for any of the following:
 - Function argument
 - Function return value
 - Inport
 - Outport
 - Interface parameter (if the parameter source is a block)
 - Data store (if the data store source is a Data Store Memory block)

For backward and forward navigation within the HTML code generation report, use the **Back** and **Forward** buttons above the **Contents** section in the upper-left corner of the report.

Interpreting the Entry Point Functions Subsection

The **Entry Point Functions** subsection of the code interface report provides the following interface information about each model entry point function, including `model_initialize`, `model_step`, and (if applicable) `model_terminate`.

Field	Description
Function:	Lists the function name. You can click the function name to go to its definition in <code>model.c</code> .
Prototype	Displays the function prototype, including the function return value, name, and arguments.
Description	Provides a text description of the function's purpose in the application.
Timing	Describes the timing characteristics of the function, such as how many times the function is called, or if it is called periodically, and at what time interval. For a multirate timing example, see the following <code>rtwdemo_mrmtbb</code> report excerpt.
Arguments	If the function has arguments, displays the number, name, data type, and Simulink description for each argument. If you selected the Report pane option Code-to-model for your model, you can click the hyperlink in the description to go to the block corresponding to the argument in the model display. For argument examples, see the <code>rtwdemo_fcncnprotoctrl</code> report excerpt below.
Return value	If the function has a return value, this field displays the return value data type and Simulink description. If you selected the Report pane option Code-to-model for your model, you can click the hyperlink in the description to go to the block corresponding to the return value in the model display. For a return value example, see the following <code>rtwdemo_fcncnprotoctrl</code> report excerpt.
Header file	Lists the name of the header file for the function. You can click the header file name to go to the header file source listing.

For example, here is the **Entry Point Functions** subsection for the model `rtwdemo_basicsc`.

Entry Point Functions

Function: [rtwdemo_basicsc_initialize](#)

Prototype	void rtwdemo_basicsc_initialize(void)
Description	Initialization entry point of generated code
Timing	Must be called exactly once
Arguments	None
Return value	None
Header file	rtwdemo_basicsc.h

Function: [rtwdemo_basicsc_step](#)

Prototype	void rtwdemo_basicsc_step(void)
Description	Output entry point of generated code
Timing	Must be called periodically, every 1 second
Arguments	None
Return value	None
Header file	rtwdemo_basicsc.h

To illustrate how timing information might be listed for a multirate model, here are the **Entry Point Functions** and **Inports** subsections for the model `rtwdemo_mrmtbb`. This multirate, discrete-time, multitasking model contains Inport blocks 1 and 2, which specify 1-second and 2-second sample times, respectively. The sample times are constrained to the specified times by the **Periodic sample time constraint** option on the **Solver** pane of the Configuration Parameters dialog box.

Entry Point Functions

Function: [rtwdemo_mrmtbb_initialize](#)

Prototype	void rtwdemo_mrmtbb_initialize(void)
Description	Initialization entry point of generated code
Timing	Must be called exactly once
Arguments	None
Return value	None
Header file	rtwdemo_mrmtbb.h

Function: [rtwdemo_mrmtbb_step0](#)

Prototype	void rtwdemo_mrmtbb_step0(void)
Description	Output entry point of generated code
Timing	Must be called periodically, every 1 second
Arguments	None
Return value	None
Header file	rtwdemo_mrmtbb.h

Function: [rtwdemo_mrmtbb_step1](#)

Prototype	void rtwdemo_mrmtbb_step1(void)
Description	Output entry point of generated code
Timing	Must be called periodically, every 2 seconds
Arguments	None
Return value	None
Header file	rtwdemo_mrmtbb.h

Function: [rtwdemo_mrmtbb_terminate](#)

Prototype	void rtwdemo_mrmtbb_terminate(void)
Description	Termination entry point of generated code
Timing	Must be called exactly once
Arguments	None
Return value	None
Header file	rtwdemo_mrmtbb.h

Inports

Block Name	Code Identifier	Data Type	Dimension
<Root>/In1_1s	rtwdemo_mrmtbb_U.In1_1s	real_T	1
<Root>/In2_2s	rtwdemo_mrmtbb_U.In2_2s	real_T	1

To illustrate how function arguments and return values are displayed in the report, here is the **Entry Point Functions** description of the model step function for the model `rtwdemo_fcnprotctrl`.

Function: [rtwdemo_fcnprotctrl_step_custom](#)

Prototype	boolean_T <code>rtwdemo_fcnprotctrl_step_custom(const real_T argIn1, const BusObject *const argIn2, BusObject *argOut2, const BusObject *const argIn3, uint8_T *argIn4)</code>		
Description	Output entry point of generated code		
Timing	Can be called at any time		
Arguments	[-]		
	# Name	Data Type	Description
	1 argIn1	const real_T	<Root>/In1
	2 argIn2	const BusObject *const	<Root>/In2
	3 argOut2	BusObject *	<Root>/Out2
	4 argIn3	const BusObject *const	<Root>/In3
	5 argIn4	uint8_T *	<Root>/In4
Return value	Data Type	Description	
	boolean_T	<Root>/Out1	
Header file	rtwdemo_fcnprotctrl.h		

Interpreting the Inports and Outports Subsections

The **Inports** and **Outports** subsections of the code interface report provide the following interface information about each inport and outport in the model.

Field	Description
Block Name	Displays the Simulink block name of the inport or outport. If you selected the Report pane option Code-to-model for your model, you can click on each inport or outport Block Name value to go to its location in the model display.
Code Identifier	Lists the identifier associated with the inport or outport data in the generated code, as follows: <ul style="list-style-type: none"> If the data is defined in the generated code, the field displays the identifier string.

Field	Description
	<ul style="list-style-type: none"> If the data is declared but not defined in the generated code — for example, if the data is resolved with an imported storage class — the field displays the identifier string prefixed with the label 'Imported data:'. If the data is neither defined nor declared in the generated code — for example, if Reusable function code interface packaging is selected for the model — the field displays the string 'Defined externally'.
Data Type	Lists the data type of the inport or outport.
Dimension	Lists the dimensions of the inport or outport (for example, 1 or [4, 5]).

For example, here are the **Inports** and **Outports** subsections for the model `rtwdemo_basicsc`.

Inports

[-]

Block Name	Code Identifier	Data Type	Dimension
<Root>/In1	input1	real32_T	1
<Root>/In2	input2	real32_T	1
<Root>/In3	input3	real32_T	1
<Root>/In4	input4	real32_T	1

Outports

Block Name	Code Identifier	Data Type	Dimension
<Root>/Out1	output	real32_T	1

Interpreting the Interface Parameters Subsection

The **Interface Parameters** subsection of the code interface report provides the following interface information about tunable parameters that are associated with the model.

Field	Description
Parameter Source	<p>Lists the source of the parameter value, as follows:</p> <ul style="list-style-type: none"> If the source of the parameter value is a block, the field displays the block name, such as <code><Root>/Gain2</code> or <code><S1>/</code>

Field	Description
	<p>Lookup1. If you selected the Report pane option Code-to-model for your model, you can click the Parameter Source value to go to the parameter's location in the model display.</p> <ul style="list-style-type: none"> If the source of the parameter value is a workspace variable, the field displays the name of the workspace variable.
Code Identifier	<p>Lists the identifier associated with the tunable parameter data in the generated code, as follows:</p> <ul style="list-style-type: none"> If the data is defined in the generated code, the field displays the identifier string. If the data is declared but not defined in the generated code — for example, if the data is resolved with an imported storage class — the field displays the identifier string prefixed with the label 'Imported data:'. If the data is neither defined nor declared in the generated code — for example, if Reusable function code interface packaging is selected for the model — the field displays the string 'Defined externally'.
Data Type	Lists the data type of the tunable parameter.
Dimension	Lists the dimensions of the tunable parameter (for example, 1 or [4, 5, 6]).

For example, here is the **Interface Parameters** subsection for the model `rtwdemo_basicsc` (with the **ExportedGlobal Storage Class** button selected in the model window).

Interface Parameters

[-]

Parameter Source	Code Identifier	Data Type	Dimension
K2	K2	real_T	1
LOWER	LOWER	real32_T	1
T1Break	T1Break	real32_T	[1 11]
T1Data	T1Data	real32_T	[1 11]
T2Break	T2Break	real32_T	[1 3]
T2Data	T2Data	real32_T	[3 3]
UPPER	UPPER	real32_T	1
K1	K1	int8_T	1

Interpreting the Data Stores Subsection

The **Data Stores** subsection of the code interface report provides the following interface information about global data stores and data stores with non-`auto` storage that are associated with the model.

Field	Description
Data Store Source	Lists the source of the data store memory, as follows: <ul style="list-style-type: none"> • If the data store is defined using a Data Store Memory block, the field displays the block name, such as <code><Root>/DS1</code>. If you selected the Report pane option Code-to-model for your model, you can click on the Data Store Source value to go to the data store's location in the model display. • If the data store is defined using a <code>Simulink.Signal</code> object, the field displays the name of the <code>Simulink.Signal</code> object.
Code Identifier	Lists the identifier associated with the data store data in the generated code, as follows: <ul style="list-style-type: none"> • If the data is defined in the generated code, the field displays the identifier string. • If the data is declared but not defined in the generated code — for example, if the data is resolved with an imported storage class — the field displays the identifier string prefixed with the label <code>'Imported data:'</code>. • If the data is neither defined nor declared in the generated code — for example, if Reusable function code interface packaging is selected for the model — the field displays the string <code>'Defined externally'</code>.
Data Type	Lists the data type of the data store.
Dimension	Lists the dimensions of the data store (for example, 1 or [1, 2]).

For example, here is the **Data Stores** subsection for the model `rtwdemo_basicsc` (with the **ExportedGlobal Storage Class** button selected in the model window).

Data Stores

Data Store Source	Code Identifier	Data Type	Dimension
<Root>/Data Store Memory	mode	boolean_T	1

Code Interface Report Limitations

The following limitations apply to the code interface section of the HTML code generation reports.

- The code interface report does not support the GRT interface with an ERT target or **C++ class** code interface packaging. For these configurations, the code interface report is not generated and does not appear in the HTML code generation report **Contents** pane.
- The code interface report supports data resolved with most custom storage classes (CSCs), except when the CSC properties are set in any of the following ways:
 - The CSC property **Type** is set to **FlatStructure**. For example, the **BitField** and **Struct** CSCs in the Simulink package have **Type** set to **FlatStructure**.
 - The CSC property **Type** is set to **Other**. For example, the **GetSet** CSC in the Simulink package has **Type** set to **Other**.
 - The CSC property **Data access** is set to **Pointer**, indicating that imported symbols are declared as pointer variables rather than simple variables. This property is accessible only when the CSC property **Data scope** is set to **Imported** or **Instance-specific**.

In these cases, the report displays empty **Data Type** and **Dimension** fields.

- For outputs, the code interface report cannot describe the associated memory (data type and dimensions) if the memory is optimized. In these cases, the report displays empty **Data Type** and **Dimension** fields.
- The code interface report does not support data type replacement using the **Code Generation > Data Type Replacement** pane of the Configuration Parameters dialog box. The data types listed in the report will link to built-in data types rather than their specified replacement data types.

Static Code Metrics

In this section...

“About Static Code Metrics” on page 16-36

“Static Code Metrics Analysis” on page 16-36

About Static Code Metrics

The code generator performs static analysis of the generated C code and provides these metrics in the **Static Code Metrics Report** section of the HTML Code Generation Report.

You can use the information in the report to:

- Find the number of files and lines of code in each file.
- Estimate the number of lines of code and stack usage per function.
- Compare the difference in terms of how many files, functions, variables, and lines of code are generated every time you change the model or MATLAB algorithm.
- Determine a target platform and allocation of RAM to the stack, based on the size of global variables plus the estimated stack size.
- Determine possible performance slow points, such as the largest global variables or the most costly call path in terms of stack usage.
- View the cyclomatic complexity of a function, which counts the number of linearly independent paths through a function.
- View the function call tree. Determine the longest call path to estimate the worst case execution timing.
- View how target functions, provided by the selected code replacement library, are used in the generated code.

For examples, see

- “Generate Static Code Metrics Report for Simulink Model” on page 16-39
- “Generate a Static Code Metrics Report for MATLAB Code” on page 16-45

Static Code Metrics Analysis

Static analysis of the generated code is performed only on the source code without executing the program. The results of the static code metrics analysis are included in

the **Static Code Metrics** section of the HTML Code Generation Report. The static code metrics report does not support the C++ target language. The report is not available if you generate a MEX function from MATLAB code.

Static analysis of the generated source code files:

- Uses the specified C data types. For Simulink models, you specify these data types in the **Hardware Implementation > Production hardware** pane of the Configuration Parameters dialog box. For code generation from MATLAB code, you specify them in the **Hardware** tab of the MATLAB Coder Project Settings dialog box or using a code generation configuration object. Actual object code metrics might differ due to target-specific compiler and platform settings.
- Includes custom code only if you specify it. For Simulink models, you specify custom code on the **Code Generation > Custom Code** pane in the model configuration. For code generation from MATLAB code, you specify it on the **Custom Code** tab of the MATLAB Coder Project Settings dialog box or using a code generation configuration object. An error report is generated if the generated code includes platform-specific files not contained in the standard C run-time library.
- For Simulink models, includes the generated code from referenced models.
- Uses 1-byte alignment for all members of a structure for estimating global and local data structure sizes. The size of a structure is calculated by summing the sizes of all of its fields. This estimation represents the smallest possible size for a structure.
- Calculates the self stack size of a function as the size of local data within a function, excluding input arguments. The accumulated stack size of a function is the self stack size plus the maximum of the accumulated stack sizes of its called functions. For example, if the accumulated stacks sizes for the called functions are represented as `accum_size1...accum_sizeN`, then the accumulated stack size for a function is

$$\text{accumulated_stack_size} = \text{self_stack_size} + \max(\text{accum_size1}, \dots, \text{accum_sizeN})$$
- When estimating the stack size of a function, static analysis stops at the first instance of a recursive call. The **Function Information** table indicates when recursion occurs in a function call path. Code generation generates only recursive code for Stateflow event broadcasting and for graphical functions if it is written as a recursive function.
- Calculates the cyclomatic complexity of a function as the number of decisions plus one:

$CC = \text{Number of decisions} + 1$

The following constructs add a decision:

- If statement

- Else-If statement
- Switch statement (1 decision for each `case` branch)
- Loop statements: While, For, Do-while

Note: Boolean operators in the above constructs do not add extra decisions.

- Does not include `ert_main.c`, because you have the option to provide your own `main.c`.

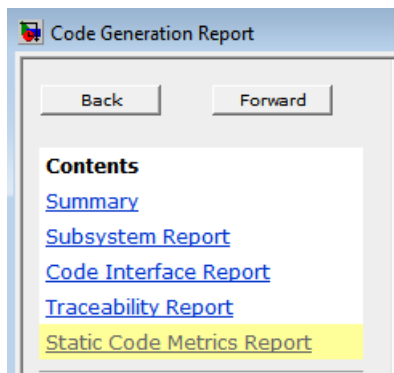
Generate Static Code Metrics Report for Simulink Model

The **Static Code Metrics Report** is a section included in the HTML Code Generation Report. For more information on the static analysis of the generated code, see “Static Code Metrics Analysis” on page 16-36.

- 1 Before generating the HTML Code Generation Report, open the Configuration Parameters dialog box for your model. On the **Code Generation > Report** pane, select the “Static code metrics” check box.

If your model includes referenced models, select the **Static code metrics** check box in each referenced model’s configuration set. Otherwise, you cannot view a separate static code metrics report for a referenced model.

- 2 Press **Ctrl+B** to build your model and generate the HTML code generation report. For more information, see “Traceability in Code Generation Report”.
- 3 If the HTML Code Generation Report is not already open, open the report. On the left navigation pane, in the **Contents** section, select **Static Code Metrics Report**.



- 4 Hover your cursor over column titles and some column values to see a description of the corresponding data.

[-] File details

File Name	Lines of Code	Lines	Generated On
fuel_rate_control.c	577	1,009	12/10/2013 3:14 PM
fuel_rate	252	341	12/10/2013 3:14 PM
fuel_rate	100	260	12/10/2013 3:14 PM
rtwtypes.h	67	135	12/10/2013 3:14 PM

Files sorted in descending order by number of lines of code.

- 5 To see the generated files and how many lines of code are generated per file, look at the **File Information** section.

1. File Information [\[hide\]](#)

[-] Summary (excludes ert_main.c)

Number of .c files : 2
 Number of .h files : 2
 Lines of code : 996
 Lines : 1,745

[-] File details

File Name	Lines of Code	Lines	Generated On
fuel_rate_control.c	577	1,009	12/10/2013 3:14 PM
fuel_rate_control_data.c	252	341	12/10/2013 3:14 PM
fuel_rate_control.h	100	260	12/10/2013 3:14 PM
rtwtypes.h	67	135	12/10/2013 3:14 PM

- 6 If your model includes referenced models, the File information section includes a **Referenced Model** column. In this column, click the referenced model name to open its static code metrics report. If the static code metrics report is not available for a referenced model, specify the **Static code metrics** parameter in the referenced model's configuration set and rebuild your model.
- 7 To view the global variables in the generated code, their size, and the number of accesses, see the **Global Variables** section.

2. Global Variables [\[hide\]](#)

Global variables defined in the generated code.

Global Variable	Size (bytes)	Reads / Writes	Reads / Writes in a Function
[+] rtDWork	55	250	137
[+] rtU	16	26	24
[+] rtM	4	0*	0*
[+] rtY	4	3	3
Total	79	279	

* The global variable is not directly used in any function.

The **Reads/Writes** column displays the total number of read and write accesses to the global variable. The **Reads/Writes in a Function** column displays the maximum number of read and write accesses to the global variable within a function. You use this information to estimate the benefit of turning on optimizations, which reduce the number of global references. For more information, see “Optimize Global Variable Usage”.

Click [+] to expand structures.

Global variables defined in the generated code.

Global Variable	Size (bytes)	Reads / Writes	Reads / Writes in a Function
[-] rtDWork	55	250	137
[+] es_o	16	22	18
ThrottleEstimation	4	2	2
PressureEstimation	4	2	2
DiscreteIntegrator_DSTATE	4	4	4
ThrottleTransient_states	4	3	3
DiscreteFilter_states	4	3	3
DiscreteFilter_states_i	4	3	3
sfEvent	4	42	25
fuel_mode	4	14	8
[+] bitsForTIDO	5	150	68
temporalCounter_i1	2	5	5
[-] rtU	16	26	24
[+] sensors	16	26	24
[-] rtM_	4	0+	0+
errorStatus	4	0	0
[-] rtY	4	3	3
fuel_rate	4	3	3
Total	79	279	

* The global variable is not directly used in any function.

- 8 To navigate from the report to the source code, click a global variable or function name. These names are hyperlinks to their definitions.
- 9 To view the function call tree of the generated code, in the **Function Information** section, click **Call Tree** at the top of the table.

3. Function Information [\[hide\]](#)

View function metrics in a call tree format or table format. Accumulated stack numbers include the estimated stack size of the function plus the maximum of the accumulated stack size of the subroutines that the function calls.

View: [Call Tree](#) | [Table](#)

Function Name	Accumulated Stack Size (bytes)	Self Stack Size (bytes)	Lines of Code	Lines	Complexity
[-] fuel_rate_control_step	60	20	257	492	47
look2_iff_linca	40	40	57	102	12
[-] Fail	4	4	67	102	18
Fueling_Mode	0	0	132	209	25
Fueling_Mode	0	0	132	209	25
fuel_rate_control_initialize	0	0	9	24	1

ert_main.c is not included in the code metrics analysis, therefore it is not shown in the call tree format. The **Complexity** column includes the cyclomatic complexity of each function.

- To view the functions in a table format, click **Table**.

3. Function Information [\[hide\]](#)

View function metrics in a call tree format or table format. Accumulated stack numbers include the estimated stack size of the function plus the maximum of the accumulated stack size of the subroutines that the function calls.

View: [Call Tree](#) | [Table](#)

Function Name	Called By (number of call sites)	Accumulated Stack Size (bytes)	Self Stack Size (bytes)	Lines of Code	Lines	Complexity
Fail	fuel_rate_control_step (9)	4	4	67	102	18
Fueling_Mode	Fail (2) fuel_rate_control_step	0	0	132	209	25
fuel_rate_control_initialize		0	0	9	24	1
fuel_rate_control_step		60	20	257	492	47
look2_iff_linca	fuel_rate_control_step (5)	40	40	57	102	12

The second column, **Called By**, lists functions that call the function listed in the first column, using the following criteria:

- If a function is called by multiple functions, all functions are listed.
- If a function has no called function, this column is empty.

For example, `Fueling_Mode` is called by `Fail` and `fuel_rate_control_step`. The number of call sites is included in parentheses. `Fail` calls `Fueling_Mode` twice.

Generate a Static Code Metrics Report for MATLAB Code

In a MATLAB Coder Project

This example shows how to generate a static code metrics report for a static C library generated from MATLAB code in a MATLAB Coder project.

By default, when you generate standalone C code with MATLAB Coder, the code generation report now includes a static code metrics report. The static code metrics report is not available for generated MEX functions.

- 1 In a local, writable folder, create a MATLAB file, `moving_average.m`, that contains:

```
function [avg,z] = moving_average(x,z)
%#codegen
    if nargin < 2,
        z = zeros(10,1);
    end
    z(2:end) = z(1:end-1); % Update buffer
    z(1) = x; % Add new value
    avg = mean(z); % Compute moving average
end
```

- 2 In the same folder, set up a MATLAB Coder project.

- a At the MATLAB command line, enter:

```
coder -new moving_average.prj
```

By default, the project opens in the MATLAB workspace on the right side.

- b On the project **Overview** tab, click the **Add files** link. Browse to the file `moving_average.m`. Click **OK** to add the file to the project.

The file is displayed on the **Overview** tab. Both inputs are undefined.

- c Define the type of input `u`.
 - i On the **Overview** tab, click the field to the right of the input parameter `x` and, from the list of input options, select **double**.
 - ii From the list of size options, select `1 x 1` to specify that the input is a scalar.

- 3 Repeat the previous step for input `z`.

- 4 In the MATLAB Coder project, click the **Build** tab.
- 5 On this tab, set the **Output type** to **C/C++ Static library**.

The default output file name is `moving_average`.

- 6 On this tab, click **More settings** to open the Project Settings dialog box.
- 7 In the Project Settings dialog box, select the Debugging tab and verify that **Static code metrics** is selected.
- 8 Click the **Build** button to generate a library.

MATLAB Coder builds the project and generates a C static library and supporting files in the default folder, `codegen/lib/moving_average` and provides a link to the code generation report.

- 9 Open the code generation report and click **Static Code Metrics Report** to open the report.
- 10 To see the generated files and how many lines of code are generated per file, look at the **File Information** section.

Static Code Metrics Report

The static code metrics report provides statistics of the generated code. Metrics are estimated from static analysis of the generated code using the C data types specified in the **Hardware Settings** > Test Hardware: **char** 8, **short** 16, **int** 32, **long** 32, **float** 32, **double** 64, **pointer** 32 bits. Actual object code metrics might differ due to target specific compiler and platform settings.

Table of Contents

1. [File Information](#)
2. [Global Variables](#)
3. [Function Information](#)

1. File Information [\[hide\]](#)

[-] Summary

Number of .c files : 6
 Number of .h files : 8
 Lines of code : 393
 Lines : 747

[-] File details

File Name	Lines of Code	Lines	Generated On
rtGetInf.c	92	140	11/27/2012 9:11 AM
rtwtypes.h	90	161	11/27/2012 9:11 AM
rtGetNaN.c	63	97	11/27/2012 9:11 AM
rt_nonfinite.c	43	88	11/27/2012 9:11 AM
rt_nonfinite.h	37	54	11/27/2012 9:11 AM
rtGetInf.h	10	24	11/27/2012 9:11 AM
moving_average.c	9	30	11/27/2012 9:11 AM
moving_average.h	8	23	11/27/2012 9:11 AM
moving_average_initialize.h	8	23	11/27/2012 9:11 AM
moving_average_terminate.h	8	23	11/27/2012 9:11 AM
rtGetNaN.h	8	22	11/27/2012 9:11 AM
moving_average_initialize.c	7	22	11/27/2012 9:11 AM
moving_average_terminate.c	6	22	11/27/2012 9:11 AM
moving_average_types.h	4	18	11/27/2012 9:11 AM

- 11 To view the global variables in the generated code and their size, see the **Global Variables** section.

2. Global Variables [\[hide\]](#)

Global variables defined in the generated code.

Global Variable	Size (bytes)
rtInf	8
rtMinusInf	8
rtNaN	8
rtInfF	4
rtMinusInfF	4
rtNaNF	4
Total	36

To navigate from the report to the source code, click a global variable name. The names are hyperlinks to their definitions.

- 12 To view the function call tree of the generated code, in the **Function Information** section, click **Call Tree** at the top of the table.

3. Function Information [\[hide\]](#)

View function metrics in a call tree format or table format. Accumulated stack numbers include the estimated stack size of the function plus the maximum of the accumulated stack size of the subroutines that the function calls.

View: [Call Tree](#) | [Table](#)

Function Name	Accumulated Stack Size (bytes)	Self Stack Size (bytes)	Lines of Code	Lines	Complexity
[+] moving_average_initialize	44	0	1	4	1
moving_average	8	8	4	13	1
moving_average_terminate	0	0	0	4	1
rtIsInf	0	0	1	4	2
rtIsInfF	0	0	1	4	2
rtIsNaN	0	0	5	8	2
rtIsNaNF	0	0	5	8	2

To navigate from the report to the function code, click a function name.

- 13 To view the functions in a table format, click **Table**.

3. Function Information [\[hide\]](#)

View function metrics in a call tree format or table format. Accumulated stack numbers include the estimated stack size of the function plus the maximum of the accumulated stack size of the subroutines that the function calls.

View: [Call Tree](#) | [Table](#)

Function Name	Called By (number of call sites)	Accumulated Stack Size (bytes)	Self Stack Size (bytes)	Lines of Code	Lines	Complexity
moving_average		8	8	4	13	1
moving_average_initialize		44	0	1	4	1
moving_average_terminate		0	0	0	4	1
rtGetInf		38	34	36	43	5
rtGetInfF	rtGetInf	4	4	3	6	1
rtGetMinusInf		38	34	36	43	5
rtGetMinusInfF	rtGetMinusInf	4	4	3	6	1
rtGetNaN		44	34	36	43	5
rtGetNaNF	rtGetNaN	10	10	19	24	4
rtIsInf		0	0	1	4	2
rtIsInfF		0	0	1	4	2
rtIsNaN		0	0	5	8	2
rtIsNaNF		0	0	5	8	2
rt_InitInfAndNaN		44	0	7	10	1

The second column, **Called By**, lists functions that call the function listed in the first column, using the following criteria:

- If a function is called by multiple functions, all functions are listed.
- If a function has no called function, this column is empty.

At the Command Line

To enable a static code metrics report at the command line:

- 1 Create a code generation configuration object for standalone code generation. For example, to generate a static library:

```
cfg = coder.config('lib');
```

- 2 Generate code, passing the configuration object as a parameter and specifying the `-report` option. For example:

```
codegen -config cfg -report foo
```

Alternatively,

- 1 Create a code generation configuration object for standalone code generation. For example, to generate a static library:

```
cfg = coder.config('lib');
```

- 2 Set the configuration object `GenerateReport` and `GenerateCodeMetricsReport` parameters to `true`.

```
cfg.GenerateReport = true;  
cfg.GenerateCodeMetricsReport = true;
```

- 3 Generate code, passing the configuration object as a parameter. For example:

```
codegen -config cfg foo
```

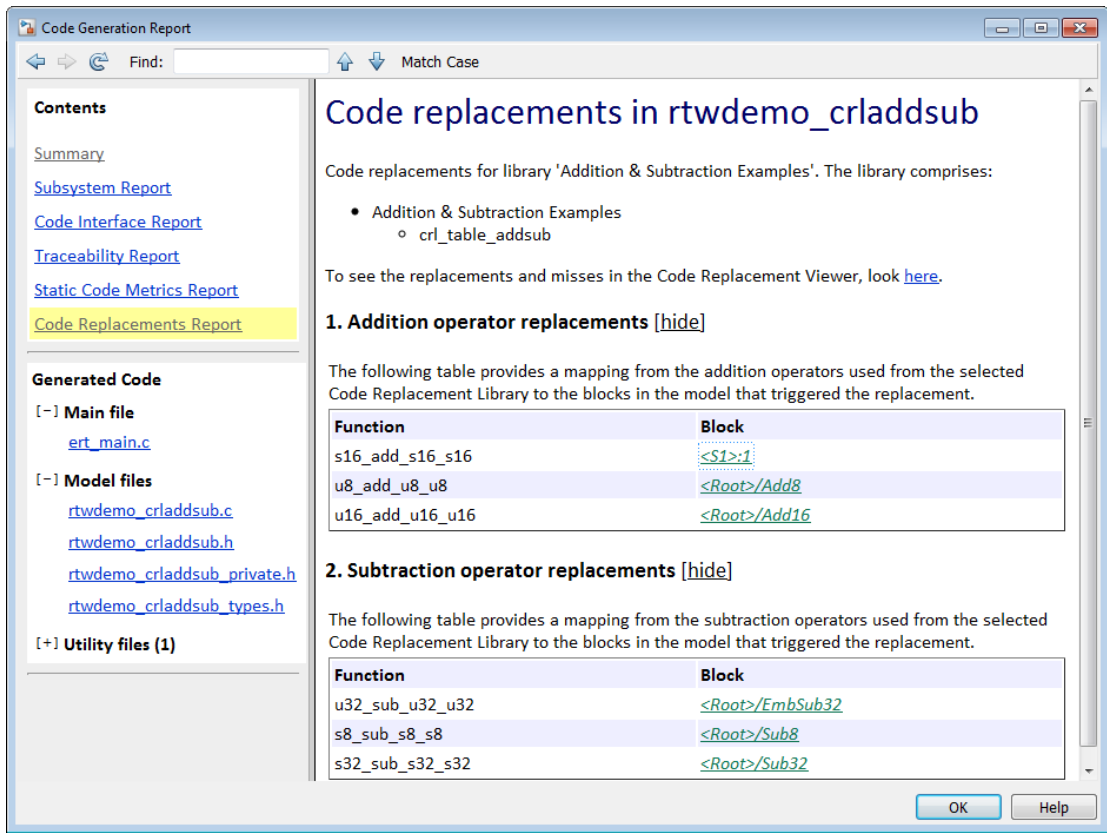

Analyze Code Replacements in the Generated Code

When you select the **Code Generation > Report** check box “**Summarize which blocks triggered code replacements**” for an ERT-based model, a Code Replacements Report section is automatically included in the generated HTML report. The Code Replacements Report section documents the code replacement library (CRL) functions that were used for code replacements during code generation, providing a mapping between each replacement instance and the Simulink block that triggered the replacement. To enable display of the Simulink block information, select the **Code Generation > Comments** check box **Include comments**. On the same pane, select either the **Simulink block / Stateflow object comments** check box or the **Simulink block descriptions** check box if present, or both.

You can use the report to:

- Determine which replacement functions were used in the generated code.
- Trace each replacement instance back to the block that triggered the replacement.

The figure below shows a Code Replacements Report generated for the CRL model `rtwdemo_crladdsub`. Each replacement function used is listed with a link to the block that triggered the replacement.



If you click a block path in the report, the block that triggered the replacement is highlighted in the model diagram. If the replacement was triggered by a Stateflow chart or a MATLAB function, a window opens to display the chart or function.

For more information, see “Trace Code Replacements Generated Using Your Code Replacement Library”.

Document Generated Code with Simulink Report Generator

In this section...

- “Generate Code for the Model” on page 16-54
- “Open the Report Generator” on page 16-54
- “Set Report Name, Location, and Format” on page 16-56
- “Include Models and Subsystems in a Report” on page 16-57
- “Customize the Report” on page 16-58
- “Generate the Report” on page 16-59

The Simulink Report Generator software creates documentation from your model in multiple formats, including HTML, PDF, RTF, Microsoft Word, and XML. This example shows one way to document a code generation project in Microsoft Word. The generated report includes:

- System snapshots (model and subsystem diagrams)
- Block execution order list
- Simulink Coder and model version information for generated code
- List of generated files
- Optimization configuration parameter settings
- Simulink Coder target selection and build process configuration parameter settings
- Subsystem map
- File name, path, and generated code listings for the source code

To adjust Simulink Report Generator settings to include custom code and then generate a report for a model, complete the following tasks:

- 1 “Generate Code for the Model” on page 16-54
- 2 “Open the Report Generator” on page 16-54
- 3 “Set Report Name, Location, and Format” on page 16-56
- 4 “Include Models and Subsystems in a Report” on page 16-57
- 5 “Customize the Report” on page 16-58
- 6 “Generate the Report” on page 16-59

A Simulink Report Generator license is required for the following report formats: PDF, RTF, Microsoft Word, and XML. For more information on generating reports in these formats, see the Simulink Report Generator documentation.

Generate Code for the Model

Before you use the Report Generator to document your project, generate code for the model.

- 1 In the MATLAB Current Folder browser, navigate to a folder where you have write access.
- 2 Create a working folder from the MATLAB command line by typing:

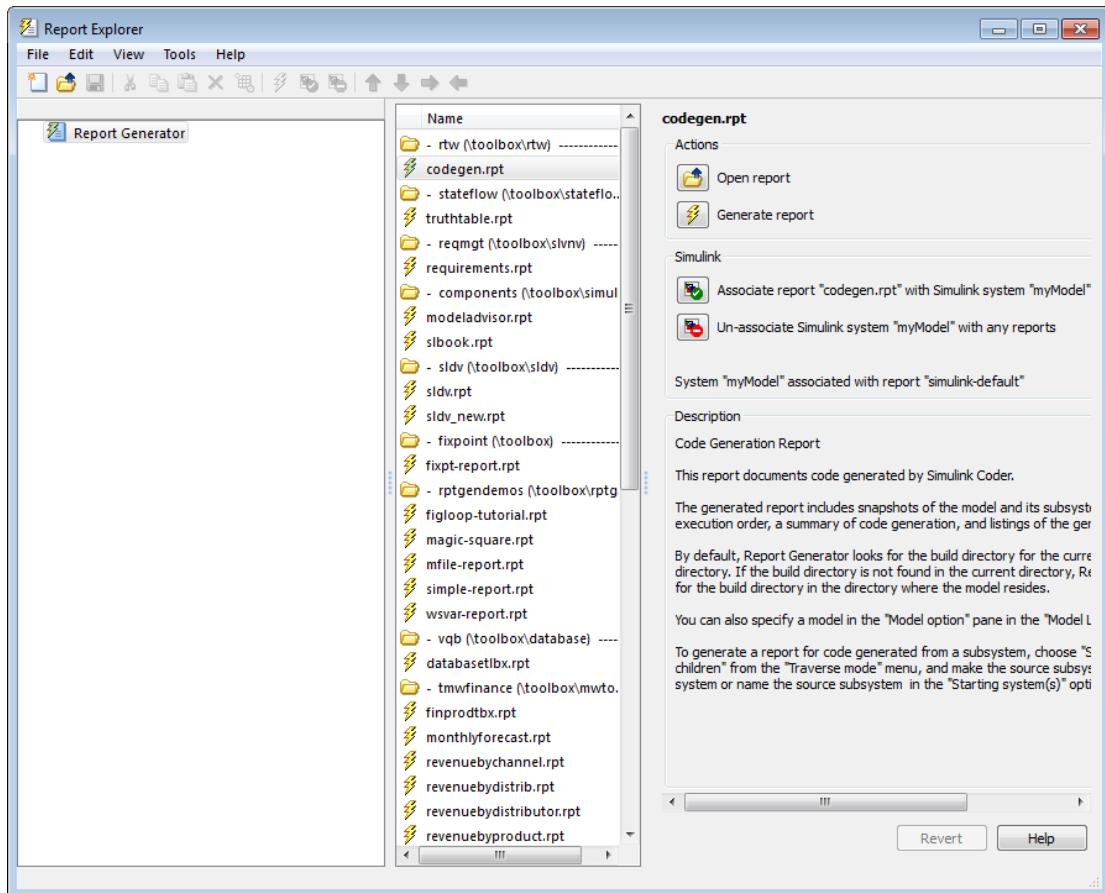
```
mkdir report_ex
```
- 3 Make `report_ex` your working folder:

```
cd report_ex
```
- 4 Open the `slexAircraftExample` model by entering the model name on the MATLAB command line.
- 5 In the model window, choose **File > Save As**, navigate to the working folder, `report_ex`, and save a copy of the `slexAircraftExample` model as `myModel`.
- 6 Open the Configuration Parameters dialog box by selecting **Model Configuration Parameters** from the **Simulation** menu.
- 7 Select the **Solver** pane. In the **Solver options** section, specify the **Type** parameter as **Fixed-step**.
- 8 Select the **Code Generation** pane. Select **Generate code only**.
- 9 Click **Apply**.
- 10 Click **Generate code**. The build process generates code for the model.


Open the Report Generator

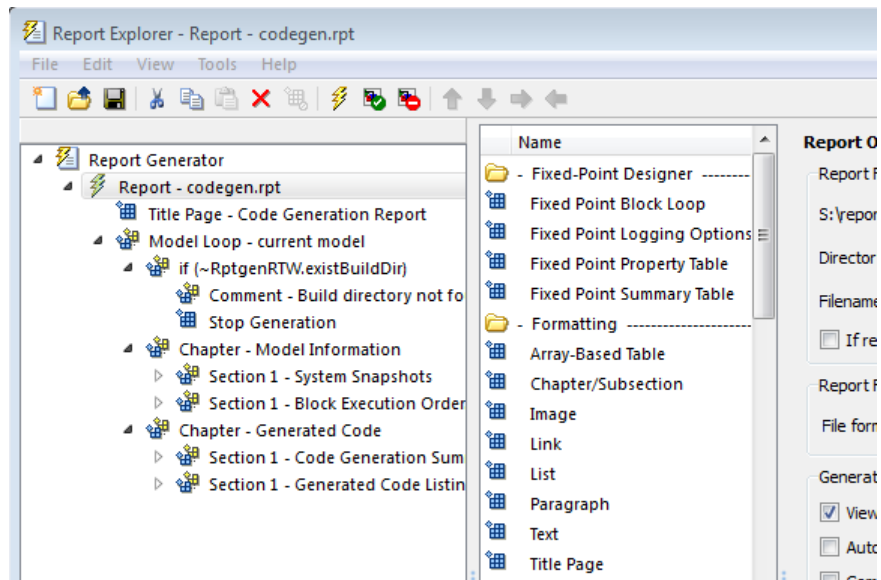
After you generate the code, open the Report Generator.

- 1 In the model diagram window, select **Tools > Report Generator**.
- 2 In the Report Explorer window, in the options pane (center), click the folder `rtw` (`\toolbox\rtw`). Click the setup file that it contains, `codegen.rpt`.



3

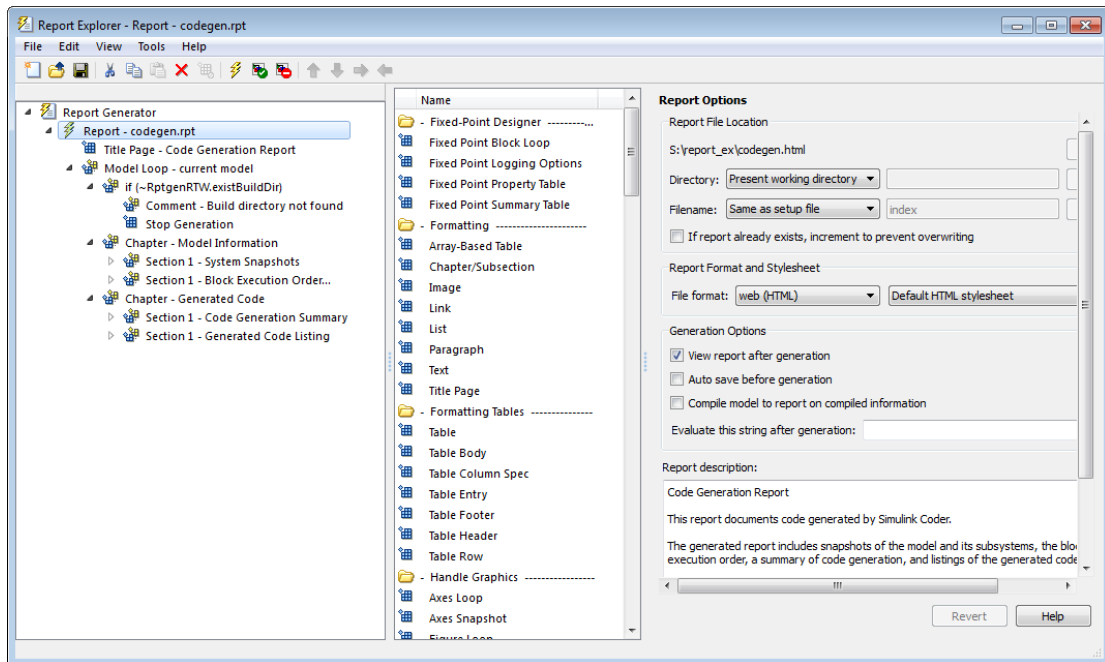
Double-click **codegen.rpt** or select it and click the **Open report** button . The Report Explorer displays the structure of the setup file in the outline pane (left).



Set Report Name, Location, and Format

Before generating a report, you can specify report output options, such as the folder, file name, and format. For example, to generate a Microsoft Word report named `MyCGModelReport.rtf`:

- 1 In the properties pane, under **Report Options**, review the options listed.

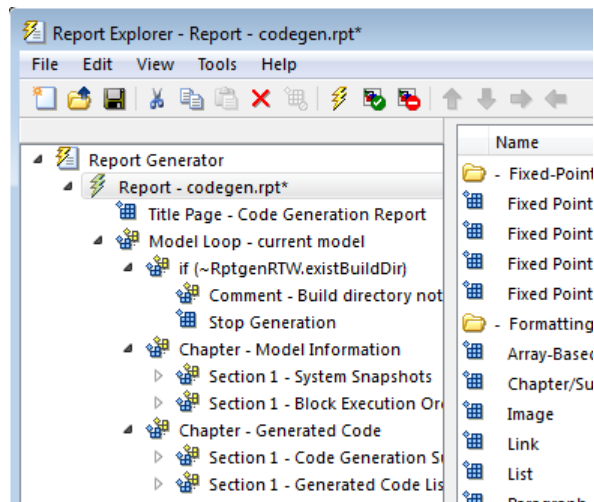


- 2 Leave the **Directory** field set to **Present working directory**.
- 3 For **Filename**, select **Custom:** and replace **index** with the name **MyModelICGreport**.
- 4 For **File format**, specify **Rich Text Format** and replace **Standard Print** with **Numbered Chapters & Sections**.

Include Models and Subsystems in a Report

Specify the models and subsystems that you want to include in the generated report by setting options in the Model Loop component.

- 1 In the outline pane (left), select **Model Loop**. Report Generator displays Model Loop component options in the properties pane.
- 2 If not already selected, select **Current block diagram** for the **Model name** option.
- 3 In the outline pane, click **Report - codegen.rpt***.



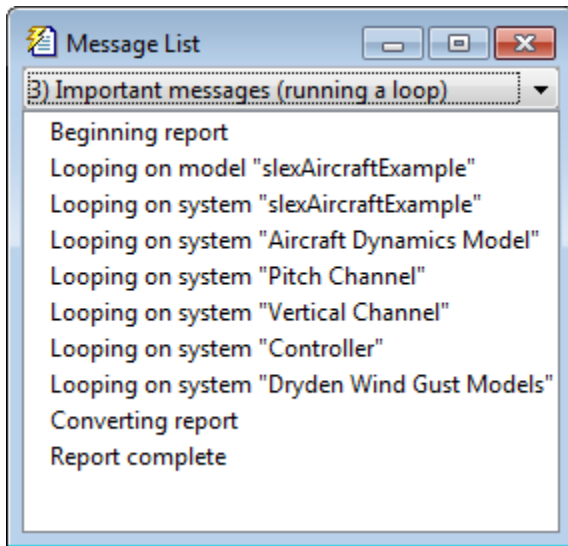
Customize the Report

After specifying the models and subsystems to include in the report, you can customize the sections included in the report.

- 1 In the outline pane (left), expand the node **Chapter - Generated Code**. By default, the report includes two sections, each containing one of two report components.
- 2 Expand the node **Section 1 — Code Generation Summary**.
- 3 Select **Code Generation Summary**. Options for the component are displayed in the properties pane.
- 4 Click **Help** to review the report customizations that you can make with the Code Generation Summary component. For this example, do not customize the component.
- 5 In the Report Explorer window, expand the node **Section 1 — Generated Code Listing**.
- 6 Select **Import Generated Code**. Options for the component are displayed in the properties pane.
- 7 Click **Help** to review the report customizations that you can make with the Import Generated Code component.

Generate the Report

After you adjust the report options, from the **Report Explorer** window, generate the report by clicking **File > Report**. A **Message List** dialog box opens, which displays messages that you can monitor as the report is generated. Model snapshots also appear during report generation. The **Message List** dialog box might be hidden behind other dialog boxes.



When the report is complete, open the report, `MyModelCGReport.rtf` in the folder `report_ex` (in this example).

For alternative ways of generating reports with the Simulink Report Generator, see "Generate Reports".

Code Replacement for Simulink Models

- “What Is Code Replacement?” on page 17-2
- “Code You Can Replace From Simulink Models” on page 17-4
- “Code Replacement Libraries” on page 17-23
- “Code Replacement Terminology” on page 17-25
- “Code Replacement Limitations” on page 17-28
- “Replace Code Generated from Simulink Models” on page 17-29
- “Choose a Code Replacement Library” on page 17-32

What Is Code Replacement?

Code replacement is a technique to change the code that the code generator produces for functions and operators to meet application code requirements. For example, you can replace generated code to meet requirements such as:

- Optimization for a specific run-time environment, including, but not limited to, specific target hardware.
- Integration with existing application code.
- Compliance with a standard, such as AUTOSAR.
- Modification of code behavior, such as enabling or disabling nonfinite or inline support.
- Application- or project-specific code requirements, such as:
 - Elimination of `math.h`.
 - Elimination of system header files.
 - Elimination of calls to `memcpy` or `memset`.
 - Use of BLAS.
 - Use of a specific BLAS.

To apply this technique, configure the code generator to apply a code replacement library (CRL) during code generation. By default, the code generator does not apply a code replacement library. You can choose from the following libraries that MathWorks provides:

- GNU C99 extensions—GNU^{®5} `gcc` math library, which provides C99 extensions as defined by compiler option `-std=gnu99`.
- Intel IPP for x86-64 (Windows)—Generates calls to the Intel[®] Performance Primitives (IPP) library for the x86-64 Windows platform.
- Intel IPP/SSE with GNU99 extensions for x86-64 (Windows)—GNU libraries for Intel Performance Primitives (IPP) and Streaming SIMD Extensions (SSE), with GNU C99 extensions.
- Intel IPP for x86/Pentium (Windows)—Generates calls to the Intel Performance Primitives (IPP) library for the x86/Pentium Windows platform.

5. GNU is a registered trademark of the Free Software Foundation.

- Intel IPP/SSE with GNU99 extensions for x86/Pentium (Windows)—Generates calls to the GNU libraries for Intel Performance Primitives (IPP) and Streaming SIMD Extensions (SSE), with GNU C99 extensions, for the x86/Pentium Windows platform.
- Intel IPP for x86-64 (Linux)—Generates calls to the Intel Performance Primitives (IPP) library for the x86-64 Linux[®] platform.
- Intel IPP/SSE with GNU99 extensions for x86-64 (Linux)—Generates calls to the GNU libraries for Intel Performance Primitives (IPP) and Streaming SIMD Extensions (SSE), with GNU C99 extensions, for the x86-64 Linux platform.

Libraries that include GNU99 extensions are intended for use with the GCC compiler. If use one of those libraries with another compiler, generated code might not compile.

Depending on the product licenses that you have, other libraries might be available . If you have an Embedded Coder license, you can view and choose from other libraries and you can create custom code replacement libraries.

Related Examples

- “Replace Code Generated from Simulink Models”
- “Choose a Code Replacement Library”

More About

- “Code You Can Replace From Simulink Models”
- “Code Replacement Libraries”
- “Code Replacement Terminology”
- “Code Replacement Limitations”

Code You Can Replace From Simulink Models

In this section...

“About Code You Can Replace” on page 17-4

“Math Functions – Simulink Support” on page 17-4

“Math Functions – Stateflow Support” on page 17-10

“Memory Functions” on page 17-15

“Nonfinite Functions” on page 17-16

“Mutex and Semaphore Functions” on page 17-17

“Lookup Table Functions” on page 17-18

“Operators” on page 17-18

About Code You Can Replace

Code that the code generator replaces depends on the code replacement library (CRL) that you use. By default, the code generator does not apply a code replacement library. Your choice of libraries is dependent on product licensing and whether you have access to custom libraries.

For information on how to explore functions and operators that a code replacement library supports, see “Choose a Code Replacement Library” license and want to develop a custom code replacement library, see Code Replacement Customization.

Math Functions – Simulink Support

When generating C/C++ code from a Simulink model, depending on code replacement libraries available in your development environment, you can configure the code generator to replace instances of the following math functions with application-specific implementations.

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
abs ¹	Integer Floating point Fixed point	Scalar Vector Matrix	Real
acos	Floating point	Scalar	Real

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
			Complex input/complex output Real input/complex output
acosd^2	Floating point	Scalar Vector Matrix	Real Complex
acosh	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
acot^2	Floating point	Scalar Vector Matrix	Real Complex
acotd^2	Floating point	Scalar Vector Matrix	Real Complex
acoth^2	Floating point	Scalar Vector Matrix	Real Complex
acsc^2	Floating point	Scalar Vector Matrix	Real Complex
acscd^2	Floating point	Scalar Vector Matrix	Real Complex
acsch^2	Floating point	Scalar Vector Matrix	Real Complex
asec^2	Floating point	Scalar Vector Matrix	Real Complex
asecd^2	Floating point	Scalar Vector Matrix	Real Complex

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
asech ²	Floating point	Scalar Vector Matrix	Real Complex
asin	Floating point	Scalar	Real Complex input/complex output Real input/complex output
asind ²	Floating point	Scalar Vector Matrix	Real Complex
asinh	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
atan	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
atan2	Floating point	Scalar Vector Matrix	Real
atan2d ²	Floating point	Scalar Vector Matrix	Real
atand ²	Floating point	Scalar Vector Matrix	Real Complex
atanh	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
ceil	<ul style="list-style-type: none"> • Floating-point • Scalar 	<ul style="list-style-type: none"> • Floating-point • Scalar 	<ul style="list-style-type: none"> • Floating-point • Scalar
cos ³	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
\cosd^2	Floating point	Scalar Vector Matrix	Real Complex
cosh	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
\cot^2	Floating point	Scalar Vector Matrix	Real Complex
\cotd^2	Floating point	Scalar Vector Matrix	Real Complex
\coth^2	Floating point	Scalar Vector Matrix	Real Complex
\csc^2	Floating point	Scalar Vector Matrix	Real Complex
\cscd^2	Floating point	Scalar Vector Matrix	Real Complex
\csch^2	Floating point	Scalar Vector Matrix	Real Complex
exactrSqrt	Integer Floating point	Scalar	Real
exp	Floating point	Scalar Vector Matrix	Real
fix	Floating point	Scalar	Real
floor	<ul style="list-style-type: none"> • Floating-point • Scalar 	<ul style="list-style-type: none"> • Floating-point • Scalar 	<ul style="list-style-type: none"> • Floating-point • Scalar

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
fmod ⁴	Floating point	Scalar	Real
frexp	Floating point	Scalar	Real
hypot	Floating point	Scalar Vector Matrix	Real
ldexp	Floating point	Scalar	Real
ln	Floating point	Scalar	Real
log	Floating point	Scalar Vector Matrix	Real
log10	Floating point	Scalar Vector Matrix	Real
log2 ²	Floating point	Scalar Vector Matrix	Real Complex
max	Integer Floating point Fixed point	Scalar	Real
min	Integer Floating point Fixed point	Scalar	Real
mod	Integer Floating point	Scalar Vector Matrix	Real
pow	Floating point	Scalar Vector Matrix	Real
rem	Floating point	Scalar Vector Matrix	Real
round	Floating point	Scalar	Real

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
rSqrt	Integer Floating point	Scalar Vector Matrix	Real
saturate	Integer Floating point Fixed point	Scalar Vector Matrix	Real
sec ²	Floating point	Scalar Vector Matrix	Real Complex
secd ²	Floating point	Scalar Vector Matrix	Real Complex
sech ²	Floating point	Scalar Vector Matrix	Real Complex
sign	Integer Floating point Fixed point	Scalar Vector Matrix	Real
signPow	Floating point	Scalar Vector Matrix	Real
sin ³	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
sincos ³	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
sind ²	Floating point	Scalar Vector Matrix	Real Complex
sinh	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
sqrt	Integer Floating point Fixed point	Scalar Vector Matrix	Real
tan	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
tand ²	Floating point	Scalar Vector Matrix	Real Complex
tanh	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
¹ Wrap on integer overflow only. Clear block parameter Saturate on integer overflow . ² Only when used with the MATLAB Function block. ³ Supports the CORDIC approximation method. ⁴ Stateflow support only.			

Math Functions – Stateflow Support

When generating C/C++ code from Stateflow charts, depending on code replacement libraries available in your development environment, you can configure the code generator to replace instances of the following math functions with application-specific implementations.

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
abs ¹	Integer Floating point	Scalar	Real
acos ²	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
			Real input/complex output
acosd^3	Floating point	Scalar Vector Matrix	Real Complex
acot^3	Floating point	Scalar Vector Matrix	Real Complex
acotd^3	Floating point	Scalar Vector Matrix	Real Complex
$\text{acoth}^{3,5}$	Floating point	Scalar Vector Matrix	Real Complex
acsc^3	Floating point	Scalar Vector Matrix	Real Complex
acscd^3	Floating point	Scalar Vector Matrix	Real Complex
acsch^3	Floating point	Scalar Vector Matrix	Real Complex
asec^3	Floating point	Scalar Vector Matrix	Real Complex
asecd^3	Floating point	Scalar Vector Matrix	Real Complex
asech^3	Floating point	Scalar Vector Matrix	Real Complex

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
asin^2	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
asind^3	Floating point	Scalar Vector Matrix	Real Complex
atan^2	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
atan2^2	Floating point	Scalar Vector Matrix	Real
atan2d^3	Floating point	Scalar Vector Matrix	Real
atand^3	Floating point	Scalar Vector Matrix	Real Complex
ceil	<ul style="list-style-type: none"> • Floating-point • Scalar 	<ul style="list-style-type: none"> • Floating-point • Scalar 	<ul style="list-style-type: none"> • Floating-point • Scalar
cos^3	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
cosd^3	Floating point	Scalar Vector Matrix	Real Complex

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
\cosh^2	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
\cot^3	Floating point	Scalar Vector Matrix	Real Complex
\cotd^3	Floating point	Scalar Vector Matrix	Real Complex
\coth^3	Floating point	Scalar Vector Matrix	Real Complex
\csc^3	Floating point	Scalar Vector Matrix	Real Complex
\cscd^3	Floating point	Scalar Vector Matrix	Real Complex
\csch^3	Floating point	Scalar Vector Matrix	Real Complex
\exp	Floating point	Scalar	Real
floor	<ul style="list-style-type: none"> • Floating-point • Scalar 	<ul style="list-style-type: none"> • Floating-point • Scalar 	<ul style="list-style-type: none"> • Floating-point • Scalar
fmod	Floating point	Scalar	Real
hypot^3	Floating point	Scalar Vector Matrix	Real
ldexp	Floating point	Scalar	Real

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
\log^2	Floating point	Scalar Vector Matrix	Real Complex
\log_{10}^2	Floating point	Scalar Vector Matrix	Real Complex
\log_2^3	Floating point	Scalar Vector Matrix	Real Complex
max	Integer Floating point	Scalar	Real
min	Integer Floating point	Scalar	Real
pow	Floating point	Scalar	Real
\sec^3	Floating point	Scalar Vector Matrix	Real Complex
\secd^3	Floating point	Scalar Vector Matrix	Real Complex
\sech^3	Floating point	Scalar Vector Matrix	Real Complex
\sin^2	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
\sind^3	Floating point	Scalar Vector Matrix	Real Complex

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
\sinh^2	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
$\sqrt{}$	Floating point	Scalar	Real
\tan^2	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
\tand^3	Floating point	Scalar Vector Matrix	Real Complex
\tanh^2	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
¹ Wrap on integer overflow only. ² For models involving vectors or matrices, the code generator replaces only functions coded in the MATLAB action language. ³ The code generator replaces only functions coded in the MATLAB action language.			

Memory Functions

Depending on code replacement libraries available in your development environment, you can configure the code generator to replace instances of the following memory functions with application-specific implementations.

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
memcmp	Void pointer (void*)	Scalar	Real

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
		Vector Matrix	Complex
memcpy	Void pointer (void*)	Scalar Vector Matrix	Real Complex
memset	Void pointer (void*)	Scalar Vector Matrix	Real Complex
memset2zero	Void pointer (void*)	Scalar Vector Matrix	Real Complex

Some target processors provide optimized functions to set memory to zero. Use the code replacement library programming interface to replace the `memset2zero` function with more efficient target-specific functions.

Nonfinite Functions

Depending on code replacement libraries available in your development environment, you can configure the code generator to replace instances of the following nonfinite functions with application-specific implementations.

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
getInf	Floating point	Scalar	Real
getMinusInf	Floating point	Scalar	Real
getNaN	Floating point	Scalar	Real
rtIsInf	Floating point	Scalar	Real Complex
rtIsNaN	Floating point	Scalar	Real Complex

Mutex and Semaphore Functions

Mutex and semaphore functions control access to resources shared by multiple processes in multicore target environments. MathWorks provides code replacement libraries that support mutex and semaphore replacement for Rate Transition and Task Transition blocks on Windows, Linux, Mac, and VxWorks platforms.

Generated mutex and semaphore code typically consists of:

- In model initialization code, an initialization function call to create a mutex or semaphore to control entry to a critical section of code.
- In model step code:
 - Before code for a data transfer between tasks enters the critical section, mutex lock or semaphore wait function calls to reserve a critical section of code.
 - After code for a data transfer between tasks finishes executing the critical section, mutex unlock or semaphore post function calls to release the critical section of code.
- In model termination code, an optional destroy function call to explicitly delete the mutex or semaphore.

Depending on code replacement libraries available in your development environment, you can configure the code generator to replace instances of the following mutex and semaphore functions with application-specific implementations.

In the following table, key is a string that identifies the function.

Function	Key
Mutex Destroy	RTW_MUTEX_DESTROY
Mutex Init	RTW_MUTEX_INIT
Mutex Lock	RTW_MUTEX_LOCK
Mutex Unlock	RTW_MUTEX_UNLOCK
Semaphore Destroy	RTW_SEM_DESTROY
Semaphore Init	RTW_SEM_INIT
Semaphore Post	RTW_SEM_POST
Semaphore Wait	RTW_SEM_WAIT

Lookup Table Functions

Depending on available code replacement libraries, you can configure the code generator to replace instances of the following lookup table functions with application-specific implementations. Support for these functions includes:

- Integer, floating-point, and fixed-point data types
- Scalar, vector, and matrix data formats
- Real and complex data

interp1D	interp5D	interpND	lookup5D
interp2D	interp5D	lookup1D	lookupND
interp3D	interp5D	lookup2D	lookupND_Direct
interp4D	interp5D	lookup3D	prelookup
interp5D	interp5D	lookup4D	

Operators

When generating C/C++ code from a Simulink model, depending on code replacement libraries available in your development environment, you can configure the code generator to replace instances of the following operators with application-specific implementations.

In the following table:

- Key is a string that identifies the operator.
- Mixed data type support indicates that you can specify different data types for different inputs.

Operator	Key	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
Addition (+)	RTW_OP_ADD	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Subtraction (-)	RTW_OP_MINUS	Integer	Scalar	Real

Operator	Key	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
		Floating point Fixed-point Mixed	Vector Matrix	Complex
Multiplication (*) ¹	RTW_OP_MUL	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Division (/)	RTW_OP_DIV	Integer Floating point Fixed-point Mixed	Scalar	Real Complex
Data type conversion (cast)	RTW_OP_CAST	Integer Floating point ² Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Shift left (<<)	RTW_OP_SL	Integer Fixed-point Mixed	Scalar Vector Matrix	Real
Shift right arithmetic (>>) ³	RTW_OP_SRA	Integer Fixed-point Mixed	Scalar Vector Matrix	Real
Shift right logical (>>)	RTW_OP_SRL	Integer Fixed-point Mixed	Scalar Vector Matrix	Real
Element- wise matrix multiplication (. *) ⁴	RTW_OP_ELEM_MUL	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Matrix right division (/)	RTW_OP_RDIV	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex

Operator	Key	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
Matrix left division (\)	RTW_OP_LDIV	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Matrix inversion (inv)	RTW_OP_INV	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Complex conjugation	RTW_OP_CONJUGATE	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Transposition (.')	RTW_OP_TRANS	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Hermitian (complex conjugate) transposition (')	RTW_OP_HERMITIAN	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Multiplication with transposition ¹	RTW_OP_TRMUL	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Multiplication with Hermitian transposition ¹	RTW_OP_HMMUL	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Greater than (>)	RTW_OP_GREATER_THAN	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex

Operator	Key	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
Greater than or equal (\geq)	RTW_OP_GREATER_THAN_OR_EQUAL	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Less than ($<$)	RTW_OP_LESS_THAN	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Less than or equal (\leq)	RTW_OP_LESS_THAN_OR_EQUAL	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Equal (\equiv)	RTW_OP_EQUAL	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Not equal (\neq)	RTW_OP_NOT_EQUAL	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
<p>¹ Can map to Basic Linear Algebra Subroutine (BLAS) multiplication functions.</p> <p>² Scaled floating point is not supported.</p> <p>³ Code replacement libraries that provide arithmetic shift right implementations should also provide logical shift right implementations, because some arithmetic shift rights are converted to logical shift rights during code generation.</p> <p>⁴ Use the multiplication ($*$) operator (RTW_OP_MUL) for scalar multiplication.</p>				

Related Examples

- “Choose a Code Replacement Library”

More About

- “What Is Code Replacement?”
- “Code Replacement Limitations”
- “Code Replacement Terminology”
- “Code Replacement Libraries”

Code Replacement Libraries

A *code replacement library* consists of one or more code replacement tables that specify application-specific implementations of functions and operators. For example, a library for a specific embedded processor specifies function and operator replacements that optimize generated code for that processor.

A *code replacement table* contains one or more *code replacement entries*, with each entry 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.

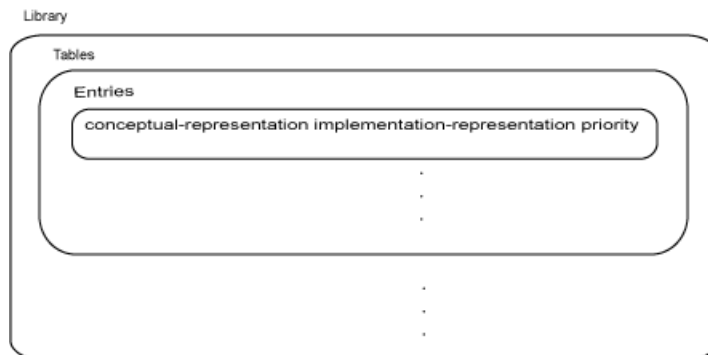


Table Entry Component	Description
Conceptual representation	<p>Identifies the table entry and contains match criteria for the code generator. Consists of:</p> <ul style="list-style-type: none"> • Function name or a key. The function name identifies most functions. For operators and some functions, a string called a key identifies a function or operator. For example, function name 'COS' and operator key 'RTW_OP_ADD'. • Conceptual arguments that observe code generator naming ('y1', 'u1', 'u2', ...), with corresponding I/O types (output or input) and data types. • Other attributes, such as an algorithm, fixed-point saturation, and rounding modes, which identify matching criteria for the function or operator.

Table Entry Component	Description
Implementation representation	Specifies replacement code. Consists of: <ul style="list-style-type: none"> • Function name. For example, 'cos_db1' or 'u8_add_u8_u8') • Implementation arguments, with corresponding I/O types (output or input) and data types. • Parameters that provide additional implementation details, such as header and source file names and paths of build resources.
Priority	Defines the entry priority relative to other entries in the table. The value can range from 0 to 100, with 0 being the highest priority. If multiple entries have the same priority, the code generator uses the first match with that priority.

When the code generator looks for a match in a code replacement library, it creates and populates a *call site object* with the function or operator conceptual representation. If a match exists, the code generator uses the matched code replacement entry populated with the implementation representation and uses it to generate code.

The code generator searches the tables in a code replacement library for a match in the order that the tables appear in the library. If the code generator finds multiple matches within a table, the priority determines the match. The code generator uses a higher-priority entry over a similar entry with a lower priority.

Related Examples

- “Replace Code Generated from Simulink Models”
- “Choose a Code Replacement Library”

More About

- “What Is Code Replacement?”
- “Code You Can Replace From Simulink Models”
- “Code Replacement Terminology”

Code Replacement Terminology

Term	Definition
Cache hit	A code replacement entry for a function or operator, defined in the specified code replacement library, for which the code generator finds a match.
Cache miss	A conceptual representation of a function or operator for which the code generator does not find a match.
Call site object	Conceptual representation of a function or operator that the code generator uses when it encounters a call site for a function or operator. The code generator uses the object to query the code replacement library for a conceptual representation match. If a match exists, the code generator returns a code replacement object, fully populated with the conceptual representation, implementation representation, and priority, and uses that object to generate replacement code.
Code replacement library	One or more code replacement tables that specify application-specific implementations of functions and operators. When configured to use a code replacement library, the code generator uses criteria defined in the library to search for matches. If a match is found, the code generator replaces code that it generates by default with application-specific code defined in the library.
Code replacement table	One or more code replacement table entries. Provides a way to group related or shared entries for use in different libraries.
Code replacement entry	Represents a potential replacement for a function or operator. Maps a conceptual representation of a function or operator to an implementation representation and priority.
Conceptual argument	Represents an input or output argument for a function or operator being replaced. Conceptual arguments observe naming conventions ('y1',

Term	Definition
	'u1', 'u2', ...) and data types familiar to the code generator.
Conceptual representation	<p>Represents match criteria that the code generator uses to qualify functions and operators for replacement. Consists of:</p> <ul style="list-style-type: none"> • Function or operator name or key • Conceptual arguments with type, dimension, and complexity specification for inputs and output • Attributes, such as an algorithm and fixed-point saturation and rounding modes
Implementation argument	Represents an input or output argument for a C or C++ replacement function. Implementation arguments observe C/C++ name and data type specifications.
Implementation representation	<p>Specifies C or C++ replacement function prototype. Consists of:</p> <ul style="list-style-type: none"> • Function name (for example, 'cos_db1' or 'u8_add_u8_u8') • Implementation arguments specifying type, type qualifiers, and complexity for the function inputs and output • Parameters that provide build information, such as header and source file names and paths of build resources and compile and link flags
Key	A string that identifies a function or operator that is being replaced. A function name or key appears in the conceptual representation of a code replacement entry. The key RTW_OP_ADD identifies the addition operator.

Term	Definition
Priority	Defines the match priority for a code replacement entry relative to other entries, which have the same name and conceptual argument list, within a code replacement library. The priority can range from 0 to 100, with 0 being the highest priority. The default is 100. If a library provides two implementations for a function or operator, the implementation with the higher priority shadows the one with the lower priority.

More About

- “What Is Code Replacement?”
- “Code Replacement Libraries”

Code Replacement Limitations

Code replacement verification — It is possible that code replacement behaves differently than you expect. For example, data types that you observe in code generator input might not match what the code generator uses as intermediate data types during an operation. Verify code replacements by examining generated code.

Related Examples

- [“Replace Code Generated from Simulink Models”](#)

More About

- [“Code You Can Replace From Simulink Models”](#)
- [“Code Replacement Libraries”](#)

Replace Code Generated from Simulink Models

This example shows how to replace generated code, using a code replacement library. Code replacement is a technique you can use to change the code that the code generator produces for functions and operators to meet application code requirements.

Prepare for Code Replacement

- 1 Make sure that MATLAB, Simulink, Simulink Coder, and a C compiler are installed on your system. Some code replacement libraries available in your development environment can also require Embedded Coder.

To install MathWorks products, see the MATLAB installation documentation. If you have installed MATLAB and want to see which other MathWorks products are installed, in the Command Window, enter `ver`.

- 2 Identify an existing or create a Simulink model for which you want the code generator to replace code.

Choose a Code Replacement Library

If you are not sure which library to use, explore the available libraries.

Configure Code Generator To Use Code Replacement Library

- 1 Configure the code generator to apply a code replacement library during code generation for the model.
 - In the Configuration Parameters dialog box, on the **Code Generation > Interface** pane, select a library from the **Code replacement library** menu.
 - Set the `CodeReplacementLibrary` parameter at the command line or programmatically.
- 2 Configure the code generator to produce code only (not build an executable) so you can verify your code replacements before building an executable.
 - In the Configuration Parameters dialog box, on the **Code Generation** pane, select **Generate code only**.
 - Set the `GenCodeOnly` parameter at the command line or programmatically.

Include Code Replacement Information In Code Generation Report

If you have an Embedded Coder license, you can configure the code generator to include a code replacement section in the code generation report. The additional information can help you verify code replacements.

- 1 Configure the code generator to generate a report. In the Configuration Parameters dialog box, on the **Code Generation > Report** pane, select **Create code generation report**. Consider having the report open automatically. Select **Open report automatically**.
- 2 Include the code replacement section in the report. On the **Code Generation > Report** pane, select **Summarize which blocks triggered code replacements**.

Generate Replacement Code

Generate C/C++ code from the model and, if you configured the code generator accordingly, a code generation report. For example, on the **Code Generation > General** pane, click **Generate Code**.

The code generator produces the code and displays the report.

Verify Code Replacements

Verify code replacements by examining the generated code. It is possible that code replacement behaves differently than you expect. For example, data types that you observe in the code generator input might not match what the code generator uses as intermediate data types during an operation.

Related Examples

- “Choose a Code Replacement Library”
- “Code Generation Configuration”
- “Review and Test Code Replacements”

More About

- “Code replacement library”
- “Generate code only”
- “Create code generation report”
- “Open report automatically”

- “Summarize which blocks triggered code replacements”
- “What Is Code Replacement?”
- “Code You Can Replace From Simulink Models”
- “Code Replacement Libraries”
- “Code Replacement Terminology”
- “Code Replacement Limitations”

External Web Sites

- Supported Compilers

Choose a Code Replacement Library

In this section...

“About Choosing a Code Replacement Library” on page 17-32

“Explore Available Code Replacement Libraries” on page 17-32

“Explore Code Replacement Library Contents” on page 17-32

About Choosing a Code Replacement Library

By default, the code generator does not use a code replacement library.

If you are considering using a code replacement library:

- 1 Explore available libraries. Identify one that best meets your application needs.
 - Consider the lists of application code replacement requirements and libraries that MathWorks provides in “What Is Code Replacement?”.
 - See “Explore Available Code Replacement Libraries”.
- 2 Explore the contents of the library. See “Explore Code Replacement Library Contents”.

If you do not find a suitable library and you have an Embedded Coder license, you can create a custom code replacement library.

Explore Available Code Replacement Libraries

You can select the code replacement library to use for code generation on the “**Code Generation > Interface** pane” in the Configuration Parameters dialog box. To view a description of a library, select and hover your cursor over the library name. A tooltip describes the library and lists the tables that it contains. The tooltip lists the tables in the order that the code generator searches for a function or operator match.

Explore Code Replacement Library Contents

Use the Code Replacement Viewer to explore the content of a code replacement library.

- 1 At the command prompt, type `RTW.viewTf1`.

```
>> RTW.viewTf1
```

The viewer opens. To view the content of a specific library, specify the name of the library as an argument in single quotes. For example:

```
>> RTW.viewTf1('GNU C99 extensions')
```

- 2** In the left pane, select the name of a library. The viewer displays information about the library in the right pane.
- 3** In the left pane, expand the library, explore the list of tables it contains, and select a table from the list. In the middle pane, the viewer displays the function and operator entries that are in that table, along with abbreviated information for each entry.
- 4** In the middle pane, select a function or operator. The viewer displays information from the table entry in the right pane.

If you select an operator entry that specifies net slope fixed-point parameters (instantiated from entry class `RTW.Tf1COperationEntryGenerator` or `RTW.Tf1COperationEntryGenerator_NetSlope`), the viewer displays an additional tab that shows fixed-point settings.

See Code Replacement Viewer for details on what the viewer displays.

Related Examples

- [“Replace Code Generated from Simulink Models”](#)

More About

- [“What Is Code Replacement?”](#)
- [“Code You Can Replace From Simulink Models”](#)
- [“Code Replacement Libraries”](#)
- [“Code Replacement Terminology”](#)
- [“Code Replacement Limitations”](#)

Deployment

Desktops

Shared Object Libraries

In this section...

- “About Host-Based Shared Libraries” on page 18-2
- “Generate Shared Library Version of Model Code” on page 18-3
- “Create Application Code to Use Shared Library” on page 18-3
- “Host-Based Shared Library Limitations” on page 18-7

About Host-Based Shared Libraries

The Embedded Coder product provides an ERT target, `ert_shrplib.tlc`, for generating a host-based shared library from your Simulink model. Selecting this target allows you to generate a shared library version of your model code for your host platform, either a Windows dynamic link library (`.dll`) file or a UNIX shared object (`.so`) file. This feature can be used to package your source code for easy distribution and shared use. The generated `.dll` or `.so` file is shareable among different applications and upgradeable without having to recompile the applications that use it.

Code generation for the `ert_shrplib.tlc` target exports

- Variables and signals of type `ExportedGlobal` as data
- Real-time model structure (`model_M`) as data
- Functions essential to executing your model code

To view a list of symbols contained in a generated shared library file, you can

- On Windows, use the Dependency Walker utility, downloadable from <http://www.dependencywalker.com>
- On UNIX, use `nm -D model.so`

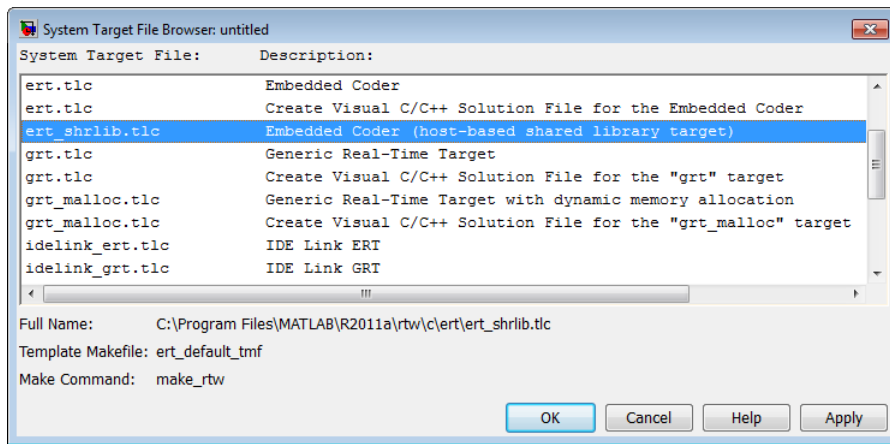
To generate and use a host-based shared library, you

- 1 Generate a shared library version of your model code
- 2 Create application code to load and use your shared library file

Generate Shared Library Version of Model Code

This section summarizes the steps to generate a shared library version of your model code.

- 1 To configure your model code for shared use by applications, open your model and select the `ert_shrplib.tlc` target on the **Code Generation** pane of the Configuration Parameters dialog box. Click **OK**.



Selecting the `ert_shrplib.tlc` target causes the build process to generate a shared library version of your model code into your current working folder. The selection does not change the code that is generated for your model.

- 2 Build the model.
- 3 After the build completes, you can examine the generated code in the model subfolder, and the `.dll` file or `.so` file that has been generated into your current folder.

Create Application Code to Use Shared Library

To illustrate how application code can load an ERT shared library file and access its functions and data, MathWorks provides the model `rtwdemo_shrplib`. Clicking the blue button in the model runs a script that:

- 1 Builds a shared library file from the model (for example, `rtwdemo_shrplib_win32.dll` on 32-bit Windows)

- 2 Compiles and links an example application, `rtwdemo_shrlib_app`, that will load and use the shared library file
- 3 Executes the example application

Note: It is recommended that you change directory to a new or empty folder before running the `rtwdemo_shrlib` script.

The model uses the following example application files, which are located in `matlabroot/toolbox/rtw/rtwdemos/shrlib_demo`.

File	Description
<code>rtwdemo_shrlib_app.h</code>	Example application header file
<code>rtwdemo_shrlib_app.c</code>	Example application that loads and uses the shared library file generated for the model
<code>run_rtwdemo_shrlib_app.m</code>	Script to compile, link, and execute the example application

You can view each of these files by clicking white buttons in the model window. Additionally, running the script places the relevant source and generated code files in your current folder. The files can be used as templates for writing application code for your own ERT shared library files.

The following sections present key excerpts of the example application files.

Example Application Header File

The example application header file `rtwdemo_shrlib_app.h` contains type declarations for the model's external input and output.

```
#ifndef _APP_MAIN_HEADER_
#define _APP_MAIN_HEADER_

typedef struct {
    int32_T Input;
} ExternalInputs_rtwdemo_shrlib;

typedef struct {
    int32_T Output;
} ExternalOutputs_rtwdemo_shrlib;

#endif /*_APP_MAIN_HEADER_*/
```

Example Application C Code

The example application `rtwdemo_shrllib_app.c` includes the following code for dynamically loading the shared library file. Notice that, depending on platform, the code invokes Windows or UNIX library commands.

```
#if (defined(_WIN32)||defined(_WIN64)) /* WINDOWS */
#include <windows.h>
#define GETSYMBOLADDR GetProcAddress
#define LOADLIB LoadLibrary
#define CLOSELIB FreeLibrary

#else /* UNIX */
#include <dlfcn.h>
#define GETSYMBOLADDR dlsym
#define LOADLIB dlopen
#define CLOSELIB dlclose

#endif

int main()
{
    void* handleLib;
    ...
    #if defined(_WIN64)
        handleLib = LOADLIB("./rtwdemo_shrllib_win64.dll");
    #else
    #if defined(_WIN32)
        handleLib = LOADLIB("./rtwdemo_shrllib_win32.dll");
    #else /* UNIX */
        handleLib = LOADLIB("./rtwdemo_shrllib.so", RTLD_LAZY);
    #endif
    #endif
    ...
    return(CLOSELIB(handleLib));
}
```

The following code excerpt shows how the C application accesses the model's exported data and functions. Notice the hooks for adding user-defined initialization, step, and termination code.

```
int32_T i;
...
void (*mdl_initialize)(boolean_T);
void (*mdl_step)(void);
void (*mdl_terminate)(void);

ExternalInputs_rtwdemo_shrllib (*mdl_Uptr);
ExternalOutputs_rtwdemo_shrllib (*mdl_Yptr);

uint8_T (*sum_outptr);
...
#if (defined(LCCDLL)||defined(BORLANDCDLL))
```

```

/* Exported symbols contain leading underscores when DLL is linked with
LCC or BORLANDC */
mdl_initialize = (void(*) (boolean_T))GETSYMBOLADDR(handleLib ,
    "_rtwdemo_shrplib_initialize");
mdl_step       = (void(*) (void))GETSYMBOLADDR(handleLib ,
    "_rtwdemo_shrplib_step");
mdl_terminate  = (void(*) (void))GETSYMBOLADDR(handleLib ,
    "_rtwdemo_shrplib_terminate");
mdl_Uptr       = (ExternalInputs_rtwdemo_shrplib*)GETSYMBOLADDR(handleLib ,
    "_rtwdemo_shrplib_U");
mdl_Yptr       = (ExternalOutputs_rtwdemo_shrplib*)GETSYMBOLADDR(handleLib ,
    "_rtwdemo_shrplib_Y");
sum_outptr     = (uint8_T*)GETSYMBOLADDR(handleLib , "_sum_out");
#else
mdl_initialize = (void(*) (boolean_T))GETSYMBOLADDR(handleLib ,
    "rtwdemo_shrplib_initialize");
mdl_step       = (void(*) (void))GETSYMBOLADDR(handleLib ,
    "rtwdemo_shrplib_step");
mdl_terminate  = (void(*) (void))GETSYMBOLADDR(handleLib ,
    "rtwdemo_shrplib_terminate");
mdl_Uptr       = (ExternalInputs_rtwdemo_shrplib*)GETSYMBOLADDR(handleLib ,
    "rtwdemo_shrplib_U");
mdl_Yptr       = (ExternalOutputs_rtwdemo_shrplib*)GETSYMBOLADDR(handleLib ,
    "rtwdemo_shrplib_Y");
sum_outptr     = (uint8_T*)GETSYMBOLADDR(handleLib , "sum_out");
#endif

if ((mdl_initialize && mdl_step && mdl_terminate && mdl_Uptr && mdl_Yptr &&
    sum_outptr)) {
    /* === user application initialization function === */
    mdl_initialize(1);
    /* insert other user defined application initialization code here */

    /* === user application step function === */
    for(i=0;i<=12;i++){
        mdl_Uptr->Input = i;
        mdl_step();
        printf("Counter out(sum_out): %d\tAmplifier in(Input): %d\tout(Output): %d\n",
            *sum_outptr, i, mdl_Yptr->Output);
        /* insert other user defined application step function code here */
    }

    /* === user application terminate function === */
    mdl_terminate();
    /* insert other user defined application termination code here */
}
else {
    printf("Cannot locate the specified reference(s) in the shared library.\n");
    return(-1);
}

```

Example Application Script

The application script `run_rtwdemo_shrlib_app.m` loads and rebuilds the model, and then compiles, links, and executes the model's shared library target file. You can view the script source file by opening `rtwdemo_shrlib` and clicking a white button to view source code. The script constructs platform-dependent command strings for compilation, linking, and execution that may apply to your development environment. To run the script, click the blue button.

Host-Based Shared Library Limitations

The following limitations apply to using ERT host-based shared libraries:

- Code generation for the `ert_shrlib.tlc` target exports only the following as data:
 - Variables and signals of type `ExportedGlobal`
 - Real-time model structure (*model_M*)
- Code generation for the `ert_shrlib.tlc` target supports the C language only (not C++). When you select the `ert_shrlib.tlc` target, language selection is greyed out on the **Code Generation** pane of the Configuration Parameters dialog box.
- On Windows systems, the `ert_shrlib` target by default does not generate or retain the `.lib` file for implicit linking (explicit linking is preferred for portability).

You can change the default behavior and retain the `.lib` file by modifying the corresponding template makefile (TMF). If you do this, the generated *model.h* file needs a small modification to be used together with the generated `ert_main.c` for implicit linking. For example, if you are using Visual C++, you need to declare `__declspec(dllimport)` in front of data to be imported implicitly from the shared library file.

- To reconstruct a model simulation using a generated host-based shared library, the application author must maintain the timing between system and shared library function calls in the original application. The timing needs to be consistent so that you can check the simulation and integration results.

Real-Time and Embedded Systems

- “Standalone Programs (No Operating System)” on page 19-2
- “Operating System Integration” on page 19-21
- “Processor Support Packages” on page 19-22

Standalone Programs (No Operating System)

In this section...

“About Standalone Program Execution” on page 19-2

“Generate a Standalone Program” on page 19-2

“Standalone Program Components” on page 19-3

“Main Program” on page 19-3

“rt_OneStep and Scheduling Considerations” on page 19-4

“Static Main Program Module” on page 19-10

“Rate Grouping Compliance and Compatibility Issues” on page 19-17

About Standalone Program Execution

By default, the Embedded Coder software generates *standalone* programs that do not require an external real-time executive or operating system. A standalone program requires minimal modification to be adapted to the target hardware. The standalone program architecture supports execution of models with either single or multiple sample rates.

Generate a Standalone Program

To generate a standalone program:

- 1 In the **Custom templates** section of the **Code Generation > Templates** pane of the Configuration Parameters dialog box, select the **Generate an example main program** option (which is on by default). This enables the **Target operating system** menu.
- 2 From the **Target operating system** menu, select **BareBoardExample** (the default selection).
- 3 Generate the code.

Different code is generated for multirate models depending on the following factors:

- Whether the model executes in single-tasking or multitasking mode.
- Whether or not reusable code is being generated.

These factors affect the scheduling algorithms used in generated code, and in some cases affect the API for the model entry point functions. The following sections discuss these variants.

Standalone Program Components

The core of a standalone program is the main loop. On each iteration, the main loop executes a background or null task and checks for a termination condition.

The main loop is periodically interrupted by a timer. The function `rt_OneStep` is either installed as a timer interrupt service routine (ISR), or called from a timer ISR at each clock step.

The execution driver, `rt_OneStep`, sequences calls to the `model_step` functions. The operation of `rt_OneStep` differs depending on whether the generating model is single-rate or multirate. In a single-rate model, `rt_OneStep` simply calls the `model_step` function. In a multirate model, `rt_OneStep` prioritizes and schedules execution of blocks according to the rates at which they run.

Main Program

- “Overview of Operation” on page 19-3
- “Guidelines for Modifying the Main Program” on page 19-4

Overview of Operation

The following pseudocode shows the execution of a main program.

```
main()
{
  Initialization (including installation of rt_OneStep as an
    interrupt service routine for a real-time clock)
  Initialize and start timer hardware
  Enable interrupts
  While(not Error) and (time < final time)
    Background task
  EndWhile
  Disable interrupts (Disable rt_OneStep from executing)
  Complete any background tasks
  Shutdown
}
```

The pseudocode is a design for a harness program to drive your model. The main program only partially implements this design. You must modify it according to your specifications.

Guidelines for Modifying the Main Program

This section describes the minimal modifications you should make in your production version of the main program module to implement your harness program.

- 1 Call `model_initialize`.
- 2 Initialize target-specific data structures and hardware, such as ADCs or DACs.
- 3 Install `rt_OneStep` as a timer ISR.
- 4 Initialize timer hardware.
- 5 Enable timer interrupts and start the timer.

Note `rtModel` is not in a valid state until `model_initialize` has been called. Servicing of timer interrupts should not begin until `model_initialize` has been called.

- 6 Optionally, insert background task calls in the main loop.
- 7 On termination of the main loop (if applicable):
 - Disable timer interrupts.
 - Perform target-specific cleanup such as zeroing DACs.
 - Detect and handle errors. Note that even if your program is designed to run indefinitely, you may need to handle severe error conditions, such as timer interrupt overruns.

You can use the macros `rtmGetErrorStatus` and `rtmSetErrorStatus` to detect and signal errors.

rt_OneStep and Scheduling Considerations

- “Overview of Operation” on page 19-5
- “Single-Rate Single-Tasking Operation” on page 19-5
- “Multirate Multitasking Operation” on page 19-6
- “Multirate Single-Tasking Operation” on page 19-8
- “Guidelines for Modifying `rt_OneStep`” on page 19-9

Overview of Operation

The operation of `rt_OneStep` depends upon

- Whether your model is single-rate or multirate. In a single-rate model, the sample times of all blocks in the model, and the model's fixed step size, are the same. A model in which the sample times and step size do not meet these conditions is termed multirate.
- Your model's solver mode (`SingleTasking` versus `MultiTasking`)

Permitted Solver Modes for Embedded Coder Targeted Models summarizes the permitted solver modes for single-rate and multirate models. Note that for a single-rate model, only `SingleTasking` solver mode is allowed.

Permitted Solver Modes for Embedded Coder Targeted Models

Mode	Single-Rate	Multirate
<code>SingleTasking</code>	Allowed	Allowed
<code>MultiTasking</code>	Disallowed	Allowed
<code>Auto</code>	Allowed (defaults to <code>SingleTasking</code>)	Allowed (defaults to <code>MultiTasking</code>)

The generated code for `rt_OneStep` (and associated timing data structures and support functions) is tailored to the number of rates in the model and to the solver mode. The following sections discuss each possible case.

Single-Rate Single-Tasking Operation

The only valid solver mode for a single-rate model is `SingleTasking`. Such models run in “single-rate” operation.

The following pseudocode shows the design of `rt_OneStep` in a single-rate program.

```
rt_OneStep()
{
    Check for interrupt overflow or other error
    Enable "rt_OneStep" (timer) interrupt
    Model_Step() -- Time step combines output, logging, update
}
```

For the single-rate case, the generated `model_step` function is

```
void model_step(void)
```

Single-rate `rt_OneStep` is designed to execute `model_step` within a single clock period. To enforce this timing constraint, `rt_OneStep` maintains and checks a timer overrun flag. On entry, timer interrupts are disabled until the overrun flag and other error conditions have been checked. If the overrun flag is clear, `rt_OneStep` sets the flag, and proceeds with timer interrupts enabled.

The overrun flag is cleared only upon successful return from `model_step`. Therefore, if `rt_OneStep` is reinterrupted before completing `model_step`, the reinterrupt is detected through the overrun flag.

Reinterruption of `rt_OneStep` by the timer is an error condition. If this condition is detected `rt_OneStep` signals an error and returns immediately. (Note that you can change this behavior if you want to handle the condition differently.)

Note that the design of `rt_OneStep` assumes that interrupts are disabled before `rt_OneStep` is called. `rt_OneStep` should be noninterruptible until the interrupt overflow flag has been checked.

Multirate Multitasking Operation

In a multirate multitasking system, code generation uses a prioritized, preemptive multitasking scheme to execute the different sample rates in your model.

The following pseudocode shows the design of `rt_OneStep` in a multirate multitasking program.

```
rt_OneStep()  
{  
  Check for base-rate interrupt overrun  
  Enable "rt_OneStep" interrupt  
  Determine which rates need to run this time step  
  
  Model_Step0()      -- run base-rate time step code  
  
  For N=1:NumTasks-1  -- iterate over sub-rate tasks  
    If (sub-rate task N is scheduled)  
      Check for sub-rate interrupt overrun  
      Model_StepN()  -- run sub-rate time step code  
    EndIf  
  EndFor  
}
```

Task Identifiers

The execution of blocks having different sample rates is broken into tasks. Each block that executes at a given sample rate is assigned a *task identifier* (`tid`), which associates it with a task that executes at that rate. Where there are `NumTasks` tasks in the system, the range of task identifiers is `0..NumTasks-1`.

Prioritization of Base-Rate and Subrate Tasks

Tasks are prioritized, in descending order, by rate. The *base-rate* task is the task that runs at the fastest rate in the system (the hardware clock rate). The base-rate task has highest priority (`tid 0`). The next fastest task (`tid 1`) has the next highest priority, and so on down to the slowest, lowest priority task (`tid NumTasks-1`).

The slower tasks, running at multiples of the base rate, are called *subrate* tasks.

Rate Grouping and Rate-Specific `model_step` Functions

In a single-rate model, the block output computations are performed within a single function, `model_step`. For multirate, multitasking models, the code generator tries to use a different strategy. This strategy is called *rate grouping*. Rate grouping generates separate `model_step` functions for the base rate task and each subrate task in the model. The function naming convention for these functions is

`model_stepN`

where `N` is a task identifier. For example, for a model named `my_model` that has three rates, the following functions are generated:

```
void my_model_step0 (void);
void my_model_step1 (void);
void my_model_step2 (void);
```

Each `model_stepN` function executes the blocks sharing `tid N`; in other words, the block code that executes within task `N` is grouped into the associated `model_stepN` function.

Scheduling `model_stepN` Execution

On each clock tick, `rt_OneStep` maintains scheduling counters and *event flags* for each subrate task. The counters are implemented as `taskCounter` arrays indexed on `tid`. The event flags are implemented as arrays indexed on `tid`.

The scheduling counters and task flags for sub-rates are maintained by `rt_OneStep`. The scheduling counters are basically clock rate dividers that count up the sample period

associated with each sub-rate task. A pair of tasks that exchanges data maintains an interaction flag at the faster rate. Task interaction flags indicate that both fast and slow tasks are scheduled to run.

The event flags indicate whether or not a given task is scheduled for execution. `rt_OneStep` maintains the event flags based on a task counter that is maintained by code in the main program module for the model. When a counter indicates that a task's sample period has elapsed, the main code sets the event flag for that task.

On each invocation, `rt_OneStep` updates its scheduling data structures and steps the base-rate task (`rt_OneStep` calls `model_step0` because the base-rate task must execute on every clock step). Then, `rt_OneStep` iterates over the scheduling flags in `tid` order, unconditionally calling `model_stepN` for any task whose flag is set. The tasks are executed in order of priority.

Preemption

Note that the design of `rt_OneStep` assumes that interrupts are disabled before `rt_OneStep` is called. `rt_OneStep` should be noninterruptible until the base-rate interrupt overflow flag has been checked (see pseudocode above).

The event flag array and loop variables used by `rt_OneStep` are stored as local (stack) variables. Therefore, `rt_OneStep` is reentrant. If `rt_OneStep` is reinterrupted, higher priority tasks preempt lower priority tasks. Upon return from interrupt, lower priority tasks resume in the previously scheduled order.

Overrun Detection

Multirate `rt_OneStep` also maintains an array of timer overrun flags. `rt_OneStep` detects timer overrun, per task, by the same logic as single-rate `rt_OneStep`.

Note If you have developed multirate S-functions, or if you use a customized static main program module, see “Rate Grouping Compliance and Compatibility Issues” on page 19-17 for information about how to adapt your code for rate grouping compatibility. This adaptation lets your multirate, multitasking models generate more efficient code.

Multirate Single-Tasking Operation

In a multirate single-tasking program, by definition, sample times in the model must be an integer multiple of the model's fixed-step size.

In a multirate single-tasking program, blocks execute at different rates, but under the same task identifier. The operation of `rt_OneStep`, in this case, is a simplified version of multirate multitasking operation. Rate grouping is not used. The only task is the base-rate task. Therefore, only one `model_step` function is generated:

```
void model_step(void)
```

On each clock tick, `rt_OneStep` checks the overrun flag and calls `model_step`. The scheduling function for a multirate single-tasking program is `rate_scheduler` (rather than `rate_monotonic_scheduler`). The scheduler maintains scheduling counters on each clock tick. There is one counter for each sample rate in the model. The counters are implemented in an array (indexed on `tid`) within the `Timing` structure within `rtModel`.

The counters are clock rate dividers that count up the sample period associated with each subrate task. When a counter indicates that a sample period for a given rate has elapsed, `rate_scheduler` clears the counter. This condition indicates that blocks running at that rate should execute on the next call to `model_step`, which is responsible for checking the counters.

Guidelines for Modifying `rt_OneStep`

`rt_OneStep` does not require extensive modification. The only required modification is to reenable interrupts after the overrun flags and error conditions have been checked. If applicable, you should also

- Save and restore your FPU context on entry and exit to `rt_OneStep`.
- Set model inputs associated with the base rate before calling `model_step0`.
- Get model outputs associated with the base rate after calling `model_step0`.

Note: If you modify `rt_OneStep` to read a value from a continuous output port after each base-rate model step, see the relevant cautionary guideline below.

- In a multirate, multitasking model, set model inputs associated with subrates before calling `model_stepN` in the subrate loop.
- In a multirate, multitasking model, get model outputs associated with subrates after calling `model_stepN` in the subrate loop.

Comments in `rt_OneStep` indicate the place to add your code.

In multirate `rt_OneStep`, you can improve performance by unrolling `for` and `while` loops.

In addition, you may choose to modify the overrun behavior to continue execution after error recovery is complete.

Also observe the following cautionary guidelines:

- You should not modify the way in which the counters, event flags, or other timing data structures are set in `rt_OneStep`, or in functions called from `rt_OneStep`. The `rt_OneStep` timing data structures (including `rtModel`) and logic are critical to the operation of the generated program.
- If you have customized the main program module to read model outputs after each base-rate model step, be aware that selecting model options **Support: continuous time** and **Single output/update function** together may cause output values read from `main` for a continuous output port to differ slightly from the corresponding output values in the model's logged data. This is because, while logged data is a snapshot of output at major time steps, output read from `main` after the base-rate model step potentially reflects intervening minor time steps. To eliminate the discrepancy, either separate the generated output and update functions (clear the **Single output/update function** option) or place a Zero-Order Hold block before the continuous output port.

Static Main Program Module

- “Overview” on page 19-10
- “Rate Grouping and the Static Main Program” on page 19-11
- “Modify the Static Main Program” on page 19-13
- “Modify Static Main to Allocate and Access Model Instance Data” on page 19-14

Overview

In most cases, the easiest strategy for deploying generated code is to use the **Generate an example main program option** to generate the `ert_main.c` or `.cpp` module (see “Generate a Standalone Program” on page 19-2).

However, if you turn the **Generate an example main program** option off, you can use a static main module as an example or template for developing your embedded applications. Static main modules provided by MathWorks include:

- `matlabroot/rtw/c/src/common/rt_main.c` — Supports Nonreusable function code interface packaging.

- `matlabroot/rtw/c/src/common/rt_malloc_main.c` — Supports Reusable function code interface packaging. The model option **Use dynamic memory allocation for model initialization** must be on and model parameter **Pass root-level I/O as** must be set to Part of model data structure.
- `matlabroot/rtw/c/src/common/rt_cppclass_main.cpp` — Supports C++ class code interface packaging.

The static main module is not part of the generated code; it is provided as a basis for your custom modifications, and for use in simulation. If your existing applications depend upon a static `ert_main.c` (developed in releases before R2012b), `rt_main.c`, `rt_malloc_main.c`, or `rt_cppclass_main.cpp`, you may need to continue using a static main program module.

When developing applications using a static main module, you should copy the module to your working folder and rename it before making modifications. For example, you could rename `rt_main.c` to `model_rt_main.c`. Also, you must modify the template makefile or toolchain settings such that the build process creates a corresponding object file, such as `model_rt_main.obj` (on UNIX, `model_rt_main.o`), in the build folder.

The static main module contains

- `rt_OneStep`, a timer interrupt service routine (ISR). `rt_OneStep` calls `model_step` to execute processing for one clock period of the model.
- A skeletal `main` function. As provided, `main` is useful in simulation only. You must modify `main` for real-time interrupt-driven execution.

For single-rate models, the operation of `rt_OneStep` and the main function are essentially the same in the static main module as they are in the autogenerated version described in “About Standalone Program Execution” on page 19-2. For multirate, multitasking models, however, the static and generated code are slightly different. The next section describes this case.

Rate Grouping and the Static Main Program

Targets based on the ERT target sometimes use a static main module and disallow use of the **Generate an example main program** option. This is done because target-specific modifications have been added to the static main module, and these modifications would not be preserved if the main program were regenerated.

Your static main module may or may not use rate grouping compatible `model_stepN` functions. If your main module is based on the static `rt_main.c`, `rt_malloc_main.c`,

or `rt_cppclass_main.cpp` module, it does not use rate-specific `model_stepN` function calls. It uses the old-style `model_step` function, passing in a task identifier:

```
void model_step(int_T tid);
```

By default, when the **Generate an example main program** option is off, the ERT target generates a `model_step` “wrapper” for multirate, multitasking models. The purpose of the wrapper is to interface the rate-specific `model_stepN` functions to the old-style call. The wrapper code dispatches to the `model_stepN` call with a `switch` statement, as in the following example:

```
void mymodel_step(int_T tid) /* Sample time: */
{
    switch(tid) {
        case 0 :
            mymodel_step0();
            break;
        case 1 :
            mymodel_step1();
            break;
        case 2 :
            mymodel_step2();
            break;
        default :
            break;
    }
}
```

The following pseudocode shows how `rt_OneStep` calls `model_step` from the static main program in a multirate, multitasking model.

```
rt_OneStep()
{
    Check for base-rate interrupt overflow
    Enable "rt_OneStep" interrupt
    Determine which rates need to run this time step

    ModelStep(tid=0)    --base-rate time step

    For N=1:NumTasks-1 -- iterate over sub-rate tasks
        Check for sub-rate interrupt overflow
        If (sub-rate task N is scheduled)
            ModelStep(tid=N)    --sub-rate time step
```

```

    EndIf
  EndFor
}

```

You can use the TLC variable `RateBasedStepFcn` to specify that only the rate-based step functions are generated, without the wrapper function. If your target calls the rate grouping compatible `model_stepN` function directly, set `RateBasedStepFcn` to 1. In this case, the wrapper function is not generated.

You should set `RateBasedStepFcn` prior to the `%include "codegenentry.tlc"` statement in your system target file. Alternatively, you can set `RateBasedStepFcn` in your `target_settings.tlc` file.

Modify the Static Main Program

As with the generated `ert_main.c` or `.cpp`, you should make a few modifications to the main loop and `rt_OneStep`. See “Guidelines for Modifying the Main Program” on page 19-4 and “Guidelines for Modifying `rt_OneStep`” on page 19-9.

Also, you should replace the `rt_OneStep` call in the main loop with a background task call or null statement.

Other modifications you may need to make are

- If applicable, follow comments in the code regarding where to add code for reading/writing model I/O and saving/restoring FPU context.

Note: If you modify `rt_main.c`, `rt_malloc_main.c`, or `rt_cppclass_main.cpp` to read a value from a continuous output port after each base-rate model step, see the relevant cautionary guideline in “Guidelines for Modifying `rt_OneStep`” on page 19-9.

- When the **Generate an example main program** option is off, `rtmodel.h` is generated to provide an interface between the main module and generated model code. If you create your own static main program module, you would normally include `rtmodel.h`.

Alternatively, you can suppress generation of `rtmodel.h`, and include `model.h` directly in your main module. To suppress generation of `rtmodel.h`, use the following statement in your system target file:

```
%assign AutoBuildProcedure = 0
```

- If you have cleared the **Terminate function required** option, remove or comment out the following in your production version of `rt_main.c`, `rt_malloc_main.c`, or `rt_cppclass_main.cpp`:
 - The `#if TERMFCN...` compile-time error check
 - The call to `MODEL_TERMINATE`
- For `rt_main.c` (not applicable to `rt_cppclass_main.cpp`): If you do *not* want to combine output and update functions, clear the **Single output/update function** option and make the following changes in your production version of `rt_main.c`:
 - Replace calls to `MODEL_STEP` with calls to `MODEL_OUTPUT` and `MODEL_UPDATE`.
 - Remove the `#if ONESTEPFCN...` error check.
- The static `rt_main.c` module does not support **Reusable function code interface** packaging. The following error check raises a compile-time error if **Reusable function code interface** packaging is used illegally.

```
#if MULTI_INSTANCE_CODE==1
```

Modify Static Main to Allocate and Access Model Instance Data

If you are using a static main program module, and your model is configured for **Reusable function code interface** packaging, but the model option **Use dynamic memory allocation for model initialization** is not selected, model instance data must be allocated either statically or dynamically by the calling main code. Pointers to the individual model data structures (such as Block IO, DWork, and Parameters) must be set up in the top-level real-time model data structure.

To support main modifications, the build process generates a subset of the following real-time model (RTM) macros, based on the data requirements of your model, into `model.h`.

RTM Macro Syntax	Description
<code>rtmGetBlockIO(rtm)</code>	Get the block I/O data structure
<code>rtmSetBlockIO(rtm, val)</code>	Set the block I/O data structure
<code>rtmGetContStates(rtm)</code>	Get the continuous states data structure
<code>rtmSetContStates(rtm, val)</code>	Set the continuous states data structure
<code>rtmGetDefaultParam(rtm)</code>	Get the default parameters data structure
<code>rtmSetDefaultParam(rtm, val)</code>	Set the default parameters data structure
<code>rtmGetPrevZCSigState(rtm)</code>	Get the previous zero-crossing signal state data structure

RTM Macro Syntax	Description
<code>rtmSetPrevZCSigState(rtm, val)</code>	Set the previous zero-crossing signal state data structure
<code>rtmGetRootDWork(rtm)</code>	Get the DWork data structure
<code>rtmSetRootDWork(rtm, val)</code>	Set the DWork data structure
<code>rtmGetU(rtm)</code>	Get the root inputs data structure (when root inputs are passed as part of the model data structure)
<code>rtmSetU(rtm, val)</code>	Set the root inputs data structure (when root inputs are passed as part of the model data structure)
<code>rtmGetY(rtm)</code>	Get the root outputs data structure (when root outputs are passed as part of the model data structure)
<code>rtmSetY(rtm, val)</code>	Set the root outputs data structure (when root outputs are passed as part of the model data structure)

Use these macros in your static main program to access individual model data structures within the RTM data structure. For example, suppose that the example model `rtwdemo_reusable` is configured with **Reusable** function code interface packaging, **Use dynamic memory allocation for model initialization** cleared, **Pass root-level I/O** as set to **Individual** arguments, and **Optimization** pane option **Remove root level I/O zero initialization** cleared. Building the model generates the following model data structures and model entry-points into `rtwdemo_reusable.h`:

```

/* Block states (auto storage) for system '<Root>' */
typedef struct {
    real_T Delay_DSTATE;          /* '<Root>/Delay' */
} D_Work;

/* Parameters (auto storage) */
struct Parameters_ {
    real_T k1;                    /* Variable: k1
                                 * Referenced by: '<Root>/Gain'
                                 */
};

/* Real-time Model Data Structure */
struct tag_RTM {
    /*
     * ModelData:
     * The following substructure contains information regarding
     * the data used in the model.
     */
    struct {
        Parameters *defaultParam;
        D_Work *dwork;
    } ModelData;
};

```

```
};

/* Model entry point functions */
extern void rtwdemo_reusable_initialize(RT_MODEL *const rtM, real_T *rtU_In1,
    real_T *rtU_In2, real_T *rtY_Out1);
extern void rtwdemo_reusable_step(RT_MODEL *const rtM, real_T rtU_In1, real_T
    rtU_In2, real_T *rtY_Out1);
```

Additionally, if **Generate an example main program** is not selected for the model, `rtwdemo_reusable.h` contains definitions for the RTM macros `rtmGetDefaultParam`, `rtmsetDefaultParam`, `rtmGetRootDWork`, and `rtmSetRootDWork`.

Also, for reference, the generated `rtmodel.h` file contains an example parameter definition with initial values (non-executing code):

```
#if 0

/* Example parameter data definition with initial values */
static Parameters rtP = {
    2.0                                /* Variable: k1
    * Referenced by: '<Root>/Gain'
    */
};                                     /* Modifiable parameters */

#endif
```

In the definitions section of your static main file, you could use the following code to statically allocate the real-time model data structures and arguments for the `rtwdemo_reusable` model:

```
static RT_MODEL rtM_;
static RT_MODEL *const rtM = &rtM_; /* Real-time model */
static Parameters rtP = {
    2.0                                /* Variable: k1
    * Referenced by: '<Root>/Gain'
    */
};                                     /* Modifiable parameters */

static D_Work rtDWork;                /* Observable states */

/* '<Root>/In1' */
static real_T rtU_In1;

/* '<Root>/In2' */
static real_T rtU_In2;

/* '<Root>/Out1' */
static real_T rtY_Out1;
```

In the body of your main function, you could use the following RTM macro calls to set up the model parameters and `DWork` data in the real-time model data structure:

```

int_T main(int_T argc, const char *argv[])
{
    ...
    /* Pack model data into RTM */

    rtmSetDefaultParam(rtm, &rtP);
    rtmSetRootDWork(rtm, &rtDWork);

    /* Initialize model */
    rtwdemo_reusable_initialize(rtm, &rtU_In1, &rtU_In2, &rtY_Out1);
    ...
}

```

Follow a similar approach to set up multiple instances of model data, where the real-time model data structure for each instance has its own data. In particular, the parameter structure (rtP) should be initialized, for each instance, to the desired values, either statically as part of the rtP data definition or at run time.

Rate Grouping Compliance and Compatibility Issues

- “Main Program Compatibility” on page 19-17
- “Make Your S-Functions Rate Grouping Compliant” on page 19-17

Main Program Compatibility

When the **Generate an example main program** option is off, code generation produces slightly different rate grouping code, for compatibility with the older static `ert_main.c` module. See “Rate Grouping and the Static Main Program” on page 19-11 for details.

Make Your S-Functions Rate Grouping Compliant

Built-in Simulink blocks, as well as DSP System Toolbox blocks, are compliant with the requirements for generating rate grouping code. However, user-written multirate inlined S-functions may not be rate grouping compliant. Noncompliant blocks generate less efficient code, but are otherwise compatible with rate grouping. To take full advantage of the efficiency of rate grouping, your multirate inlined S-functions must be upgraded to be fully rate grouping compliant. You should upgrade your TLC S-function implementations, as described in this section.

Use of noncompliant multirate blocks to generate rate-grouping code generates dead code. This can cause two problems:

- Reduced code efficiency.

- Warning messages issued at compile time. Such warnings are caused when dead code references temporary variables before initialization. Since the dead code does not run, this problem does not affect the run-time behavior of the generated code.

To make your S-functions rate grouping compliant, you can use the following TLC functions to generate `ModelOutputs` and `ModelUpdate` code, respectively:

```
OutputsForTID(block, system, tid)
UpdateForTID(block, system, tid)
```

The code listings below illustrate generation of output computations without rate grouping (Listing 1) and with rate grouping (Listing 2). Note the following:

- The `tid` argument is a task identifier (`0..NumTasks-1`).
- Only code guarded by the `tid` passed in to `OutputsForTID` is generated. The `if (%<LibIsSFCnSampleHit(portName)>)` test is not used in `OutputsForTID`.
- When generating rate grouping code, `OutputsForTID` and/or `UpdateForTID` is called during code generation. When generating non-rate-grouping code, `Outputs` and/or `Update` is called.
- In rate grouping compliant code, the top-level `Outputs` and/or `Update` functions call `OutputsForTID` and/or `UpdateForTID` functions for each rate (`tid`) involved in the block. The code returned by `OutputsForTID` and/or `UpdateForTID` must be guarded by the corresponding `tid` guard:

```
if (%<LibIsSFCnSampleHit(portName)>)
```

as in Listing 2.

Listing 1: Outputs Code Generation Without Rate Grouping

```
%% multirate_blk.tlc

%implements "multirate_blk" "C"

%% Function: mdlOutputs =====
%% Abstract:
%%
%% Compute the two outputs (input signal decimated by the
%% specified parameter). The decimation is handled by sample times.
%% The decimation is only performed if the block is enabled.
%% Each ports has a different rate.
%%
%% Note, the usage of the enable should really be protected such that
%% Neach task has its own enable state. In this example, the enable
%% occurs immediately which may or may not be the expected behavior.
```



```

%%
%function Outputs(block, system) Output
/* %<Type> Block: %<Name> */
%assign enable = LibBlockInputSignal(0, "", "", 0)
{
  int_T *enabled = &%<LibBlockIWork(0, "", "", 0)>;

  %if LibGetSFcnTIDType("InputPortIdx0") == "continuous"
    %% Only check the enable signal on a major time step.
    if (%<LibIsMajorTimeStep()> && ...
        %<LibIsSFcnSampleHit("InputPortIdx0")>) {
      *enabled = (%<enable> > 0.0);
    }
  %else
    if (%<LibIsSFcnSampleHit("InputPortIdx0")>) {
      *enabled = (%<enable> > 0.0);
    }
  %endif

  if (*enabled) {
    %assign signal = LibBlockInputSignal(1, "", "", 0)
    if (%<LibIsSFcnSampleHit("OutputPortIdx0")>) {
      %assign y = LibBlockOutputSignal(0, "", "", 0)
      %<y> = %<signal>;
    }
    if (%<LibIsSFcnSampleHit("OutputPortIdx1")>) {
      %assign y = LibBlockOutputSignal(1, "", "", 0)
      %<y> = %<signal>;
    }
  }
}

%endfunction
%% [EOF] sfun_multirate.tlc

```

Listing 2: Outputs Code Generation With Rate Grouping

```

%% example_multirateblk.tlc

%implements "example_multirateblk" "C"

%% Function: mdlOutputs =====
%% Abstract:
%%
%% Compute the two outputs (the input signal decimated by the
%% specified parameter). The decimation is handled by sample times.
%% The decimation is only performed if the block is enabled.
%% All ports have different sample rate.
%%
%% Note: the usage of the enable should really be protected such that
%% each task has its own enable state. In this example, the enable
%% occurs immediately which may or may not be the expected behavior.
%%
%function Outputs(block, system) Output

```

```
%assign portIdxName = ["InputPortIdx0","OutputPortIdx0","OutputPortIdx1"]
%assign portTID      = [%<LibGetGlobalTIDFromLocalSFcnTID("InputPortIdx0")>, ...
                      %<LibGetGlobalTIDFromLocalSFcnTID("OutputPortIdx0")>, ...
                      %<LibGetGlobalTIDFromLocalSFcnTID("OutputPortIdx1")>]

%foreach i = 3
    %assign portName = portIdxName[i]
    %assign tid      = portTID[i]
    if (%<LibIsSFcnSampleHit(portName)>) {
        %<OutputsForTID(block,system,tid)>
    }
%endforeach

%endfunction

%function OutputsForTID(block, system, tid) Output
/* %<Type> Block: %<Name> */
%assign enable = LibBlockInputSignal(0, "", "", 0)
%assign enabled = LibBlockIWork(0, "", "", 0)
%assign signal = LibBlockInputSignal(1, "", "", 0)

%switch(tid)
    %case LibGetGlobalTIDFromLocalSFcnTID("InputPortIdx0")
        %if LibGetSFcnTIDType("InputPortIdx0") == "continuous"
            %% Only check the enable signal on a major time step.
            if (%<LibIsMajorTimeStep()>) {
                %<enabled> = (%<enable> > 0.0);
            }
        %else
            %<enabled> = (%<enable> > 0.0);
        %endif
        %break
    %case LibGetGlobalTIDFromLocalSFcnTID("OutputPortIdx0")
        if (%<enabled>) {
            %assign y = LibBlockOutputSignal(0, "", "", 0)
            %<y> = %<signal>;
        }
        %break
    %case LibGetGlobalTIDFromLocalSFcnTID("OutputPortIdx1")
        if (%<enabled>) {
            %assign y = LibBlockOutputSignal(1, "", "", 0)
            %<y> = %<signal>;
        }
        %break
    %default
        %<error it out>
%endswitch

%endfunction

%% [EOF] sfun_multirate.tlc
```

Operating System Integration

Embedded Coder supports integration for Linux, Texas Instruments™ DSP/BIOS™, and Wind River VxWorks. For details, see “Embedded Systems”.

Processor Support Packages

Embedded Coder supports integration for specific processors. For details, see “Embedded Systems”.

Export Code Generated from Model to External Application

- “Export Function-Call Subsystems” on page 20-2
- “Control Generation of Function Prototypes” on page 20-12
- “C++ Class Interface Control” on page 20-14

Export Function-Call Subsystems

In this section...

- “Exporting Function-Call Subsystems” on page 20-2
- “Requirements for Exporting Function-Call Subsystems” on page 20-3
- “Techniques for Exporting Function-Call Subsystems” on page 20-5
- “Optimize Exported Function-Call Subsystems” on page 20-7
- “Export Functions That Depend on Elapsed Time” on page 20-7
- “Function-Call Subsystem Export” on page 20-8
- “Function-Call Subsystems Export Limitations” on page 20-11

Exporting Function-Call Subsystems

Embedded Coder software provides code export capabilities that you can use to

- Automatically generate code for
 - A function-call subsystem that contains only blocks that support code generation
 - A virtual subsystem that contains only such subsystems and a few other types of blocks
- Create a SIL block that represents the generated code

You can use these capabilities only if the subsystem and its interface to the Simulink model conform to certain requirements and constraints, as described in “Requirements for Exporting Function-Call Subsystems” on page 20-3. For limitations that apply, see “Function-Call Subsystems Export Limitations” on page 20-11. To see an example of exported function-call subsystems, type `rtwdemo_exporting_functions` in the MATLAB Command Window.

Note: For models designed in earlier releases, Embedded Coder software also supports the ability to export functions from triggered subsystems. In general, the requirements and limitations stated for exporting functions from function-call subsystems also apply to exporting functions from triggered subsystems, with the following exceptions:

- Triggered subsystems from which you intend to export functions must be encapsulated in a single top-level virtual subsystem.

- Triggered subsystems do not have to meet the requirements in “Trigger Signals Require a Common Source” on page 20-4 and “Requirements for Exported Virtual Subsystems” on page 20-4.
 - The section “Export Functions That Depend on Elapsed Time” on page 20-7 is not applicable to exporting functions from triggered subsystems.
-

Additional Information

See the following in the Simulink documentation for additional information relating to exporting function-call subsystems:

- “Systems and Subsystems”
- “Signals”
- “Create a Triggered Subsystem”
- “Function-Call Subsystems and S-Functions”
- “Host-Specific Code”

If you want to use Stateflow blocks to trigger exportable function-call subsystems, you may also need information from “Interface with Simulink”.

Requirements for Exporting Function-Call Subsystems

To be exportable as code, a function-call subsystem, or a virtual subsystem that contains such subsystems, must meet certain requirements. Most requirements are similar for either type of export, but some apply only to virtual subsystems. The requirements for Simulink code generation also apply.

For brevity, *exported subsystem* in this section means only an exported function-call subsystem or an exported virtual subsystem that contains such subsystems. The requirements listed do not necessarily apply to other types of exported subsystems.

Requirements for Exported Subsystems

These requirements apply to both exported function-call subsystems and exported virtual subsystems that contain such subsystems.

Blocks Must Support Code Generation

All blocks within an exported subsystem must support code generation. However, blocks outside the subsystem need not support code generation unless they will be converted to code in some other context.

Blocks Must Not Use Absolute Time

Certain blocks use absolute time. Blocks that use absolute time are not supported in exported function-call subsystems. For a complete list of such blocks, see “Limitations on the Use of Absolute Time” in the Simulink Coder documentation.

Blocks Must Not Depend on Elapsed Time

Certain blocks, like the Sine Wave block and Discrete Integrator block, depend on elapsed time. If an exported function-call subsystem contains a block that depends on elapsed time, the subsystem must specify periodic execution. See “Export Functions That Depend on Elapsed Time” on page 20-7 in the Simulink Coder documentation.

Trigger Signals Require a Common Source

If more than one trigger signal crosses the boundary of an exported system, all of the trigger signals must be periodic and originate from the same function-call initiator.

Trigger Signals Must Be Scalar

A trigger signal that crosses the boundary of an exported subsystem must be scalar. Input and output data signals that do not act as triggers need not be scalar.

Data Signals Must Be Nonvirtual

A data signal that crosses the boundary of an exported system cannot be a virtual bus, and cannot be implemented as a `Goto-From` connection. Every data signal crossing the export boundary must be scalar, muxed, or a nonvirtual bus.

Requirements for Exported Virtual Subsystems

These requirements apply only to exported virtual subsystems that contain function-call subsystems.

Virtual Subsystem Must Use Only Permissible Blocks

The top level of an exported virtual subsystem that contains function-call subsystem blocks can contain only the following other types of blocks:

- Input and Output blocks (ports)

- Constant blocks (including blocks that resolve to constants, such as Add)
- Merge blocks
- Virtual connection blocks (Mux, Demux, Bus Creator, Bus Selector, Signal Specification)
- Signal-viewer blocks, such as **Scope** blocks

These restrictions do *not* apply within function-call subsystems, whether or not they appear in a virtual subsystem. They apply only at the top level of an exported virtual subsystem that contains one or more function-call subsystems.

Constant Blocks Must Be Inlined

When a constant block appears at the top level of an exported virtual subsystem, the containing model must check **Inline parameters** on the **Optimization > Signals and Parameters** pane of the Configuration Parameters dialog box.

Constant Outputs Must Specify a Storage Class

When a constant signal drives an output port of an exported virtual subsystem, the signal must specify a storage class.

Techniques for Exporting Function-Call Subsystems

- “General Workflow” on page 20-5
- “Specify a Custom Initialize Function Name” on page 20-6
- “Specify a Custom Description” on page 20-6

General Workflow

To export a function-call subsystem, or a virtual subsystem that contains function-call subsystems,

- 1 Check that the subsystem to be exported satisfies the “Requirements for Exporting Function-Call Subsystems” on page 20-3.
- 2 In the Configuration Parameters dialog box:
 - a On the **Code Generation** pane, specify an ERT code generation target such as `ert.tlc`.
 - b If you want a SIL block with the generated code, go to the **Verification** pane and, from the **Create block** drop-down list, select **SIL**.
 - c Click **OK** or **Apply**.

- 3 Right-click the subsystem block and choose **C/C++ Code > Export Functions** from the context menu.

The **Build code for subsystem: *Subsystem*** dialog box appears. This dialog box is not specific to exporting function-call subsystems, and generating code does not require entering information in the box.

- 4 Click **Build**.

The MATLAB Command Window displays messages similar to the code generation sequence. Simulink generates code and places it in the working folder.

If you set **Create block** to **SIL** in step 2b, Simulink opens a new window that contains an S-function block that represents the generated code. This block has the same size, shape, and connectors as the original subsystem.

Code generation and optional block creation are now complete. You can test and use the code and optional block as you could for generated ERT code and S-function block.

Specify a Custom Initialize Function Name

You can specify a custom name for the initialize function of your exported function as an argument to the `rtwbuild` command. When used for this purpose, the command takes the following form:

```
blockHandle = rtwbuild('subsystem', 'Mode', 'ExportFunctionCalls',...  
                    'ExportFunctionInitializeFunctionName', 'fcname')
```

where *fcname* specifies the desired function name. For example, if you specify the name 'myinitfcn', the build process emits code similar to the following:

```
/* Model initialize function */  
void myinitfcn(void){  
    ...  
}
```

Specify a Custom Description

You can enter a custom description for an exported function using the Block Properties dialog box of an Inport block. To do this, go to the subsystem that is to be exported as a function, right-click on the Inport block that drives the control port of the subsystem, and select **Properties**. In the **General** tab, use the **Description** field to enter your descriptive text. During function export, the text you enter is emitted to the generated code in the header for the Inport block. For example, if you open the example program

rtwdemo_exporting_functions and enter a description in the Block Properties dialog box for port t_1tic_A, code similar to the following is emitted:

```
/*
 * Output and update for exported function: t_1tic_A
 *
 * My custom description of the exported function
 */
void t_1tic_A(void)
{
...
}
```

Optimize Exported Function-Call Subsystems

To optimize the code generated for a function-call subsystem or virtual block that contains such subsystems, you can

- Specify a storage class for every input signal and output signal that crosses the boundary of the subsystem.
- For each function-call subsystem to be exported (whether directly or within a virtual subsystem):
 - 1 Right-click the subsystem and choose **Block Parameters (Subsystem)** from the context menu.
 - 2 Select the **Code Generation** tab and set the **Function packaging** parameter to Auto.
 - 3 Click **OK** or **Apply**.

Export Functions That Depend on Elapsed Time

Some blocks, such as the Sine Wave block (if sample-based) and the Discrete-Time Integrator block, depend on elapsed time. See “Absolute and Elapsed Time Computation” in the Simulink Coder documentation for more information.

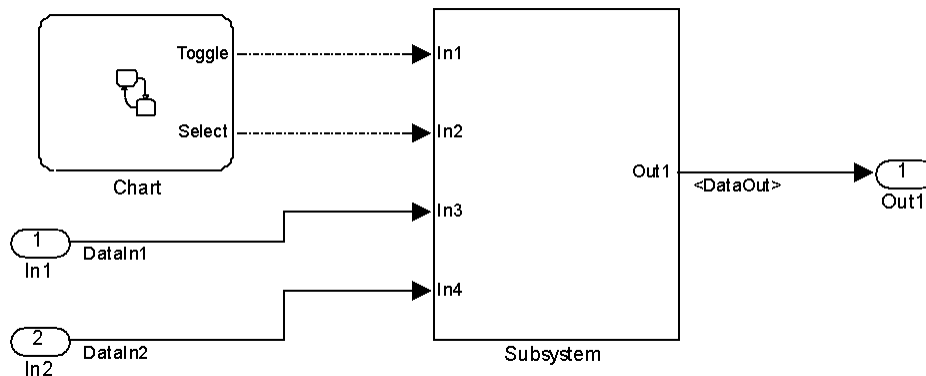
When a block that depends on elapsed time exists in a function-call subsystem, the subsystem cannot be exported unless it specifies periodic execution. To specify for this:

- 1 Right-click the Trigger block in the function-call subsystem and choose **Block Parameters** from the context menu.
- 2 Specify **periodic** in the **Sample time type** field.

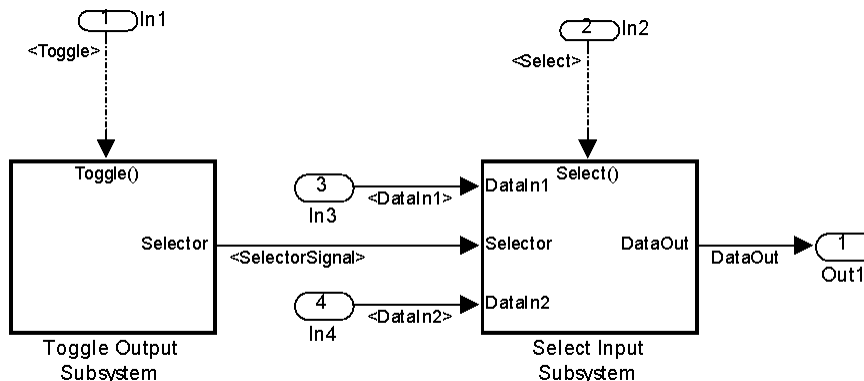
- 3 Set the **Sample time** to the same granularity specified (directly or by inheritance) in the function-call initiator.
- 4 Click **OK** or **Apply**.

Function-Call Subsystem Export

This example shows a virtual subsystem that contains two function-call subsystems, and the associated code that implements the virtual subsystem. The first figure shows the top level of a model that uses a Stateflow chart named **Chart** to input two function-call trigger signals (denoted by dash-dot lines) to a virtual subsystem named **Subsystem**.



The next figure shows the contents of **Subsystem** in the previous figure. The subsystem contains two function-call subsystems, each driven by one of the signals input from the top level.



In the preceding model, the Stateflow chart can assert either of two scalar signals, `Toggle` and `Select`.

- Asserting `Toggle` toggles the Boolean state of the function-call subsystem `Toggle Output Subsystem`.
- Asserting `Select` causes the function-call subsystem `Select Input Subsystem` to assign the value of `DataIn1` or `DataIn2` to its output signal. The value assigned depends on the current state of `Toggle Output Subsystem`.

The following generated code implements the subsystem named `Subsystem`. The code is typical for virtual subsystems that contain function-call subsystems. It specifies an initialization function and a function for each contained subsystem, and would also include functions to enable and disable subsystems if applicable.

```
#include "Subsystem.h"
#include "Subsystem_private.h"

/* Exported block signals */
real_T DataIn1;           /* '<Root>/In3' */
real_T DataIn2;          /* '<Root>/In4' */
real_T DataOut;          /* '<S4>/Switch' */
boolean_T SelectorSignal; /* '<S5>/Logical Operator' */

/* Exported block states */
boolean_T SelectorState; /* '<S5>/Unit Delay' */

/* Real-time model */
RT_MODEL_Subsystem Subsystem_M_;
RT_MODEL_Subsystem *Subsystem_M = &Subsystem_M_;

/* Initial conditions for exported function: Toggle */

void Toggle_Init(void)
{
    /* Initial conditions for function-call system: '<S1>/Toggle Output Subsystem' */

    /* InitializeConditions for UnitDelay: '<S5>/Unit Delay' */
    SelectorState = Subsystem_P.UnitDelay_X0;
}

/* Output and update for exported function: Toggle */

void Toggle(void)
{
    /* Output and update for function-call system: '<S1>/Toggle Output Subsystem' */

    /* Logic: '<S5>/Logical Operator' incorporates:
     * UnitDelay: '<S5>/Unit Delay'
     */
    SelectorSignal = !SelectorState;
}
```

```
    /* Update for UnitDelay: '<S5>/Unit Delay' */
    SelectorState = SelectorSignal;
}

/* Output and update for exported function: Select */

void Select(void)
{
    /* Output and update for function-call system: '<S1>/Select Input Subsystem' */

    /* Switch: '<S4>/Switch' incorporates:
     * Inport: '<Root>/In3'
     * Inport: '<Root>/In4'
     */
    if(SelectorSignal) {
        DataOut = DataIn1;
    } else {
        DataOut = DataIn2;
    }
}

/* Model initialize function */

void Subsystem_initialize(void)
{
    /* initialize error status */
    rtmSetErrorStatus(Subsystem_M, (const char_T *)0);

    /* block I/O */

    /* exported global signals */
    DataOut = 0.0;
    SelectorSignal = FALSE;

    /* states (dwork) */

    /* exported global states */
    SelectorState = FALSE;

    /* external inputs */
    DataIn1 = 0.0;
    DataIn2 = 0.0;

    Toggle_Init();
}

/* Model terminate function */

void Subsystem_terminate(void)
{
    /* (no terminate code required) */
}
```

Function-Call Subsystems Export Limitations

The function-call subsystem export capabilities have the following limitations:

- Subsystem block parameters do not control the names of the files containing the generated code. Such filenames begin with the name of the exported subsystem.
- Subsystem block parameters do not control the names of top-level functions in the generated code. Each function name reflects the name of the signal that triggers the function, or for an unnamed signal, the block from which the signal originates.
- The software cannot export reusable code for a function-call subsystem. The **Code interface packaging** value **Reusable function** does not apply for a function-call subsystem.
- The function-call subsystem export capability does not support **C++ class** code interface packaging.
- The software supports a SIL or PIL block in Accelerator mode only if its function-call initiator is noninlined in Accelerator mode. Examples of noninlined initiators include Stateflow charts.
- The SIL block must be driven by a Level-2 S-function initiator block, such as a Stateflow chart or the built-in Function-call Generator block.
- An asynchronous (sample-time) function-call system can be exported, but the software does not support the SIL or PIL block wrapper for an asynchronous system.
- The software does not support MAT-file logging for exported function calls. Specifications that enable MAT-file logging is ignored.
- The use of the TLC function **LibIsFirstInit** is deprecated for exported function calls.

Control Generation of Function Prototypes

The Embedded Coder software provides a **Configure Model Functions** button, located on the **Code Generation > Interface** pane of the Configuration Parameters dialog box, that allows you to control the model function prototypes that are generated for ERT-based Simulink models.

By default, the function prototype of an ERT-based model's generated *model_step* function resembles the following:

```
void model_step(void);
```

The function prototype of an ERT-based model's generated *model_initialize* function resembles the following:

```
void model_initialize(void);
```

(For more detailed information about the default calling interface for the *model_step* function, see the *model_step* reference page.)

The **Configure Model Functions** button on the **Interface** pane provides you flexible control over the model function prototypes that are generated for your model. Clicking **Configure Model Functions** launches a Model Interface dialog box. Based on the **Function specification** value you specify for your model function (supported values include **Default model initialize and step functions** and **Model specific C prototypes**), you can preview and modify the function prototypes. Once you validate and apply your changes, you can generate code based on your function prototype modifications.

For more information about using the **Configure Model Functions** button and the Model Interface dialog box, see “Sample Procedure for Configuring Function Prototypes” and the example model *rtwdemo_fcncnprotoctrl*, which is preconfigured to demonstrate function prototype control.

Alternatively, you can use function prototype control functions to programmatically control model function prototypes. For more information, see “Configure Function Prototypes Programmatically” “Configure Function Prototypes Programmatically”.

You can also control model function prototypes for nonvirtual subsystems, if you generate subsystem code using right-click build. To launch the **Model Interface for subsystem** dialog box, use the `RTW.configSubsystemBuild` function.

Right-click building the subsystem generates the step and initialization functions according to the customizations you make. For more information, see “Configure Function Prototypes for Nonvirtual Subsystems”.

For limitations that apply, see “Function Prototype Control Limitations”.

C++ Class Interface Control

Using the **Code interface packaging** option `C++ class`, on the **Code Generation > Interface** pane of the Configuration Parameters dialog box, you can generate a C++ class interface to model code. The generated interface encapsulates required model data into C++ class attributes and model entry point functions into C++ class methods. The benefits of C++ class encapsulation include:

- Greater control over access to model data
- Ability to multiply instantiate model classes
- Easier integration of model code into C++ programming environments

C++ class encapsulation also works for right-click builds of nonvirtual subsystems. (For information on requirements that apply, see “Configure C++ Class Interfaces for Nonvirtual Subsystems”.)

The general procedure for generating C++ class interfaces to model code is as follows:

- 1 Configure your model to use an `ert.tlc` system target file provided by MathWorks.
- 2 Select the C++ language for your model.
- 3 Select `C++ class` code interface packaging for your model.
- 4 Optionally, configure related C++ class interface settings for your model code, using either a graphical user interface (GUI) or application programming interface (API).
- 5 Generate model code and examine the results.

To get started with an example, see “Simple Use of C++ Class Control”. For more details about configuring C++ class interfaces for your model code, see “Customize C++ Class Interfaces Using Graphical Interfaces” and “Customize C++ Class Interfaces Programmatically”. For limitations that apply, see “C++ Class Interface Control Limitations”.

Note: For an example of C++ class code generation, see the example model `rtwdemo_cppclass`.

Code Replacement Customization for Simulink Models

- “What Is Code Replacement Customization?” on page 21-3
- “Code You Can Replace From Simulink Models” on page 21-4
- “Code Replacement Match and Replacement Process” on page 21-22
- “Code Replacement Customization Limitations” on page 21-24
- “Develop a Code Replacement Library” on page 21-26
- “Quick Start Library Development” on page 21-27
- “Identify Code Replacement Requirements” on page 21-37
- “Prepare for Code Replacement Library Development” on page 21-40
- “Define Code Replacement Mappings” on page 21-42
- “Specify Build Information for Replacement Code” on page 21-58
- “Register Code Replacement Mappings” on page 21-66
- “Review and Test Code Replacements” on page 21-74
- “Deploy Code Replacement Library” on page 21-87
- “Math Function Code Replacement” on page 21-88
- “Memory Function Code Replacement” on page 21-90
- “Nonfinite Function Code Replacement” on page 21-93
- “Semaphore and Mutex Function Replacement” on page 21-96
- “Algorithm-Based Code Replacement” on page 21-103
- “Lookup Table Function Code Replacement” on page 21-106
- “Data Alignment for Code Replacement” on page 21-125
- “Replace MATLAB Functions with Custom Code Using `coder.replace`” on page 21-136
- “Replace MATLAB Functions Specified in MATLAB Function Blocks” on page 21-138

- “Reserved Identifiers and Code Replacement” on page 21-142
- “Customize Matching and Replacement Process for Functions” on page 21-144
- “Scalar Operator Code Replacement” on page 21-149
- “Addition and Subtraction Operator Code Replacement” on page 21-152
- “Small Matrix Operation to Processor Code Replacement” on page 21-157
- “Matrix Multiplication Operation to MathWorks BLAS Code Replacement” on page 21-162
- “Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement” on page 21-170
- “Remap Operator Output to Function Input” on page 21-177
- “Customize Matching and Replacement Process for Operators” on page 21-180
- “Fixed-Point Operator Code Replacement” on page 21-187
- “Binary-Point-Only Scaling Code Replacement” on page 21-197
- “Slope Bias Scaling Code Replacement” on page 21-201
- “Net Slope Scaling Code Replacement” on page 21-205
- “Equal Slope and Zero Net Bias Code Replacement” on page 21-212
- “Data Type Conversions (Casts) and Operator Code Replacement” on page 21-216
- “Shift Left Operations and Code Replacement” on page 21-221

What Is Code Replacement Customization?

Customize how and when the code generator replaces C/C++ code that it generates by default for functions and operators by developing a custom code replacement library. You can develop libraries interactively with the Code Replacement Tool or programmatically.

- Develop libraries tailored to specific application requirements
- Add identifiers to the list of reserved keywords the code generator considers during code replacement
- Customize the code generator’s matching and replacement process for functions

To get started, “Quick Start Library Development”.

Related Examples

- “Quick Start Library Development”
- “Develop a Code Replacement Library”

More About

- “What Is Code Replacement?”
- “Code You Can Replace From Simulink Models”
- “Code Replacement Libraries”
- “Code Replacement Terminology”
- “Code Replacement Customization Limitations”

External Web Sites

- “Replacing Math Functions and Operators”

Code You Can Replace From Simulink Models

In this section...

“About Code You Can Replace” on page 21-4

“Math Functions – Simulink Support” on page 21-4

“Math Functions – Stateflow Support” on page 21-10

“Memory Functions” on page 21-15

“Nonfinite Functions” on page 21-16

“Mutex and Semaphore Functions” on page 21-17

“Operators” on page 21-18

About Code You Can Replace

Code that the code generator replaces depends on the code replacement library (CRL) that you use. By default, the code generator does not apply a code replacement library. Your choice of libraries is dependent on product licensing and whether you have access to custom libraries.

For information on how to explore functions and operators that a code replacement library supports, see “Choose a Code Replacement Library” license and want to develop a custom code replacement library, see Code Replacement Customization.

Math Functions – Simulink Support

When generating C/C++ code from a Simulink model, depending on code replacement libraries available in your development environment, you can configure the code generator to replace instances of the following math functions with application-specific implementations.

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
abs ¹	Integer Floating point Fixed point	Scalar Vector Matrix	Real
acos	Floating point	Scalar	Real Complex input/complex output Real input/complex output

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
acosd^2	Floating point	Scalar Vector Matrix	Real Complex
acosh	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
acot^2	Floating point	Scalar Vector Matrix	Real Complex
acotd^2	Floating point	Scalar Vector Matrix	Real Complex
acoth^2	Floating point	Scalar Vector Matrix	Real Complex
acsc^2	Floating point	Scalar Vector Matrix	Real Complex
acscd^2	Floating point	Scalar Vector Matrix	Real Complex
acsch^2	Floating point	Scalar Vector Matrix	Real Complex
asec^2	Floating point	Scalar Vector Matrix	Real Complex
asecd^2	Floating point	Scalar Vector Matrix	Real Complex
asech^2	Floating point	Scalar Vector Matrix	Real Complex

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
asin	Floating point	Scalar	Real Complex input/complex output Real input/complex output
asind ²	Floating point	Scalar Vector Matrix	Real Complex
asinh	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
atan	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
atan2	Floating point	Scalar Vector Matrix	Real
atan2d ²	Floating point	Scalar Vector Matrix	Real
atand ²	Floating point	Scalar Vector Matrix	Real Complex
atanh	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
ceil	<ul style="list-style-type: none"> • Floating-point • Scalar 	<ul style="list-style-type: none"> • Floating-point • Scalar 	<ul style="list-style-type: none"> • Floating-point • Scalar
cos ³	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
cosd ²	Floating point	Scalar Vector Matrix	Real Complex

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
cosh	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
cot ²	Floating point	Scalar Vector Matrix	Real Complex
cotd ²	Floating point	Scalar Vector Matrix	Real Complex
coth ²	Floating point	Scalar Vector Matrix	Real Complex
csc ²	Floating point	Scalar Vector Matrix	Real Complex
cscd ²	Floating point	Scalar Vector Matrix	Real Complex
csch ²	Floating point	Scalar Vector Matrix	Real Complex
exactrSqrt	Integer Floating point	Scalar	Real
exp	Floating point	Scalar Vector Matrix	Real
fix	Floating point	Scalar	Real
floor	• Floating-point • Scalar	• Floating-point • Scalar	• Floating-point • Scalar
fmod ⁴	Floating point	Scalar	Real
frexp	Floating point	Scalar	Real

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
hypot	Floating point	Scalar Vector Matrix	Real
ldexp	Floating point	Scalar	Real
ln	Floating point	Scalar	Real
log	Floating point	Scalar Vector Matrix	Real
log10	Floating point	Scalar Vector Matrix	Real
log2 ²	Floating point	Scalar Vector Matrix	Real Complex
max	Integer Floating point Fixed point	Scalar	Real
min	Integer Floating point Fixed point	Scalar	Real
mod	Integer Floating point	Scalar Vector Matrix	Real
pow	Floating point	Scalar Vector Matrix	Real
rem	Floating point	Scalar Vector Matrix	Real
round	Floating point	Scalar	Real

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
rSqrt	Integer Floating point	Scalar Vector Matrix	Real
saturate	Integer Floating point Fixed point	Scalar Vector Matrix	Real
sec ²	Floating point	Scalar Vector Matrix	Real Complex
secd ²	Floating point	Scalar Vector Matrix	Real Complex
sech ²	Floating point	Scalar Vector Matrix	Real Complex
sign	Integer Floating point Fixed point	Scalar Vector Matrix	Real
signPow	Floating point	Scalar Vector Matrix	Real
sin ³	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
sincos ³	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
sind ²	Floating point	Scalar Vector Matrix	Real Complex
sinh	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
sqrt	Integer Floating point Fixed point	Scalar Vector Matrix	Real
tan	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
tand ²	Floating point	Scalar Vector Matrix	Real Complex
tanh	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
<p>¹ Wrap on integer overflow only. Clear block parameter Saturate on integer overflow.</p> <p>² Only when used with the MATLAB Function block.</p> <p>³ Supports the CORDIC approximation method.</p> <p>⁴ Stateflow support only.</p>			

Math Functions – Stateflow Support

When generating C/C++ code from Stateflow charts, depending on code replacement libraries available in your development environment, you can configure the code generator to replace instances of the following math functions with application-specific implementations.

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
abs ¹	Integer Floating point	Scalar	Real
acos ²	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
			Real input/complex output
acosd^3	Floating point	Scalar Vector Matrix	Real Complex
acot^3	Floating point	Scalar Vector Matrix	Real Complex
acotd^3	Floating point	Scalar Vector Matrix	Real Complex
$\text{acoth}^{3,5}$	Floating point	Scalar Vector Matrix	Real Complex
acsc^3	Floating point	Scalar Vector Matrix	Real Complex
acscd^3	Floating point	Scalar Vector Matrix	Real Complex
acsch^3	Floating point	Scalar Vector Matrix	Real Complex
asec^3	Floating point	Scalar Vector Matrix	Real Complex
asecd^3	Floating point	Scalar Vector Matrix	Real Complex
asech^3	Floating point	Scalar Vector Matrix	Real Complex

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
asin ²	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
asind ³	Floating point	Scalar Vector Matrix	Real Complex
atan ²	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
atan2 ²	Floating point	Scalar Vector Matrix	Real
atan2d ³	Floating point	Scalar Vector Matrix	Real
atand ³	Floating point	Scalar Vector Matrix	Real Complex
ceil	<ul style="list-style-type: none"> • Floating-point • Scalar 	<ul style="list-style-type: none"> • Floating-point • Scalar 	<ul style="list-style-type: none"> • Floating-point • Scalar
cos ³	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
cosd ³	Floating point	Scalar Vector Matrix	Real Complex

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
\cosh^2	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
\cot^3	Floating point	Scalar Vector Matrix	Real Complex
\cotd^3	Floating point	Scalar Vector Matrix	Real Complex
\coth^3	Floating point	Scalar Vector Matrix	Real Complex
\csc^3	Floating point	Scalar Vector Matrix	Real Complex
\cscd^3	Floating point	Scalar Vector Matrix	Real Complex
\csch^3	Floating point	Scalar Vector Matrix	Real Complex
\exp	Floating point	Scalar	Real
floor	<ul style="list-style-type: none"> • Floating-point • Scalar 	<ul style="list-style-type: none"> • Floating-point • Scalar 	<ul style="list-style-type: none"> • Floating-point • Scalar
fmod	Floating point	Scalar	Real
hypot^3	Floating point	Scalar Vector Matrix	Real
ldexp	Floating point	Scalar	Real

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
\log^2	Floating point	Scalar Vector Matrix	Real Complex
\log_{10}^2	Floating point	Scalar Vector Matrix	Real Complex
\log_2^3	Floating point	Scalar Vector Matrix	Real Complex
max	Integer Floating point	Scalar	Real
min	Integer Floating point	Scalar	Real
pow	Floating point	Scalar	Real
\sec^3	Floating point	Scalar Vector Matrix	Real Complex
\secd^3	Floating point	Scalar Vector Matrix	Real Complex
\sech^3	Floating point	Scalar Vector Matrix	Real Complex
\sin^2	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
\sind^3	Floating point	Scalar Vector Matrix	Real Complex

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
\sinh^2	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
$\sqrt{}$	Floating point	Scalar	Real
\tan^2	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
\tand^3	Floating point	Scalar Vector Matrix	Real Complex
\tanh^2	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
¹ Wrap on integer overflow only. ² For models involving vectors or matrices, the code generator replaces only functions coded in the MATLAB action language. ³ The code generator replaces only functions coded in the MATLAB action language.			

Memory Functions

Depending on code replacement libraries available in your development environment, you can configure the code generator to replace instances of the following memory functions with application-specific implementations.

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
memcmp	Void pointer (void*)	Scalar	Real

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
		Vector Matrix	Complex
memcpy	Void pointer (void*)	Scalar Vector Matrix	Real Complex
memset	Void pointer (void*)	Scalar Vector Matrix	Real Complex
memset2zero	Void pointer (void*)	Scalar Vector Matrix	Real Complex

Some target processors provide optimized functions to set memory to zero. Use the code replacement library programming interface to replace the `memset2zero` function with more efficient target-specific functions.

Nonfinite Functions

Depending on code replacement libraries available in your development environment, you can configure the code generator to replace instances of the following nonfinite functions with application-specific implementations.

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
getInf	Floating point	Scalar	Real
getMinusInf	Floating point	Scalar	Real
getNaN	Floating point	Scalar	Real
rtIsInf	Floating point	Scalar	Real Complex
rtIsNaN	Floating point	Scalar	Real Complex

Mutex and Semaphore Functions

Mutex and semaphore functions control access to resources shared by multiple processes in multicore target environments. MathWorks provides code replacement libraries that support mutex and semaphore replacement for Rate Transition and Task Transition blocks on Windows, Linux, Mac, and VxWorks platforms.

Generated mutex and semaphore code typically consists of:

- In model initialization code, an initialization function call to create a mutex or semaphore to control entry to a critical section of code.
- In model step code:
 - Before code for a data transfer between tasks enters the critical section, mutex lock or semaphore wait function calls to reserve a critical section of code.
 - After code for a data transfer between tasks finishes executing the critical section, mutex unlock or semaphore post function calls to release the critical section of code.
- In model termination code, an optional destroy function call to explicitly delete the mutex or semaphore.

Depending on code replacement libraries available in your development environment, you can configure the code generator to replace instances of the following mutex and semaphore functions with application-specific implementations.

In the following table, key is a string that identifies the function.

Function	Key
Mutex Destroy	RTW_MUTEX_DESTROY
Mutex Init	RTW_MUTEX_INIT
Mutex Lock	RTW_MUTEX_LOCK
Mutex Unlock	RTW_MUTEX_UNLOCK
Semaphore Destroy	RTW_SEM_DESTROY
Semaphore Init	RTW_SEM_INIT
Semaphore Post	RTW_SEM_POST
Semaphore Wait	RTW_SEM_WAIT

Operators

When generating C/C++ code from a Simulink model, depending on code replacement libraries available in your development environment, you can configure the code generator to replace instances of the following operators with application-specific implementations.

In the following table:

- Key is a string that identifies the operator.
- Mixed data type support indicates that you can specify different data types for different inputs.

Operator	Key	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
Addition (+)	RTW_OP_ADD	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Subtraction (-)	RTW_OP_MINUS	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Multiplication (*) ¹	RTW_OP_MUL	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Division (/)	RTW_OP_DIV	Integer Floating point Fixed-point Mixed	Scalar	Real Complex
Data type conversion (cast)	RTW_OP_CAST	Integer Floating point ² Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Shift left (<<)	RTW_OP_SL	Integer Fixed-point	Scalar Vector	Real

Operator	Key	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
		Mixed	Matrix	
Shift right arithmetic (>>) ³	RTW_OP_SRA	Integer Fixed-point Mixed	Scalar Vector Matrix	Real
Shift right logical (>>)	RTW_OP_SRL	Integer Fixed-point Mixed	Scalar Vector Matrix	Real
Element-wise matrix multiplication (.*) ⁴	RTW_OP_ELEM_MUL	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Matrix right division (/)	RTW_OP_RDIV	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Matrix left division (\)	RTW_OP_LDIV	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Matrix inversion (inv)	RTW_OP_INV	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Complex conjugation	RTW_OP_CONJUGATE	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Transposition (.')	RTW_OP_TRANS	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex

Operator	Key	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
Hermitian (complex conjugate) transposition (')	RTW_OP_HERMITIAN	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Multiplication with transposition ¹	RTW_OP_TRMUL	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Multiplication with Hermitian transposition ¹	RTW_OP_HMMUL	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Greater than (>)	RTW_OP_GREATER_THAN	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Greater than or equal(>=)	RTW_OP_GREATER_THAN_OR_EQUAL	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Less than (<)	RTW_OP_LESS_THAN	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Less than or equal (<=)	RTW_OP_LESS_THAN_OR_EQUAL	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Equal (==)	RTW_OP_EQUAL	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex

Operator	Key	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
Not equal (!=)	RTW_OP_NOT_EQUAL	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
<p>¹ Can map to Basic Linear Algebra Subroutine (BLAS) multiplication functions.</p> <p>² Scaled floating point is not supported.</p> <p>³ Code replacement libraries that provide arithmetic shift right implementations should also provide logical shift right implementations, because some arithmetic shift rights are converted to logical shift rights during code generation.</p> <p>⁴ Use the multiplication (*) operator (RTW_OP_MUL) for scalar multiplication.</p>				

Related Examples

- “Develop a Code Replacement Library”
- “Quick Start Library Development”
- “Choose a Code Replacement Library”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code Replacement Libraries”
- “Code Replacement Terminology”
- “Code Replacement Customization Limitations”

Code Replacement Match and Replacement Process

When the code generator encounters a call site for a function or operator, it:

- 1** Creates and partially populates a code replacement entry object with the function or operator name or key and conceptual arguments.
- 2** Uses the entry object to query the configured code replacement library for a conceptual representation match. The code generator searches the tables in a code replacement library for a match in the order that the tables appear in the library. When searching for a match, the code generator takes into account:
 - Conceptual name or key
 - Arguments, including quantity, type, type qualifiers, and complexity
 - Algorithm (computation method)
 - Fixed-point saturation and rounding modes
 - Priority
- 3** When a match exists, the code generator returns a code replacement object, fully populated with the conceptual representation, implementation representation, and priority. If the code generator finds multiple matches within a table, the entry priority determines the match. The priority can range from 0 to 100. The highest priority is 0. The code generator uses a higher-priority entry over a similar entry with a lower priority.
- 4** Uses the C or C++ replacement function prototype in the code replacement object to generate code.

When the code generator encounters a call site for a function or operator, it:

- 1** Creates and partially populates a code replacement entry object with the function or operator name or key and conceptual arguments.
- 2** Uses the entry object to query the configured code replacement library for a conceptual representation match. The code generator searches the tables in a code replacement library for a match in the order that the tables appear in the library. When searching for a match, the code generator takes into account:
 - Conceptual name or key
 - Arguments, including quantity, type, type qualifiers, and complexity
 - Algorithm (computation method)

- Fixed-point saturation and rounding modes
 - Priority
- 3** When a match exists, the code generator returns a code replacement object, fully populated with the conceptual representation, implementation representation, and priority. If the code generator finds multiple matches within a table, the entry priority determines the match. The priority can range from 0 to 100. The highest priority is 0. The code generator uses a higher-priority entry over a similar entry with a lower priority.
 - 4** Uses the C or C++ replacement function prototype in the code replacement object to generate code.

Related Examples

- “Customize Matching and Replacement Process for Functions”
- “Customize Matching and Replacement Process for Operators”

More About

- “Code Replacement Libraries”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Code Replacement Customization Limitations

- Code replacement verification — It is possible that code replacement behaves differently than you expect. For example, data types that you observe in code generator input might not match what the code generator uses as intermediate data types during an operation. Verify code replacements by examining generated code. See “Review and Test Code Replacements”.
- Tokens in file paths—You can include tokens in file paths when specifying build information for a code replacement entry by using the programming interface only. The ability to include tokens is not available from the Code Replacement Tool. See “Specify Build Information for Replacement Code”.
- Addition and subtraction operation replacements—See “Addition and Subtraction Operator Code Replacement” for relevant limitations.
- Data alignment—
 - Not supported for
 - Software-in-the-loop (SIL)
 - Processor-in-the-loop (PIL)
 - Model reference parameters
 - Exported functions in Stateflow charts
 - Data alignment is not honored in code generation for shared utilities if the replacement would impose alignment requirements on the shared utility interface arguments. Under these conditions, replacement does not occur. Replacement is allowed if the registered data alignment type specification supports alignment of local variables, and the replacement involves only local variables.
 - For `Simulink.Bus`:
 - If structure field alignment is not supported by user registered alignment specifications, aligning `Simulink.Bus` objects is not supported unless the `Simulink.Bus` is imported.
 - When aligning a `Simulink.Bus` data object, the elements in the bus object are aligned on the same boundary, which is the lowest common multiple of the alignment requirements for each individual bus element.
 - When you specify alignment for functions that occur in a model reference hierarchy, and the same function data is operated on by multiple models in the hierarchy, the bottommost model dictates the alignment for the rest of the

hierarchy. If the alignment requirement for a function in a model higher in the hierarchy cannot be honored due to the alignment set by a model lower in the hierarchy, the replacement in the higher model does not occur (in some cases, an error message is generated). To work around this issue, if the shared data is represented by a bus or signal object, you can manually set the alignment property on the shared data by setting the alignment property of the `Simulink.Bus` or `Simulink.Signal` object.

See “Data Alignment for Code Replacement”.

- `coder.replace` function — See `coder.replace` for relevant limitations.

Related Examples

- “Review and Test Code Replacements”
- “Specify Build Information for Replacement Code”
- “Data Alignment for Code Replacement”
- “Replace MATLAB Functions with Custom Code Using `coder.replace`”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “Code Replacement Libraries”
- “Code Replacement Terminology”

Develop a Code Replacement Library

- 1 “Identify Code Replacement Requirements”
- 2 “Prepare for Code Replacement Library Development”
- 3 “Define Code Replacement Mappings”
- 4 “Specify Build Information for Replacement Code”
- 5 “Register Code Replacement Mappings”
- 6 “Review and Test Code Replacements”
- 7 “Deploy Code Replacement Library”

Related Examples

- “Replace MATLAB Functions with Custom Code Using `coder.replace`”
- “Replace MATLAB Functions Specified in MATLAB Function Blocks”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code You Can Replace From Simulink Models”
- “Code Replacement Libraries”
- “Code Replacement Terminology”
- “Code Replacement Customization Limitations”

External Web Sites

- “Replacing Math Functions and Operators”

Quick Start Library Development

This example shows how to develop a code replacement library that includes an entry for generating replacement code for the math function `sin`. You use the Code Replacement Tool.

Prerequisites

To complete this example, install the following software:

- MATLAB
- Simulink
- Simulink Coder
- Embedded Coder

For instructions on installing MathWorks products, see the “Installation Guide”. If you have installed MATLAB and want to see what other MathWorks products are installed, in the Command Window, enter `ver`.

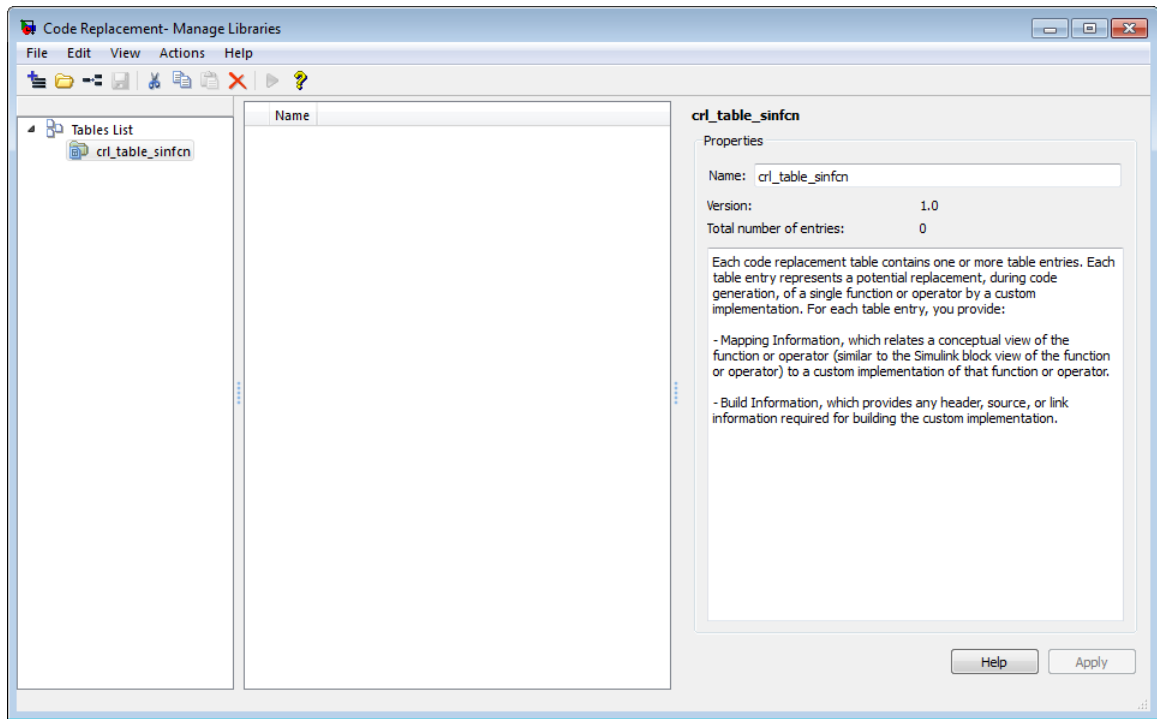
For a list of supported compilers, see http://www.mathworks.com/support/compilers/current_release/.

Open the Code Replacement Tool

- 1 Start a new MATLAB session.
- 2 Create or navigate (`cd`) to an empty folder.
- 3 At the command prompt, enter the `crtool` command. The Code Replacement Tool window opens.

Create Code Replacement Table

- 1 In the Code Replacement Tool window, select **File > New table**.
- 2 In the right pane, name the table `crl_table_sinfcn` and click **Apply**. Later, when you save the table, the tool saves it with the file name `crl_table_sinfcn.m`.



Create Table Entry

Create a table entry that maps a `sin` function with double input and double output to a custom implementation function.

- 1 In the left pane, select table `crl_table_sinfcn`. Then, select **File > New entry > Function**. The new entry appears in the middle pane, initially without a name.
- 2 In the middle pane, select the new entry.
- 3 In the right pane, on the **Mapping Information** tab, from the **Function** menu, select `sin`.
- 4 Leave **Algorithm** set to **Unspecified**, and leave parameters in the **Conceptual function** group set to default values.
- 5 In the **Replacement function** group, name the replacement function `sin_dbl`.
- 6 Leave the remaining parameters in the **Replacement function** group set to default values.

- 7 Click **Apply**. The tool updates the **Function signature preview** to reflect the specified replacement function name.
- 8 Scroll to the bottom of the **Mapping Information** tab and click **Validate entry**. The tool validates your entry.

The following figure shows the completed mapping information.

Mapping Information

Build Information

Function: sin

Entry information

Algorithm: Unspecified

Conceptual function

Used by code generation process for matching purposes

Conceptual arguments

y1
u1

Argument properties

Data type: double

Complex

Argument type: Scalar

Make conceptual and implementation argument types the same

Replacement function

Function prototype

Name: sin_dbl C++ namespace:

Function returns void

Function arguments

y1(return arg)
u1

Argument properties

Data type: double I/O type: OUTPUT

Const Pointer Complex

✕ +

Function signature preview

`double sin_dbl(double u1);`

Implementation attributes

Integer saturation mode: Unspecified Saturation

Rounding mode: Unspecified Rounding

Allow expressions as inputs

Function modifies internal or global state

[Click here to add Build Information](#)

Validation

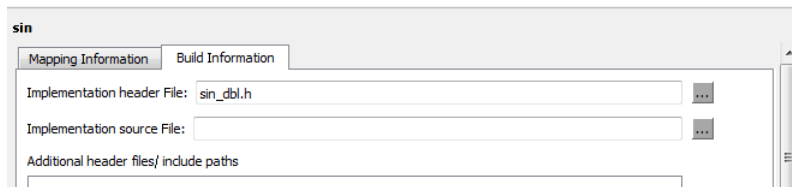
Validate entry Status: Validated

Help

Apply

Specify Build Information for Replacement Code

- 1 On the **Build Information** tab, for the **Implementation header file** parameter, enter `sin_dbl.h`.
- 2 Leave the remaining parameters set to default values.
- 3 Click **Apply**.



- 4 Optionally, you can revalidate the entry. Return to the **Mapping Information** tab and click **Validate entry**.

Create Another Table Entry

Create an entry that maps a `sin` function with `single` input and `double` output to a custom implementation function named `sin_sgl`. Create the entry by copying and pasting the `sin_dbl` entry.

- 1 In the middle pane, select the `sin_dbl` entry.
- 2 Select **Edit > Copy**
- 3 Select **Edit > Paste**
- 4 On the **Mapping Information** tab, in the **Conceptual function** section, set the data type of input argument `u1` to `single`.
- 5 In the **Replacement function** section, name the function `sin_sgl`. Set the data type of input argument `u1` to `single`.
- 6 Click **Apply**. Note the changes that appear for the **Function signature preview**.
- 7 On the **Build Information** tab, for the **Implementation header file** parameter, enter `sin_sgl.h`. Leave the remaining parameters set to default values and click **Apply**.

Validate the Code Replacement Table

- 1 Select **Actions > Validate table**.

- 2 If the tool reports errors, fix them, and rerun the validation. Repeat fixing and validating errors until the tool does not report errors. The following figure shows a validation report.

	Name	Implementation	NumIn	In1Type	In2Type	Out1Type	Out2Type	Priority
✓	sin	sin_dbl	1	double		double		100
✓	sin	sin_sgl	1	single		double		100

Save the Code Replacement Table

Save the code replacement table to a MATLAB file in your working folder. Select **File > Save table**. By default, the tool uses the table name to name the file. For this example, the tool saves the table in the file `cr1_table_sinfcn.m`.

Review the Code Replacement Table Definition

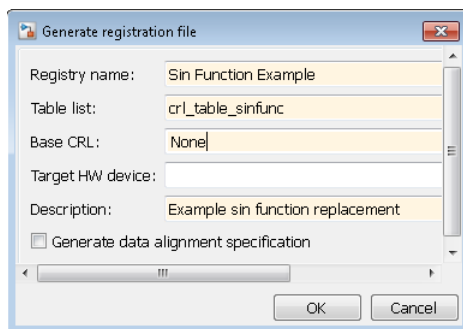
Consider reviewing the MATLAB code for your code replacement table definition. After using the tool to create an initial version of a table definition file, you can update, enhance, or copy the file in a text editor.

To review it, in MATLAB or another text editor, open the file `cr1_table_sinfcn.m`.

Generate a Registration File

Before you can use your code replacement table, you must register it as part of a code replacement library. Use the Code Replacement Tool to generate a registration file.

- 1 In the Code Replacement Tool, select **File > Generate registration file**.
- 2 In the **Generate registration file** dialog box, edit the dialog box fields to match the following figure, and then click **OK**.



- 3 In the **Select location** dialog box, specify a location for the registration file. The location must be on the MATLAB path or in the current working folder. Save the file. The tool saves the file as `rtwTargetInfo.m`.

Register the Code Replacement Table

At the command prompt, enter:

```
sl_refresh_customizations
```

Review and Test Code Replacements

Apply your code replacement library. Verify that the code generator makes code replacements that you expect.

- 1 Check for errors. At the command line, invoke the table definition file. For example:

```
tbl = crl_table_sinfcn
```

```
tbl =
```

```
Tf1Table with properties:
```

```

        Version: '1.0'
    ReservedSymbols: []
StringResolutionMap: []
        AllEntries: [2x1 RTW.Tf1CFunctionEntry]
        EnableTrace: 1

```

If an error exists in the definition file, the invocation triggers a message. Fix the error and try again.

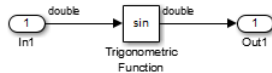
- 2 Use the Code Replacement Viewer to check your code replacement entries. For example:

```
RTW.viewTf1('Sin Function Example')
```

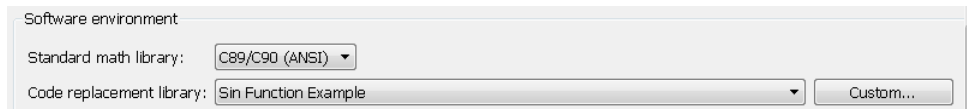
In the viewer, select entries in your table and verify that the content is what you expect. The viewer can help you detect issues such as:

- Incorrect argument order.
- Conceptual argument names that do not match what is expected by the code generator.

- Incorrect priority settings.
- 3 Identify an existing or create a new Simulink model that includes a Trigonometric Function block that is set to the `sin` function. For example:

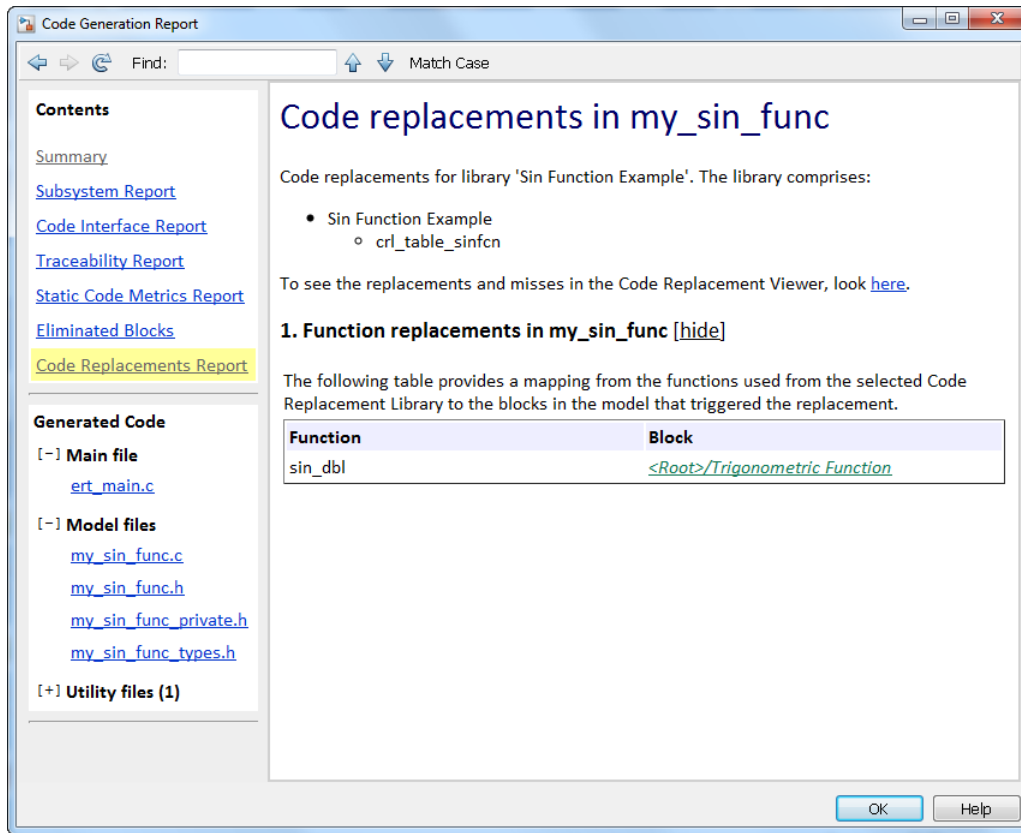


- 4 Open the model and configure it for code generation with an Embedded Coder (ERT-based) target.
- 5 See whether your library is listed as an available option for the **Code Generation > Interface > Code replacement library** model configuration parameter. If it is, select it.



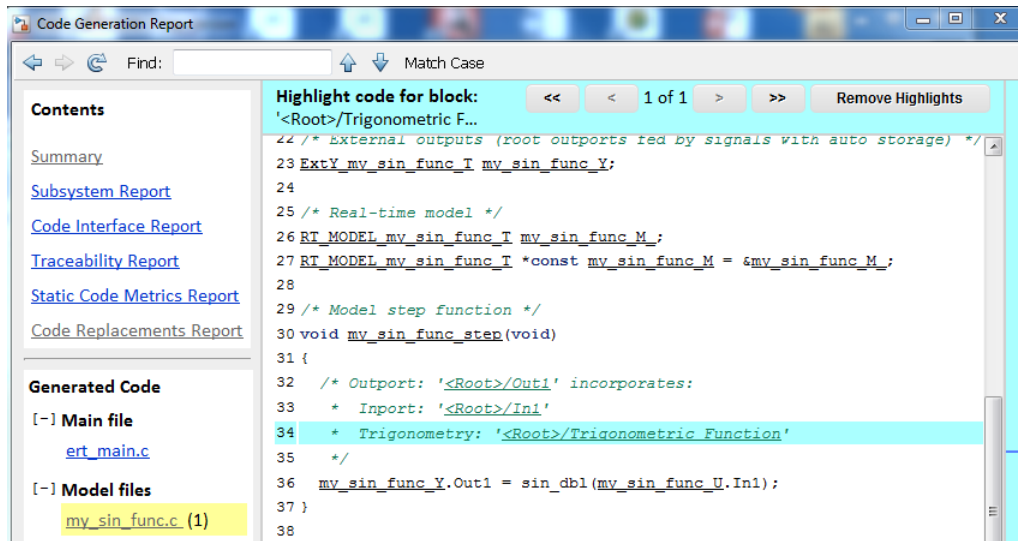
If it is not listed, open the registration file, `rtwTargetInfo.m`. See whether you entered the correct code replacement table name when you created the file. If you hover the cursor over the selected library, a tool tip appears. This tip contains information derived from your code replacement library registration file, such as the library description and the list of tables it contains.

- 6 Configure the code generation report for code replacement analysis by setting the following parameters:
 - On the **Code Generation > Report** pane, select **Create code generation report**, **Open report automatically**, **Code-to-model**, **Model-to-code**, and **Summarize which blocks triggered code replacements**.
 - On the **Code Generation > Comments** pane, select **Include comments**, **Simulink block / Stateflow object comments**, and **Simulink block descriptions**.
- 7 Configure the model to generate code only. Before you build an executable, confirm that the code generator is replacing code as expected.
- 8 Generate code for the model.
- 9 Review code replacement results in the Code Replacement Report section of the code generation report.



The report indicates that the code generator found a match and applied the replacement code for the function `sin_dbl`.

- Review the code replacements. In the model window, right-click the Trigonometric Function block. Select **C/C++ Code > Navigate to C/C++ Code**. The code generation report opens and highlights the code replacement in `my_sin_func.c`. In this case, the code generator replaced `sin` with `sin_dbl`.



Related Examples

- “Develop a Code Replacement Library”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code You Can Replace From Simulink Models”
- “Code Replacement Libraries”
- “Code Replacement Terminology”
- “Code Replacement Customization Limitations”

External Web Sites

- “Replacing Math Functions and Operators”

Identify Code Replacement Requirements

In this section...

“Mapping Information Requirements” on page 21-37

“Build Information Requirements” on page 21-38

“Registration Information Requirements” on page 21-38

Mapping Information Requirements

- Are you defining a code replacement mapping for the first time?
- Are you updating code replacement entries in an existing library? Or, are you creating a new library?
- Are you rapid prototyping code replacements?
- Can you base your mappings on existing mappings?
- What type of code do you want to replace? Options include:
 - Math operation
 - Function
 - BLAS operation
 - CBLAS operation
 - Net slope fixed-point operation
 - Semaphore or mutex functions
- Do you want to change the inline or nonfinite behavior for functions?
- What specific functions and operations do you want to replace?
- What input and output arguments does the function or operator that you are replacing take? For each argument, what is the data type, complexity, and dimensionality?
- What does the prototype for your replacement code look like?
 - What is the replacement function name?
 - What are the input and output arguments?
 - Are there return values?

- What is the data type, complexity, and dimensionality of each argument and return value?

Build Information Requirements

- Does your replacement function implementation require a header file? If yes, specify the header file.
- If the replacement function implementation requires a header file, what is the path for that file?
- Is the source file for the replacement function in your working folder? If not, you can explicitly specify the source file name and extension. For example, if the file is required in the generated makefile or specified in a build information object, specify the source file.
- Does the replacement function use additional include files? If yes, what are they and what are the paths for those files?
- Does the replacement function use additional source files? If yes, what are they and what are the paths for those files?
- What compiler flags are required for compiling code that includes the replacement code?
- What linker flags are required for building an executable that includes the replacement code?
- Are the required header, source, and object files for building an executable that includes your replacement code in the working folder for your project? If not, before starting the build process, do you want the code generator to copy required files to the build folder?

Registration Information Requirements

- What do you want to name your code replacement library?
- What code replacement tables do you want to include in the library? What are the file names and paths for the tables?
- What is the purpose of the library? You can document the purpose as the library description.
- Does the library apply to specific hardware devices? If yes, what devices?
- Are you developing a hierarchy of code replacement libraries? Is the library that you are developing based (dependent) on another library? For example, you can specify a

general TI device library as the base library for a more specific TI C28x device library.

- Do you need to specify data alignment for the library? What data alignments are required? For each specification, what type of alignment is required and for what programming language?

Related Examples

- “Develop a Code Replacement Library”
- “Define Code Replacement Mappings”
- “Specify Build Information for Replacement Code”
- “Register Code Replacement Mappings”
- “Review and Test Code Replacements”
- “Quick Start Library Development”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code You Can Replace From Simulink Models”
- “Code Replacement Libraries”
- “Code Replacement Terminology”
- “Code Replacement Customization Limitations”

Prepare for Code Replacement Library Development

After you identify your code replacement requirements, prepare for library development by reviewing this checklist:

- Get familiar with the library development process.
- Decide whether to define code replacement mappings and produce a registration file interactively with the Code Replacement Tool or programmatically.
- Identify or develop MATLAB code and Simulink models to test your code replacement library.
- Consider the hierarchy and organization of your library. A library can consist of multiple tables and each table can include multiple entries. How do you want to organize the library to optimize reuse of tables and entries? For example, a registration file can define code replacement tables organized in a hierarchy of code replacement libraries based on entries that increase in specificity:
 - Common entries
 - Entries for TI devices
 - Entries for TI C6xx devices
 - Entries specific to the TI C67x device
- If support files, such as header files, additional source files, and dynamically linked libraries are not in your current working folder, note their location. You need to specify the paths for such files.

Related Examples

- “Develop a Code Replacement Library”
- “Define Code Replacement Mappings”
- “Specify Build Information for Replacement Code”
- “Register Code Replacement Mappings”
- “Review and Test Code Replacements”
- “Quick Start Library Development”

More About

- “What Is Code Replacement?”

- “What Is Code Replacement Customization?”
- “Code You Can Replace From Simulink Models”
- “Code Replacement Libraries”
- “Code Replacement Terminology”

Define Code Replacement Mappings

In this section...

“Defining Code Replacement Mappings” on page 21-42

“Define Mappings Interactively with the Code Replacement Tool” on page 21-43

“Define Mappings Programmatically” on page 21-46

Defining Code Replacement Mappings

A code replacement mapping associates a conceptual representation of a function or operator that is familiar to the code generator with a custom implementation representation that specifies a C or C++ replacement function prototype. You capture a mapping as an entry in a code replacement table:

- Interactively, by using the Code Replacement Tool.
- Programmatically, by using a MATLAB programming interface.

The following table lists situations to help you decide when to use each approach.

Situation	Approach
Defining mappings for the first time.	Code Replacement Tool.
Rapid prototyping mappings.	Code Replacement Tool to quickly generate, register, and test mappings.
Developing a mapping as a template or starting point for defining similar mappings.	Code Replacement Tool to generate definition code that you can copy and modify.
Modifying a registration file, including copying and pasting content.	MATLAB Editor to update the programming interface directly.
Defining mappings that specify attributes not available from the Code Replacement Tool (for example, sets of algorithm parameters).	Programming interface.

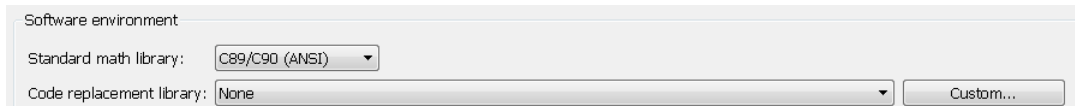
Define Mappings Interactively with the Code Replacement Tool

This example shows how to use the Code Replacement Tool to develop code replacement mappings. The tool is ideal for getting started with developing mappings, rapid prototyping, and developing a mapping to use as a starting point for defining similar mappings.

Open the Code Replacement Tool

Do one of the following:

- In the Command Window, enter the command `crtool`.
- In the Simulink Editor, open the Configuration Parameters dialog box and navigate to the **Code Generation > Interface >** pane. To the right of the **Code replacement library** parameter, click the **Custom**.



The **Custom** button is available only for ERT-based targets. Although an Embedded Coder license is not required to create a custom code replacement library, you must have an Embedded Coder license to use a such a library.

By default, the tool displays, left to right, a root pane, a list pane, and a dialog pane. You can manipulate the display:

- Drag boundaries to widen, narrow, shorten, or lengthen panes, and to resize table columns.
- Select **View > Show dialog pane** to hide or display the rightmost pane.
- Click a table column heading to sort the table based on contents of the selected column.
- Right-click a table column heading and select **Hide** to remove the column from the display. (You cannot hide the **Name** column.)

Create a Code Replacement Table

- 1 In the Code Replacement Tool window, select **File > New table**.

- 2 In the right pane, name the table and click **Apply**. Later, when you save the table, the tool uses the table name that you specify to name the file. For example, if you enter the name `my_sinfcn`, the tool names the file `my_sinfcn.m`.

Create Table Entries

Create one or more table entries. Each entry maps the conceptual representation of a function or operator to your implementation representation. The information that you enter depends on the type of entry you create. You must enter the following information:

- 1 In the left pane, select the table to which you want to add the entry.
- 2 Select **File > New entry > entry-type**, where **entry-type** is one of:
 - Math Operation
 - Function
 - BLAS Operation
 - CBLAS Operation
 - Net Slope Fixed Point Operation
 - Semaphore entry
 - Customization entry

The new entry appears in the middle pane, initially without a name.

- 3 In the middle pane, select the new entry.
- 4 In the right pane, on the **Mapping Information** tab, from the **Function** or **Operation** menu, select the function or operation that you want the code generator to replace. Regardless of the entry type, you must make a selection from this menu. Your selection determines what other information you specify.

With the exception of customization entries, you also specify information for your replacement function prototype. You can also specify implementation attributes, such as the rounding modes to apply.

- 5 If prompted, specify additional entry information that you want the code generator to use when searching for a match. For example, when you select an addition or subtraction operation, the tool prompts you to specify an algorithm (**Cast before operation** or **Cast after operation**).
- 6 Review the conceptual argument information that the tool populates for the function or operation. Conceptual input and output arguments represent arguments for

the function or operator being replaced. Conceptual arguments observe naming conventions ('y1', 'u1', 'u2', ...) and data types familiar to the code generator.

If you do not want the data types for your implementation to be the same as the conceptual argument types, clear the **Make the conceptual and implementation argument types the same** check box. For example, most ANSI-C functions operate on and return `double` data. Clear the check box if want to map a conceptual representation of the function, such as `sin`, which specifies an argument and return value of type `single` (`single sin(single)`), to an implementation representation that specifies an argument and return value of type `double` (`double sin(double)`). In this case, the code generator produces the following code:

```
y = (single) sin(u1);
```

If you select `Custom` for a function entry, specify only conceptual argument information.

- 7 Specify the name and argument information for your replacement function. As you enter the information and click **Apply**, the tool updates the **Function signature preview**.
- 8 Specify additional implementation attributes that apply. For example, depending on the type and name of the entry that you specify, the tool prompts you to specify:
 - Integer saturation mode
 - Rounding modes
 - Whether to allow inputs that include expressions
 - Whether a function modifies internal or global state
- 9 Click **Apply**.

Validate Tables and Entries

The Code Replacement Tool provides a way to validate the syntax of code replacement tables and table entries as you define them. If the tool finds validation errors, you can address them and retry the validation. Repeat the process until the tool does not report errors.

To...	Do...
Validate table entries	Select an entry, scroll to the bottom of the Mapping Information tab, and click Validate entry .

To...	Do...
	Alternatively, select one or more entries, right-click, and select Validate entries .
Validate a table	Select the table. Then, select Actions > Validate table .

Save a Table

When you save a table, the tool validates unvalidated content.

- 1 Select **File > Save table**.
- 2 In the Browse For Folder dialog box, specify a location and name for the file. Typically, you select a location on the MATLAB path. By default, the tool names the file using the name that you specify for the table with the extension `.m`.
- 3 Click **Save**.

Open and Modify Tables

After you save a code replacement table, to make changes:

- 1 Select **File > Open table**.
- 2 In the Import file dialog box, browse to the MATLAB file that contains the table.

Repeat the sequence to open and work on multiple tables.

If you open multiple tables, you can manage the tables together. For example, use the tool to:

- Create new table entries.
- Delete entries.
- Copy and paste or cut and paste information between tables.

Define Mappings Programmatically

This example shows how to define a code replacement mapping programmatically. The programming interface for defining code replacement table mappings is ideal for

- Modifying tables created with the Code Replacement Tool.
- Defining mappings for specialized entries that you cannot create with the Code Replacement Tool.

- Replicating and modifying similar entries and tables.

Create a Code Replacement Table

- 1 Create a table definition file that contains a function definition. For example:

```
function hTable = crl_table_sinfcn()
```

- 2 Within the function body, create the table by calling the function `RTW.Tf1Table`.

```
hTable = RTW.Tf1Table;
```

Create a Table Entry

For each function or operator that you want the code generator to replace, map a conceptual representation of the function or operator to an implementation representation as a table entry.

- 1 Within the body of a table definition file, create a code replacement entry object. Call one of the following functions:

Entry Type	Function
Math operation	<code>RTW.Tf1COperationEntry</code>
Function	<code>RTW.Tf1CFunctionEntry</code>
BLAS operation	<code>RTW.Tf1BlasEntryGenerator</code>
CBLAS operation	<code>RTW.Tf1CBlasEntryGenerator</code>
Fixed-point addition and subtraction operations (support for <code>SlopesMustBeTheSame</code> and <code>ZeroNetBias</code> parameters)	<code>RTW.Tf1COperationEntryGenerator</code>
Net slope fixed-point operation	<code>RTW.Tf1COperationEntryGenerator_NetSlope</code>
Semaphore or mutex entry	<code>RTW.Tf1CSemaphoreEntry</code>
Custom function entry	<code>MyCustomFunctionEntry</code> (where <code>MyCustomFunctionEntry</code> is a class derived from <code>RTW.Tf1CFunctionEntryML</code>)
Custom operation entry	<code>MyCustomOperationEntry</code> (where <code>MyCustomOperationEntry</code> is a class derived from <code>RTW.Tf1COperationEntryML</code>)

For example:

```
hEnt = RTW.Tf1CFunctionEntry;
```

You can combine steps of creating the entry, setting entry parameters, creating conceptual and implementation arguments, and adding the entry to a table with a single function call to `registerCFunctionEntry`, `registerCPPFunctionEntry`, or `registerCPromotableMacroEntry` if you are creating an entry for a function and the function implementation meets the following criteria:

- Implementation argument names and order match the names and order of corresponding conceptual arguments.
- Input arguments are of the same type.
- The return and input argument names follow the code generator's default naming conventions:
 - Return argument is `y1`.
 - Input arguments are `u1`, `u2`, ..., `un`.

For example:

```
registerCFunctionEntry(hTable, 100, 1, 'sin', 'double', ...
    'sin_dbl', 'double', 'sin_dbl.h', '', '');
```

Set Entry Parameters

Set entry parameters, such as the priority, algorithm information, and implementation (replacement) function name. Call the function listed in the following table for the entry type that you created.

Entry Type	Function
Math operation	setTf1COperationEntryParameters
Function	setTf1CFunctionEntryParameters
BLAS operation	setTf1COperationEntryParameters
CBLAS operation	setTf1COperationEntryParameters
Fixed-point addition and subtraction operations where there is a many-to-one mapping, such as a mapping	setTf1COperationEntryParameters

Entry Type	Function
for a range of fixed-point types to the same replacement function (support for SlopesMustBeTheSame and ZeroNetBias parameters)	
Net slope fixed-point operation	setTf1COperationEntryParameters
Semaphore or mutex entry	setTf1CSemaphoreEntryParameters
Custom function entry	setTf1CFunctionEntryParameters
Custom operation entry	setTf1COperationEntryParameters

To see a list of the parameters that you can set, at the command line, create a new entry and omit the semicolon at the end of the command. For example:

```
hEnt = RTW.Tf1CFunctionEntry
```

```
hEnt =
```

```
Tf1CFunctionEntry with properties:
```

```

    Implementation: [1x1 RTW.CImplementation]
    SlopesMustBeTheSame: 0
    BiasMustBeTheSame: 0
    AlgorithmParams: []
        ImplType: 'FCN_IMPL_FUNCT'
    AdditionalHeaderFiles: {0x1 cell}
    AdditionalSourceFiles: {0x1 cell}
        OtherFiles: {0x1 cell}
    AdditionalIncludePaths: {0x1 cell}
    AdditionalSourcePaths: {0x1 cell}
        AdditionalLinkObjs: {0x1 cell}
    AdditionalLinkObjsPaths: {0x1 cell}
        AdditionalLinkFlags: {0x1 cell}
    AdditionalCompileFlags: {0x1 cell}
        SearchPaths: {0x1 cell}
        Key: ''
        Priority: 100
    ConceptualArgs: [0x1 handle]
    EntryInfo: []
    GenCallback: ''
    GenFileName: ''
    SaturationMode: 'RTW_SATURATE_UNSPECIFIED'
```

```

        RoundingModes: {'RTW_ROUND_UNSPECIFIED'}
    TypeConversionMode: 'RTW_EXPLICIT_CONVERSION'
        AcceptExprInput: 1
            SideEffects: 0
            UsageCount: 0
        RecordedUsageCount: 0
            Description: ''
    StoreFcnReturnInLocalVar: 0
        TraceManager: [1x1 RTW.Tf1TraceManager]
    
```

To see the implementation parameters, enter:

```
hEnt.Implementation
```

```
ans =
```

```
CImplementation with properties:
```

```

        HeaderFile: ''
        SourceFile: ''
        HeaderPath: ''
        SourcePath: ''
        Return: []
    StructFieldMap: []
        Name: ''
        Arguments: [0x1 handle]
    ArgumentDescriptor: []
    
```

For example, to set entry parameters for the `sin` function and name your replacement function `sin_dbl`, use the following function call:

```

setTf1CFunctionEntryParameters(hEnt, ...
    'Key', 'sin', ...
    'ImplementationName', 'sin_dbl');
    
```

Create Conceptual Arguments

Create conceptual arguments and add them to the entry's array of conceptual arguments.

- Specify output arguments before input arguments.
- Specify argument names that comply with code generator argument naming conventions:
 - `y1` for a return argument
 - `u1, u2, ..., un` for input arguments

- Specify data types that are familiar to the code generator.
- The function signature, including argument naming, order, and attributes, must fulfill the signature match sought by function or operator callers.
- The code generator determines the size of the value for an argument with an unsized type, such as integer, based on hardware implementation configuration settings.

For each argument:

- 1 Identify whether the argument is for input or output, the name, and data type. If you do not know what arguments to specify for a supported function or operation, use the Code Replacement Tool to find them. For example, to find the conceptual arguments for the `sin` function, open the tool, create a table, create a function entry, and in the **Function** menu select `sin`.
- 2 Create and add the conceptual argument to an entry. You can choose a method from the methods listed in this table.

If	Then
You want simpler code or want to explicitly specify whether the argument is scalar or nonscalar (vector or matrix).	<p>Call the function <code>createAndAddConceptualArg</code>. For example:</p> <pre>createAndAddConceptualArg(hEnt, ... 'RTW.Tf1ArgNumeric', ... 'Name', 'y1',... 'IOType', 'RTW_IO_OUTPUT',... 'DataTypeMode', 'double');</pre> <p>The second argument specifies whether the argument is scalar (<code>RTW.Tf1ArgNumeric</code> or <code>RTW.Tf1ArgMatrix</code>).</p>
You want to create an argument based on a built-in argument definition (for example, scalar or nonscalar).	<p>Call <code>getTf1ArgFromString</code> to create the argument. Then, call <code>addConceptualArg</code> to add the argument to the entry.</p> <pre>arg = getTf1ArgFromString(hEnt, 'y1', 'double'); arg.IOType = 'RTW_IO_OUTPUT'; addConceptualArg(hEnt, arg);</pre>

The following code shows the conceptual output and input argument definitions for the `sin` function:

```
% Conceptual Args
```

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

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

Create Implementation Arguments

Create implementation arguments for the C or C++ replacement function and add them to the entry.

- When replacing code, the code generator uses the argument names to determine how it passes data to the implementation function.
- For function replacements, the order of implementation argument names must match the order of the conceptual argument names.
- For operator replacements, the order of implementation argument names do not have to match the order of the conceptual argument names. For example, for an operator replacement for addition, $y1=u1+u2$, the conceptual arguments are $y1$, $u1$, and $u2$, in that order. If the signature of your implementation function is `t myAdd(t u2, t u1)`, where `t` is a valid C type, based on the argument name matches, the code generator passes the value of the first conceptual argument, $u1$, to the second implementation argument of `myAdd`. The code generator passes the value of the second conceptual argument, $u2$, to the first implementation argument of `myAdd`.
- For operator replacements, you can remap operator output arguments to implementation function input arguments.

For each argument:

- 1 Identify whether the argument is for input or output, the name, and the data type.
- 2 Create and add the implementation argument to an entry. You can choose a method from the methods listed in this table.

If	Then
You want to populate implementation arguments as copies of previously created matching conceptual arguments	Call the function <code>copyConceptualArgsToImplementation</code> . For example: <code>copyConceptualArgsToImplementation(hEnt);</code>

If	Then
You want to create and add implementation arguments individually, or vary argument attributes, while maintaining conceptual argument order	<p>Call functions <code>createAndSetCImplementationReturn</code> and <code>createAndAddImplementationArg</code> . For example:</p> <pre>createAndSetCImplementationReturn(hEnt, '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);</pre>

If	Then
<p>You want to minimize the amount of code, or specify constant arguments to pass to the implementation function</p>	<p>Create the argument with a call to the function <code>getTf1ArgFromString</code>. Then, use the convenience method <code>setReturn</code> or <code>addArgument</code> to specify whether an argument is a return value or argument and to add the argument to the entry's array of implementation arguments. For example:</p> <pre>arg = getTf1ArgFromString(hEnt, 'y1', 'double'); arg.IOType = 'RTW_IO_OUTPUT'; hEnt.Implementation.setReturn(arg); arg = getTf1ArgFromString(hEnt, 'u1', 'double'); hEnt.Implementation.addArgument(arg);</pre> <p>The following call to <code>getTf1ArgFromString</code> passes the constant 0 to argument <code>u2</code>:</p> <pre>arg = getTf1ArgFromString(hEnt, 'u2', 'int16', 0); hEnt.Implementation.addArgument(arg);</pre> <p>For semaphore and mutex entries, use the functions <code>getTf1DWorkFromString</code> and <code>addDWorkArg</code> to create and add a <code>DWork</code> argument to the entry. Then create implementation arguments as shown above with <code>getTf1ArgFromString</code> and the convenience methods <code>setReturn</code> and <code>addArgument</code>. For example:</p> <pre>arg = getTf1DWorkFromString('d1', 'void*'); hEnt.addDWorkArg(arg); arg = hEnt.getTf1ArgFromString('y1', 'void'); arg.IOType = 'RTW_IO_OUTPUT'; hEnt.Implementation.setReturn(arg); arg = hEnt.getTf1ArgFromString('u1', 'integer'); hEnt.Implementation.addArgument(arg); arg = hEnt.getTf1ArgFromString('d1', 'void**'); hEnt.Implementation.addArgument(arg);</pre>

The following code shows the implementation output and input argument definitions for the `sin` function:

```
% Implementation Args

arg = hEnt.getTf1ArgFromString('y1', 'double');
arg.IOType = 'RTW_IO_OUTPUT';
hEnt.Implementation.setReturn(arg);

arg = hEnt.getTf1ArgFromString('u1', 'double');
hEnt.Implementation.addArgument(arg);
```

Add an Entry to a Table

Add an entry to a code replacement table by calling the function `addEntry`.

```
addEntry(hTable, hEnt);
```

Validate an Entry

After you create or modify a code replacement table entry, validate it by invoking it at the MATLAB command line. For example:

```
hTbl = crl_table_sinfcn

hTbl =

RTW.Tf1Table
  Version: '1.0'
 AllEntries: [2x1 RTW.Tf1CFunctionEntry]
ReservedSymbols: []
StringResolutionMap: []
```

If the table includes errors, MATLAB reports them. The following examples shows how MATLAB reports a typo in a data type name:

```
hTbl = crl_table_sinfcn

??? RTW_CORE:tf1:Tf1Table: Unsupported data type, 'dooble'.

Error in ==> crl_table_sinfcn at 7
hTable.registerCFunctionEntry(100, 1, 'sin', 'dooble', 'sin_dbl', ...
```

Save a Table

Save the table definition file. Use the name of the table definition function to name the file, for example, `crl_table_sinfcn.m`.

Related Examples

- “Identify Code Replacement Requirements”
- “Prepare for Code Replacement Library Development”
- “Specify Build Information for Replacement Code”
- “Register Code Replacement Mappings”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”
- “Math Function Code Replacement”
- “Memory Function Code Replacement”
- “Nonfinite Function Code Replacement”
- “Semaphore and Mutex Function Replacement”
- “Algorithm-Based Code Replacement”
- “Lookup Table Function Code Replacement”
- “Data Alignment for Code Replacement”
- “Replace MATLAB Functions with Custom Code Using `coder.replace`”
- “Replace MATLAB Functions Specified in MATLAB Function Blocks”
- “Customize Matching and Replacement Process for Functions”
- “Scalar Operator Code Replacement”
- “Addition and Subtraction Operator Code Replacement”
- “Small Matrix Operation to Processor Code Replacement”
- “Matrix Multiplication Operation to MathWorks BLAS Code Replacement”
- “Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement”
- “Remap Operator Output to Function Input”
- “Customize Matching and Replacement Process for Operators”
- “Fixed-Point Operator Code Replacement”
- “Binary-Point-Only Scaling Code Replacement”

- “Slope Bias Scaling Code Replacement”
- “Net Slope Scaling Code Replacement”
- “Equal Slope and Zero Net Bias Code Replacement”
- “Data Type Conversions (Casts) and Operator Code Replacement”
- “Shift Left Operations and Code Replacement”

More About

- “Code You Can Replace From Simulink Models”
- “Code Replacement Libraries”
- “Code Replacement Terminology”
- “Code Replacement Customization Limitations”

External Web Sites

- “Replacing Math Functions and Operators”

Specify Build Information for Replacement Code

In this section...

“Specifying Build Information” on page 21-58

“Specify Build Information Interactively with the Code Replacement Tool” on page 21-59

“Specify Build Information Programmatically” on page 21-61

Specifying Build Information

A code replacement table entry can specify build information for the code generator to use when replacing code for a match. The build information can include:

- Paths and file names for header files
- Paths and file names for source files
- Paths and file names for object files
- Compile flags
- Link flags

Add build information to an entry:

- Interactively, by using the Code Replacement Tool.
- Programmatically, by using a MATLAB programming interface.

The following table lists situations to help you decide when to use each approach.

Situation	Approach
Creating code replacement entries for the first time.	Code Replacement Tool.
You used the Code Replacement Tool to create the entries for which the build information applies.	Code Replacement Tool to quickly specify the build information.
Rapid prototyping entries.	Code Replacement Tool to quickly generate, register, and test entries.

Situation	Approach
Developing an entry to use as a template or starting point for defining similar entries.	Code Replacement Tool to generate entry code that you can copy and modify.
Modifying existing mappings.	MATLAB Editor to update the programming interface directly.

As you specify build information, consider the following information:

- If you use the programming interface, paths that you specify can include tokens. A token is a variable defined as a string or cell array of strings in the MATLAB workspace that you enclose with dollar signs (*\$variable\$*). The code generator evaluates and replaces a token with the defined value. For example, consider the path *\$myfolder\$*\folder1, where *myfolder* is a string variable defined in the MATLAB workspace as 'd:\work\source\module1'. The code generator generates the custom path as d:\work\source\module1\folder1.
- If an entry uses header, source, or object files that reside in a folder other than the build folder, you must copy the files to the build folder before the code generator can build the code. The Code Replacement Tool and programming information provide a way to specify this copy operation.

Specify Build Information Interactively with the Code Replacement Tool

The Code Replacement Tool provides a quick, easy way for you to specify build information for code replacement table entries. It is ideal for getting started with defining a table entry, rapid prototyping, and developing table entries to use as a starting point for defining similar mappings.

- 1 Determine the information that you must specify.
- 2 Open the Code Replacement Tool.
- 3 Select the code replacement table entry for which you want to specify the build information. In the left pane, select the table that contains the entry. In the middle pane, select the entry that you want to modify.
- 4 In the right pane, select the **Build Information** tab.
- 5 On the **Build Information** tab, specify your build information.

Field	Specify
Implementation header file	File name and extension for the header file the code generator needs to generate the replacement code. For example, <code>sin_dbl.h</code> .
Implementation source file	File name and extension for the C or C++ source file the code generator needs to generate the replacement code. For example, <code>sin_dbl.c</code> .
Additional header files/include paths	Paths and file names for additional header files the code generator needs to generate the replacement code. For example, <code>C:\libs\headerFiles</code> and <code>C:\libs\headerFiles\common.h</code> .
Additional source files/ paths	Paths and file names for additional source files the code generator needs to generate the replacement code. For example, <code>C:\libs\srcFiles</code> and <code>C:\libs\srcFiles\common.c</code> .
Additional object files/ paths	Paths and file names for additional object files the linker needs to build the replacement code. For example, <code>C:\libs\objFiles</code> and <code>C:\libs\objFiles\common.obj</code> .
Additional link flags	Flags the linker needs to generate an executable file for the replacement code.
Additional compile flags	Flags the compiler needs to generate object code for the replacement code.
Copy files to build directory	Whether the code generator needs to copy header, source, or object files from an external location to the build folder before building the replacement code.

- 6 Click **Apply**.
- 7 Select the **Mapping Information** tab. Scroll to the bottom of that table and click **Validate entry**. The tool validates the changes that you made to the entry.
- 8 Save the table that includes the entry that you just modified.

Specify Build Information Programmatically

The programming interface for specifying build information for a code replacement entry is ideal for:

- Modifying entries created with the Code Replacement Tool.
- Replicating and then modifying similar entries and tables.

The basic workflow for specifying build information programmatically is:

- 1 Identify or create the code replacement entry that you want to specify the build information.
- 2 Determine what information to specify.
- 3 Specify your build information.

Specify	Action
Implementation header file	<p>Use one of the following:</p> <ul style="list-style-type: none"> • Set properties <code>ImplementationHeaderFile</code> and <code>ImplementationHeaderPath</code> in a call to <code>setTf1CFunctionEntryParameters</code>, <code>setTf1COperationEntryParameters</code>, or <code>setTf1CSemaphoreEntryParameters</code>. For example: <pre>setTf1CFunctionEntryParameters(hEnt, ... 'ImplementationHeaderFile', 'sin_dbl.h', ... 'ImplementationHeaderPath', 'D:/Lib/headerFiles' 'Key', 'sin', ... 'ImplementationName', 'sin_dbl');</pre> <ul style="list-style-type: none"> • Set argument <code>headerFile</code> in a call to <code>registerCFunctionEntry</code>, <code>registerCPPFunctionEntry</code>, or <code>registerCPromotableMacroEntry</code>
Implementation source file	<p>Set properties <code>ImplementationSourceFile</code> and <code>ImplementationSourcePath</code> in a call to <code>setTf1CFunctionEntryParameters</code>, <code>setTf1COperationEntryParameters</code>, or <code>setTf1CSemaphoreEntryParameters</code>. For example:</p> <pre>setTf1CFunctionEntryParameters(hEnt, ... 'ImplementationHeaderFile', 'sin_dbl.c', ...</pre>

Specify	Action
	<pre data-bbox="457 296 1144 378">'ImplementationHeaderPath', 'D:/lib/sourceFiles' 'Key', 'sin', ... 'ImplementationName', 'sin_dbl');</pre>
Additional header files/include paths	<p data-bbox="397 395 1273 487">For each file, specify the file name and path in calls to the functions <code>addAdditionalHeaderFile</code> and <code>addAdditionalIncludePath</code>. For example:</p> <pre data-bbox="397 517 1176 687">libdir = fullfile('\$MATLAB_ROOT'),'..', '..', 'lib'); hEnt = RTW.Tf1CFunctionEntry; addAdditionalHeaderFile(hEnt, 'common.h'); addAdditionalIncludePath(hEnt, fullfile(libdir, 'include'));</pre>
Additional source files/paths	<p data-bbox="397 703 1258 795">For each file, specify the file name and path in calls to the functions <code>addAdditionalSourceFile</code> and <code>addAdditionalSourcePath</code>. For example:</p> <pre data-bbox="397 824 1176 994">libdir = fullfile('\$MATLAB_ROOT'),'..', '..', 'lib'); hEnt = RTW.Tf1CFunctionEntry; addAdditionalSourceFile(hEnt, 'common.c'); addAdditionalSourcePath(hEnt, fullfile(libdir, 'src'));</pre>
Additional object files/paths	<p data-bbox="397 1010 1233 1102">For each file, specify the file name and path in calls to the functions <code>addAdditionalLinkObj</code> and <code>addAdditionalLinkObjPath</code>. For example:</p> <pre data-bbox="397 1131 1206 1302">libdir = fullfile('\$MATLAB_ROOT'),'..', '..', 'lib'); hEnt = RTW.Tf1CFunctionEntry; addAdditionalLinkObj(hEnt, 'sin.o'); addAdditionalLinkObjPath(hEnt, fullfile(libdir, 'bin'));</pre>
Compile flags	<p data-bbox="397 1317 1334 1376">Set the entry property <code>AdditionalCompileFlags</code> to a cell array of strings representing the required compile flags. For example:</p> <pre data-bbox="397 1406 1132 1487">hEnt = RTW.Tf1CFunctionEntry; hEnt.AdditionalCompileFlags = {'-Zi -Wall', '-O3'};</pre>

Specify	Action
Link flags	Set the entry property <code>AdditionalLinkFlags</code> to a cell array of strings representing the required link flags. For example: <pre>hEnt = RTW.Tf1CFunctionEntry; hEnt.AdditionalCompileFlags = {'-MD -Gy', '-T'};</pre>
Copy of header, source, and link object files that are specified in an entry to the build folder	Use one of the following: <ul style="list-style-type: none"> Set property <code>GenCallback</code> to <code>'RTW.copyFileToBuildDir'</code> in a call to <code>setTf1CFunctionEntryParameters</code>, <code>setTf1COperationEntryParameters</code>, or <code>setTf1CSemaphoreEntryParameters</code>. For example: <pre>setTf1CFunctionEntryParameters(hEnt, ... 'ImplementationHeaderFile', 'sin_dbl.h', ... 'ImplementationHeaderPath', 'D:/lib/headerFiles' 'Key', 'sin', ... 'ImplementationName', 'sin_dbl' 'GenCallback', 'RTW.copyFileToBuildDir');</pre> Set argument <code>genCallback</code> in a call to <code>registerCFunctionEntry</code>, <code>registerCPPFunctionEntry</code>, or <code>registerCPromotableMacroEntry</code> to <code>'RTW.copyFileToBuildDir'</code>. After generating the C or C++ code, the code generator calls the function <code>RTW.copyFileToBuildDir</code> . The specified files are then available for the remaining steps of the build process.

4 Save the table that includes the entry that you added or modified.

The following example defines a table entry for an optimized multiplication function that takes signed 32-bit integers and returns a signed 32-bit integer, taking saturation into account. Multiplications in the generated code are replaced with calls to the optimized function. The optimized function does not reside in the build folder. For the code generator to access the files, they must be copied into the build folder to be compiled and linked into the application.

The table entry specifies the source and header file names and paths. To request the copy operation, the table entry sets the `genCallback` to `'RTW.copyFileToBuildDir'` in the call to the `setTf1COperationEntryParameters` function. In this example,

the header file `s32_mul.h` contains an inlined function that invokes assembly functions contained in `s32_mul.s`. If a match occurs for the table entry, the function `RTW.copyFileToBuildDir` copies the specified source and header files to the build folder.

```
function hTable = make_my_crl_table

hTable = RTW.Tf1Table;

op_entry = RTW.Tf1COperationEntry;
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_MUL', ...
    'Priority', 100, ...
    'SaturationMode', 'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes', {'RTW_ROUND_UNSPECIFIED'}, ...
    'ImplementationName', 's32_mul_s32_s32_sat', ...
    'ImplementationHeaderFile', 's32_mul.h', ...
    'ImplementationSourceFile', 's32_mul.s', ...
    'ImplementationHeaderPath', {fullfile('${MATLAB_ROOT}','crl')}, ...
    'ImplementationSourcePath', {fullfile('${MATLAB_ROOT}','crl')}, ...
    'GenCallback', 'RTW.copyFileToBuildDir');
.
.
.
addEntry(hTable, op_entry);
```

The following example uses the functions `addAdditionalHeaderFile`, `addAdditionalIncludePath`, `addAdditionalSourceFile`, `addAdditionalSourcePath`, `addAdditionalLinkObj`, and `addAdditionalLinkObjPath` with the code generation callback function `RTW.copyFileToBuildDir`.

```
hTable = RTW.Tf1Table;

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

op_entry = RTW.Tf1COperationEntry;
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_ADD', ...
    'Priority', 90, ...
    'SaturationMode', 'RTW_SATURATE_UNSPECIFIED', ...
    'RoundingModes', {'RTW_ROUND_UNSPECIFIED'}, ...
    'ImplementationName', 's32_add_s32_s32', ...
    'ImplementationHeaderFile', 's32_add_s32_s32.h', ...
    'ImplementationSourceFile', 's32_add_s32_s32.c'...
    'GenCallback', 'RTW.copyFileToBuildDir');

addAdditionalHeaderFile(op_entry, 'all_additions.h');
addAdditionalIncludePath(op_entry, fullfile(libdir, 'include'));
addAdditionalSourceFile(op_entry, 'all_additions.c');
addAdditionalSourcePath(op_entry, fullfile(libdir, 'src'));
```

```
addAdditionalLinkObj(op_entry, 'addition.o');  
addAdditionalLinkObjPath(op_entry, fullfile(libdir, 'bin'));  
.  
.  
addEntry(hTable, op_entry);
```

Related Examples

- “Identify Code Replacement Requirements”
- “Prepare for Code Replacement Library Development”
- “Define Code Replacement Mappings”
- “Register Code Replacement Mappings”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “What Is Code Replacement Customization?”
- “Code Replacement Libraries”
- “Code Replacement Terminology”
- “Code Replacement Customization Limitations”

External Web Sites

- “Replacing Math Functions and Operators”

Register Code Replacement Mappings

In this section...

“Code Replacement Library Registration” on page 21-66

“Create Registration File Interactively with the Code Replacement Tool” on page 21-67

“Create Registration File Programmatically” on page 21-69

“Register a Code Replacement Library” on page 21-71

“Registration Files That Define Multiple Code Replacement Libraries” on page 21-71

“Registration Files That Define Code Replacement Library Hierarchies” on page 21-72

Code Replacement Library Registration

After you define code replacement entries in a code replacement table, you can include the table in a code replacement library that you register with the code generator. When registered, a library appears in the list of available code replacement libraries that you can choose from when configuring the code generator.

Register a code replacement table as a code replacement library:

- Interactively, by using the Code Replacement Tool
- Programmatically, by using a MATLAB programming interface

The following table lists situations when you might consider one approach over the other.

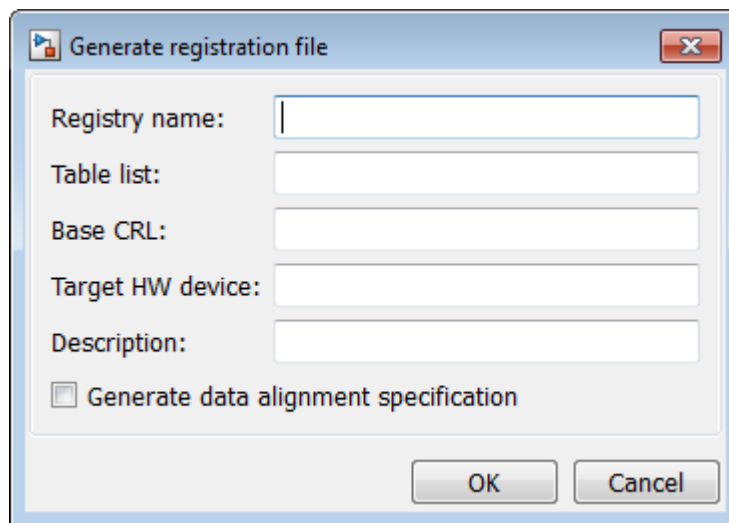
If...	Then...
Registering a code replacement table for the first time	Use the Code Replacement Tool.
You used the Code Replacement Tool to create the table	Use the Code Replacement Tool to quickly register the table.
Rapid prototyping code replacement	Use the Code Replacement Tool to quickly generate, register, and test entries.
Creating registration file to use as a template or starting point for defining similar registration files	Use the Code Replacement Tool to generate code that you can copy and modify.

If...	Then...
Modifying existing registration files	Use the MATLAB Editor to update the registration file.
Defining multiple code replacement libraries in one registration file	Use the MATLAB Editor to create a new or extend an existing registration file.
Defining code replacement library hierarchy in a registration file	Use the MATLAB Editor to create a new or extend an existing registration file.

Create Registration File Interactively with the Code Replacement Tool

The Code Replacement tool provides a quick, easy way for you to create a registration file for a code replacement table. It is ideal for getting started, rapid prototyping, and generating a registration file that you want to use as a starting point for similar registrations.

- 1 After you validate and save a code replacement table, select **File > Generate registration file** to open the **Generate registration file** dialog box.



- 2 Enter the registration information. Minimally, specify:

For...	Specify...
Registry name	String naming the code replacement library. For example, <code>Sin Function Example</code> .
Table list	<p>Strings naming one or more code replacement tables to include in the library. Specify each table as one of the following:</p> <ul style="list-style-type: none"> • Name of a table file on the MATLAB search path • Absolute path to a table file • Path to a table file relative to <code>\$(MATLAB_ROOT)</code> <p>You can specify multiple tables. If you do, separate the table specifications with a comma. For example:</p> <pre>crl_table_sinfcn, c:/work_crl/crl_table_muldiv</pre> <p>See “Registration Files That Define Multiple Code Replacement Libraries” on page 21-71 for examples of each type of table specification.</p>

Optionally, you can specify:

For...	Specify...
Description	Text string that describes the purpose and content of the library.
Target HW device	Strings naming one or more hardware devices the code replacement library supports. Separate names with a comma. To support all device types, enter an asterisk (*). For example, <code>TI C28x, TI C62x</code> .
Base CRL	String naming a code replacement library that you want to serve as a base library for the library you are registering. Use this field to specify library hierarchies. For example, you can specify a general <code>TI device library</code> as the base library for a more specific <code>TI C28x device library</code> .

For...	Specify...
Generate data alignment specification	Flag that enables data alignment specification.

Create Registration File Programmatically

The programming interface for creating a registration file for a code replacement table is ideal for:

- Modifying registration files created with the Code Replacement Tool
- Replicating and modifying similar registration files
- Defining multiple code replacement libraries in one registration file

The basic workflow for creating a registration file programmatically consists of the following steps:

- 1 Define an `rtwTargetInfo` function. The code generator recognizes this function as a customization file. The function definition must include at least the following content:

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@loc_register_tf1);

function this = loc_register_tf1

this(1) = RTW.Tf1Registry;
this(1).Name = 'crl-name';
this(1).TableList = {'table',...};
```

For...	Replace...
<code>this(1).Name = 'crl-name';</code>	<code>crl-name</code> with a string naming the code replacement library. For example, <code>Sin Function Example</code> .
<code>this(1).TableList = {'table',...};</code>	<code>table</code> with a string that identifies the code replacement table that contains your code replacement entries. Specify a table as one of the following:

For...	Replace...
	<ul style="list-style-type: none"> • Name of a table file on the MATLAB search path • Absolute path to a table file • Path to a table file relative to \$(MATLAB_ROOT) <p>You can specify multiple tables. If you do, separate the table specifications with commas.</p> <p>See “Registration Files That Define Multiple Code Replacement Libraries” on page 21-71 for examples of each type of table specification.</p>

Optionally, you can specify:

For...	Replace...
<code>this(1).Description = 'text'</code>	<code>text</code> with a string that describes the purpose and content of the library.
<code>this(1).TargetHWDeviceType = {'device-type',...}</code>	<code>device-type</code> with a string that names a hardware device the code replacement library supports. You can specify multiple device types. Separate device types with a comma. For example, TI C28x, TI C62x. To support all device types, enter an asterisk (*).
<code>this(1).BaseTfl = 'base-lib'</code>	<code>base-lib</code> with a string that names a code replacement library that you want to serve as a base library for the library you are registering. Use this field to specify library hierarchies. For example, you can specify a general TI device library as the base library for a TI C28x device library.

For example:

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@loc_register_tfl);

function this = loc_register_tfl

this(1) = RTW.TflRegistry;
this(1).Name = 'Sin Function Example';
this(1).TableList = {'crl_table_sinfcn'};
this(1).TargetHWDeviceType = {'*'};
this(1).Description = 'Example - sin function replacement';
```

- 2 Save the file with the name `rtwTargetInfo.m`.
- 3 Place the file on the MATLAB path. When the file is on the MATLAB path, the code generator reads the file after starting and applies the customizations during the current MATLAB session.

Register a Code Replacement Library

Before you can use the code replacement tables defined in a registration file, you must refresh Simulink customizations within the current MATLAB session. To initiate a refresh, enter the following command:

```
sl_refresh_customizations
```

Registration Files That Define Multiple Code Replacement Libraries

Use the programming interface to create a registration file that defines multiple code replacement libraries. The following example defines multiple code replacement libraries. The `TableList` fields specify code replacement tables that reside at different locations. The tables reside on the MATLAB search path or at locations specified using path strings.

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@locCrlRegFcn);

function thisCrl = locCrlRegFcn

% Register a code replacement library for use with model: rtwdemo_crladdsub
thisCrl(1) = RTW.TflRegistry;
```

```

thisCr1(1).Name = 'Addition & Subtraction Examples';
thisCr1(1).Description = 'Example of addition/subtraction op replacement';
thisCr1(1).TableList = {'crl_table_addsub'};
thisCr1(1).TargetHWDeviceType = {'*'};

% Register a code replacement library for use with model: rtwdemo_crlmuldiv
thisCr1(2) = RTW.TflRegistry;
thisCr1(2).Name = 'Multiplication & Division Examples';
thisCr1(2).Description = 'Example of mult/div op repl for built-in integers';
thisCr1(2).TableList = {'c:/work_crl/crl_table_muldiv'};
thisCr1(2).TargetHWDeviceType = {'*'};

% Register a code replacement library for use with model: rtwdemo_crlfixpt
thisCr1(3) = RTW.TflRegistry;
thisCr1(3).Name = 'Fixed-Point Examples';
thisCr1(3).Description = 'Example of fixed-point operator replacement';
thisCr1(3).TableList = {fullfile('$MATLAB_ROOT'), ...
    'toolbox', 'rtw', 'rtwdemos', 'crl_demo', 'crl_table_fixpt'};
thisCr1(3).TargetHWDeviceType = {'*'};

```

Registration Files That Define Code Replacement Library Hierarchies

Using the programming interface, you can organize multiple code replacement libraries in a hierarchy. The following example shows a registration file that defines four code replacement tables organized in a hierarchy of four code replacement libraries. The tables include entries that increase in specificity: common entries, entries for TI devices, entries for TI C6xx devices, and entries specific to the TI C67x device.

```

function rtwTargetInfo(cm)

cm.registerTargetInfo(@locCrlRegFcn);

function thisCrl = locCrlRegFcn

% Register a code replacement library that includes common entries
thisCrl(1) = RTW.TflRegistry;
thisCrl(1).Name = 'Common Replacements';
thisCrl(1).Description = 'Common code replacement entries shared by other libraries';
thisCrl(1).TableList = {'crl_table_general'};
thisCrl(1).TargetHWDeviceType = {'*'};

% Register a code replacement library for TI devices
thisCrl(2) = RTW.TflRegistry;
thisCrl(2).Name = 'TI Device Replacements';
thisCrl(2).Description = 'Code replacement entries shared across TI devices';
thisCrl(2).TableList = {'crl_table_TI_devices'};
thisCrl(2).TargetHWDeviceType = {'TI C28x', 'TI C55x', 'TI C62x', 'TI C64x', 'TI 67x'};
thisCrl(1).BaseTfl = 'Common Replacements'

% Register a code replacement library for TI c6xx devices
thisCrl(3) = RTW.TflRegistry;
thisCrl(3).Name = 'TI c6xx Device Replacements';
thisCrl(3).Description = 'Code replacement entries shared across TI C6xx devices';
thisCrl(3).TableList = {'crl_table_TIC6xx_devices'};
thisCrl(3).TargetHWDeviceType = {'TI C62x', 'TI C64x', 'TI 67x'};

```

```
% Register a code replacement library for the TI c67x device
thisCrl(3) = RTW.TflRegistry;
thisCrl(3).Name = 'TI c67x Device Replacements';
thisCrl(3).Description = 'Code replacement entries for the TI C67x device';
thisCrl(3).TableList = {'crl_table_TIC67x_device'};
thisCrl(3).TargetHWDeviceType = {'TI 67x'};
```

Related Examples

- “Define Code Replacement Mappings”
- “Data Alignment for Code Replacement”
- “Specify Build Information for Replacement Code”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”
- “Deploy Code Replacement Library”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code Replacement Libraries”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Review and Test Code Replacements

In this section...

“Code Replacement Table Validation” on page 21-74

“Validate a Table Definition File” on page 21-74

“Review Table Content” on page 21-75

“Verify Library Registration and Content” on page 21-76

“Review and Trace Code Replacements Generated Using a Code Replacement Library” on page 21-77

“Determine Why Code Replacement Functions Are Not Used” on page 21-81

Code Replacement Table Validation

After you create or modify a code replacement table, use the following techniques to examine and validate the table and its entries.

- Invoke the table definition file
- Use the Code Replacement Viewer to examine libraries, tables, and entries
- Trace code replacements from the source for which you applied the code replacement library
- Examine code replacement cache hits and misses logged during code generation

Validate a Table Definition File

After you create or modify a code replacement table definition file, validate it. At the MATLAB command line, specify the name of the table in a call to the `isvalid` function. For example,

```
isvalid(crl_table_sinfcn)
```

```
ans =
```

```
1
```

MATLAB displays errors that it finds. In the following example, MATLAB detects a typo in a data type name.

```
isvalid(cr1_table_sinfcn)
```

```
??? RTW_CORE:tf1:Tf1Table: Unsupported data type, 'dooble'.
```

```
Error in ==> cr1_table_sinfcn at 7
```

```
hTable.registerCFUNCTIONEntry(100, 1, 'sin', 'dooble', 'sin_dbl', ...
```

Review Table Content

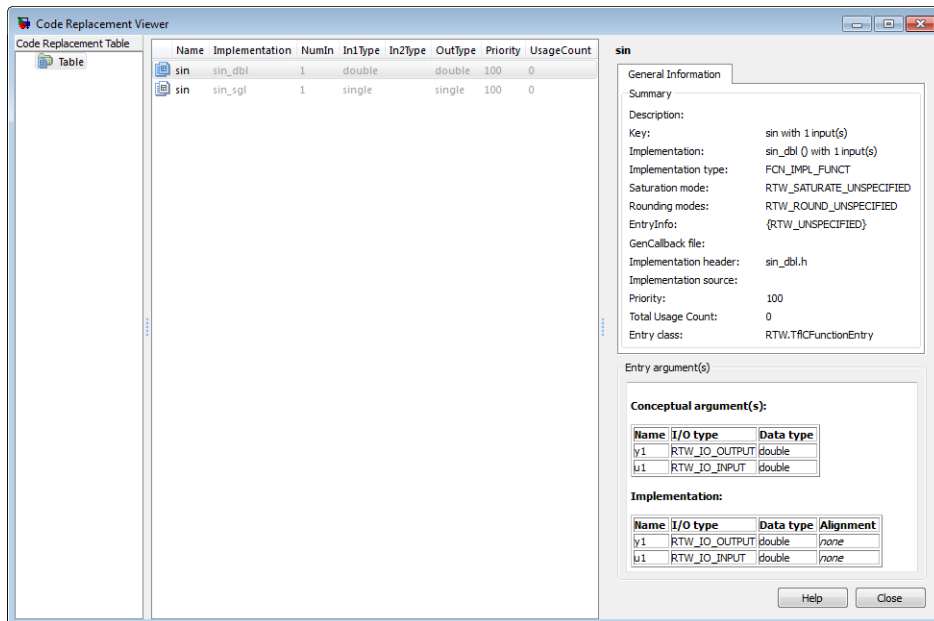
After you create or modify a code replacement table, use the Code Replacement Viewer to review the table and its entries.

- 1 Open the viewer. At the MATLAB command line, enter the following command, where *table* is the name of the code replacement table you want to review:

```
RTW.viewTf1(table)
```

For example,

```
RTW.viewTf1(cr1_table_sinfcn)
```



- 2 Select entries in the table and verify that the display of the contents meets your expectations. For example:

- 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.

Verify Library Registration and Content

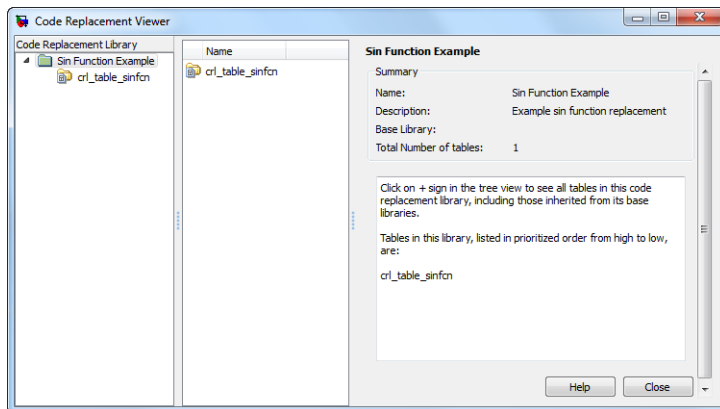
After you register a library that includes your code replacement table, use the Code Replacement Viewer to verify that the library is registered and examine the library and the tables it contains.

- 1 Open the viewer. At the MATLAB command line, enter the following command, where *library* is the name of the code replacement library you want to review:

```
RTW.viewTf1('library')
```

For example:

```
RTW.viewTf1('Sin Function Example')
```



If the tool does not display your library,

- The code replacement library registration file, `rtwTargetInfo.m`, might contain an error.
- You might need to refresh the library registration information. Initiate a refresh with the following command:

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

- 2 Select your library and examine and compare its tables. Common problems that you can detect include:
 - Relative order of tables in the library is not correct (by default, the tool displays tables in search order)
 - Table entry problems as listed in the previous section

Review and Trace Code Replacements Generated Using a Code Replacement Library

After you register a library that includes your code replacement tables, generate code and verify that the code generator replaces code as you expect. The following example illustrates two complementary approaches:

- Check the Code Replacement Report section of the code generation report for expected replacements
- Trace code replacements

For models that consist of model hierarchies, repeat the following procedure for each model in the hierarchy. Generate code for and review the trace information of each referenced model separately (stand alone). Logged cache hit and miss information captured in the Code Replacement Viewer is valid for the last model for which code was generated only. The code generator overwrites logged information as you generate code for each model in the hierarchy.

- 1 Open an ERT-based model for which you anticipate that a function or operator replacement should occur. This example uses the example model `rtwdemo_crladbsub`.
- 2 Configure the code generator to use your code replacement library. In the Configuration Parameters dialog box, on the **Code Generation > Interface** pane, set the **Code replacement library** parameter to the name of your library. For the example model, the library is set to `Addition & Subtraction Examples`.

- 3 Configure the code generation report. On the **Code Generation > Report** pane, select:
 - **Create code generation report**
 - **Open report automatically**
 - **Model-to-code**
 - **Summarize which blocks triggered code replacements**
- 4 Configure comments for the generated code. On the **Code Generation > Comments** pane, select:
 - **Include comments**
 - Either or both of **Simulink block / Stateflow object comments** and **Simulink block descriptions**

These options include Simulink block information in the **Code Replacements Report**.

- 5 Configure the code generator to generate code only. On the **Code Generation** pane, select **Generate code only**. You want to review your code replacements in the generated code before building an executable.
- 6 Click **Generate code** to generate code for the model.
- 7 Open the **Code Replacements Report** section of the code generation report. That report lists the replacement functions that the code generator used, and provides a mapping between each replacement instance and the Simulink block that triggered the replacement.

Code replacements in rtwdemo_crladddsub

Code replacements for library 'Addition & Subtraction Examples'. The library comprises:

- Addition & Subtraction Examples
 - crl_table_addsub

To see the replacements and misses in the Code Replacement Viewer, look [here](#).

1. Addition operator replacements [\[hide\]](#)

The following table provides a mapping from the addition operators used from the selected Code Replacement Library to the blocks in the model that triggered the replacement.

Function	Block
s16_add_s16_s16	<S1>:1
u8_add_u8_u8	<Root>/Add8
u16_add_u16_u16	<Root>/Add16

2. Subtraction operator replacements [\[hide\]](#)

The following table provides a mapping from the subtraction operators used from the selected Code Replacement Library to the blocks in the model that triggered the replacement.

Function	Block
u32_sub_u32_u32	<Root>/EmbSub32
s8_sub_s8_s8	<Root>/Sub8
s32_sub_s32_s32	<Root>/Sub32

Review the report.

- Check whether expected function and operator code replacements occurred.
 - In the replacements sections, click each block link to see the source that triggered the reported code replacement.
 - Click the link that opens the Code Replacement Viewer. In the viewer, review code replacement hits and misses. For more information, see “Determine Why Code Replacement Functions Are Not Used” on page 21-81.
- 8** In the Simulink model window, use model-to-code highlighting to trace code replacements. Identify and right-click a block for which you expected code replacement to occur. Then, select **C/C++ Code > Navigate to C/C++ Code**. The code generation report appears with the corresponding replacement code highlighted. In the example model rtwdemo_crladddsub, right-click the Add8 block and select **C/C++ Code > Navigate to C/C++ Code**.

```

25
26  /* Real-time model */
27  RT_MODEL rtM;
28  RT_MODEL *const rtM = &rtM;
29
30  /* Model step function */
31  void rtwdemo_crladdsub_step(void)
32  {
33      /* Output: '<Root>/Out1' incorporates:
34       * Inport: '<Root>/In1'
35       * Inport: '<Root>/In2'
36       * Sum: '<Root>/Add8'
37       */
38      rtY.Out1 = u8_add_u8_u8(rtU.In1, rtU.In2);
39
40      /* Output: '<Root>/Out2' incorporates:
41       * Inport: '<Root>/In3'
42       * Inport: '<Root>/In4'
43       * Sum: '<Root>/Add16'
44       */
45      rtY.Out2 = u16_add_u16_u16(rtU.In3, rtU.In4);
46

```

Inspect the generated code to see if the function or operator replacement occurred as expected.

If a function or operator is not replaced as expected, a call site request did not match one of your table entries. The code generator used a higher-priority (lower priority value) match or did not find a match.

To analyze the code replacement match and replacement behavior, use the following resources:

- Code Replacement Viewer, as described in “Review Table Content” on page 21-75 and “Verify Library Registration and Content” on page 21-76
- Code generation reports, with bidirectional tracing between models and generated code
- Information about code replacement library cache hits and misses logged during code generation, as described in “Determine Why Code Replacement Functions Are Not Used” on page 21-81

Determine Why Code Replacement Functions Are Not Used

- “Code Replacement Hits and Misses” on page 21-81
- “View Cache Hits and Misses Using Viewer Trace Information” on page 21-81
- “View Cache Hits and Misses Using Command-Line” on page 21-84

Code Replacement Hits and Misses

Code replacement can behave differently than you expect. To verify code replacements, first inspect the generated code, as described in “Review and Trace Code Replacements Generated Using a Code Replacement Library” on page 21-77.

To analyze replacement behavior, in addition to examining the Code Replacements Report and examining your code replacement tables in the Code Replacement Viewer, you can view the library cache hits and misses logged the last time code was generated. This approach provides information on how to configure your table entry to get the desired replacement. The following techniques can help you determine why the code generator does not use code replacement functions:

- “View Cache Hits and Misses Using Viewer Trace Information” on page 21-81
- “View Cache Hits and Misses Using Command-Line” on page 21-84

For models that consist of model hierarchies, you must generate code for and review the trace information of each referenced model separately (stand alone). Logged cache hit and miss information captured in the Code Replacement Viewer is valid for the last model for which code was generated only. The code generator overwrites logged information as you generate code for each model in the hierarchy.

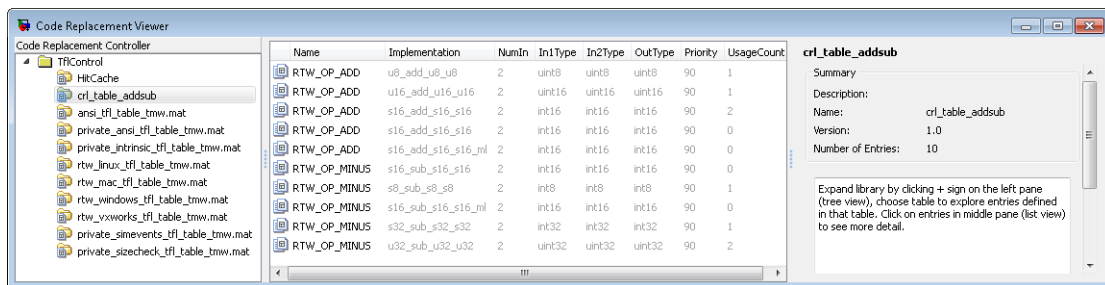
View Cache Hits and Misses Using Viewer Trace Information

When debugging a code replacement table entry, you can use hit and miss information in the Code Replacement Viewer to help determine why the code generator did not use a replacement function in the generated code.

The following example shows how to display the cache hits and misses logged for a code replacement table entry applied during code generation of example model `rtwdemo_crladdsub`.

- 1 Open an ERT-based model for which you anticipate that a function or operator replacement should occur. This example uses the example model `rtwdemo_crladdsub`.

- 2 Configure the code generator to use your code replacement library. In the Configuration Parameters dialog box, on the **Code Generation > Interface** pane, set the **Code replacement library** parameter to the name of your library. For the example model, the library is set to **Addition & Subtraction Examples**.
- 3 Configure the code generation report. On the **Code Generation > Report** pane, select:
 - **Create code generation report**
 - **Open report automatically**
 - **Summarize which blocks triggered code replacements**
- 4 Generate code. A complete rebuild of a single standalone model is required to display cache hits and misses.
- 5 Open the Code Replacements Report section of the code generation report. Click the link for opening the Code Replacement Viewer.
- 6 In the left pane of the viewer, select the code replacement table you want to examine. In the middle pane, the viewer lists information for entries in the table, including a usage count, which is the number of times the table entry was matched and its replacement function used during code generation. The following display shows entries for code replacement table `cr1_table_addsub`.



- 7 In the middle pane, select the table entry to examine. In the right pane, the viewer displays **General Information** and **Trace Information** about the table entry.

The screenshot shows the RTW tool interface. On the left is a table of operations, and on the right is a detailed view of the selected entry, RTW_OP_ADD.

Name	Implementation	NumIn	In1Type	In2Type	OutType	Priority	UsageCount
RTW_OP_ADD	u8_add_u8_u8	2	uint8	uint8	uint8	90	1
RTW_OP_ADD	u16_add_u16_u16	2	uint16	uint16	uint16	90	1
RTW_OP_ADD	s16_add_s16_s16	2	int16	int16	int16	90	2
RTW_OP_ADD	s16_add_s16_s16_ml	2	int16	int16	int16	90	0
RTW_OP_ADD	s16_add_s16_s16_ml	2	int16	int16	int16	90	0
RTW_OP_MINUS	s16_sub_s16_s16	2	int16	int16	int16	90	0
RTW_OP_MINUS	s8_sub_s8_s8	2	int8	int8	int8	90	1
RTW_OP_MINUS	s16_sub_s16_s16_ml	2	int16	int16	int16	90	0
RTW_OP_MINUS	s32_sub_s32_s32	2	int32	int32	int32	90	1
RTW_OP_MINUS	u32_sub_u32_u32	2	uint32	uint32	uint32	90	2

The right pane shows the details for RTW_OP_ADD:

- General Information:**
 - Description: RTW_OP_ADD with 2 input(s)
 - Key: s16_add_s16_s16 () with 2 input(s)
 - Implementation: FCN_IMPL_FUNCT
 - Saturation mode: RTW_WRAP_ON_OVERFLOW
 - Rounding modes: RTW_ROUND_SIMPLEST
 - GenCallback file: s16_add_s16_s16.h
 - Implementation header: s16_add_s16_s16.h
 - Implementation source: s16_add_s16_s16.c
 - Priority: 90
 - Total Usage Count: 2
 - Entry class: RTW.TFICOperationEntry
- Trace Information:**
 - Entry argument(s):

Name	I/O type	Data type
y1	RTW_IO_OUTPUT	int16
u1	RTW_IO_INPUT	int16
u2	RTW_IO_INPUT	int16
 - Implementation:

Name	I/O type	Data type	Alignment
y1	RTW_IO_OUTPUT	int16	none
u1	RTW_IO_INPUT	int16	none
u2	RTW_IO_INPUT	int16	none

- 8 In the right pane, select the **Trace Information** tab. The following example shows trace information for the `int16` addition table entry.

The screenshot shows the RTW tool interface. On the left is a table of operations, and on the right is a detailed view of the selected entry, RTW_OP_ADD, with the Trace Information tab selected.

Name	Implementation	NumIn	In1Type	In2Type	OutType	Priority	UsageCount
RTW_OP_ADD	u8_add_u8_u8	2	uint8	uint8	uint8	90	1
RTW_OP_ADD	u16_add_u16_u16	2	uint16	uint16	uint16	90	1
RTW_OP_ADD	s16_add_s16_s16	2	int16	int16	int16	90	2
RTW_OP_ADD	s16_add_s16_s16	2	int16	int16	int16	90	0
RTW_OP_ADD	s16_add_s16_s16...	2	int16	int16	int16	90	0
RTW_OP_MINUS	s16_sub_s16_s16	2	int16	int16	int16	90	0
RTW_OP_MINUS	s8_sub_s8_s8	2	int8	int8	int8	90	1
RTW_OP_MINUS	s16_sub_s16_s16...	2	int16	int16	int16	90	0
RTW_OP_MINUS	s32_sub_s32_s32	2	int32	int32	int32	90	1
RTW_OP_MINUS	u32_sub_u32_u32	2	uint32	uint32	uint32	90	2

The right pane shows the details for RTW_OP_ADD:

- Trace Information:**
 - Hit Source Locations:

Source Location
rtvdemo_criaddsub/ChartAdd16:1
rtvdemo_criaddsub/ChartAdd16:1
 - Miss Source Locations:

Source Location	Miss Reason
rtvdemo_criaddsub/Add16	Mismatched rounding mode: actual RTW_ROUND_SIMPLEST, expected RTW_ROUND_CEILING.
rtvdemo_criaddsub/Add8	Mismatched saturation mode: actual RTW_WRAP_ON_OVERFLOW, expected RTW_SATURATE_ON_OVERFLOW.

The **Trace Information** tab lists **Hit Source Locations** and **Miss Source Locations**. The usage count for the entry is 2. The **Trace Information** tab lists 2 hits and 2 misses. The display provides links to each source location (the source for which code replacement was considered) and, for misses, lists a **Miss Reason**. In the display above, in one case a saturation mode setting did not match between the table entry and the source block, and in another case the signedness attribute did

not match. If an expected replacement did not occur, you can use the hit and miss information to modify the match criteria in the code replacement table entry.

View Cache Hits and Misses Using Command-Line

To display the code replacement cache hits and misses logged during the most recent code generation session in the command window, use the following command:

```
>> cr1=get_param('model', 'TargetFcnLibHandle')
```

The resulting display includes the following fields:

Field	Description
HitCache	Table containing code replacement entries that were matched during a code generation session. These entries represent function implementations that should appear in the generated code.
MissCache	Table containing code replacement entries that failed to match during a code generation session. These entries are created by the code generation process for the purpose of querying the library to locate a registered implementation.

In the following example, the most recent code generation session logged one cache hit and zero cache misses. You can examine the logged `HitCache` entry using its table index.

```
a=get_param('my_sin_func', 'TargetFcnLibHandle')
a =
RTW.Tf1Control
    Version: '1.0'
    HitCache: [2x1 RTW.Tf1CFunctionEntry]
    MissCache: []
    TLCCallList: [0x1 handle]
    Tf1Tables: [10x1 RTW.Tf1Table]
    TargetHWWordLengths: '8,16,32,32'
    SupportNonFinite: 1
    TLCSupported: 1
    Recording: 0
    LoadedLibrary: 'Sin Function Example'
    TargetCharacteristics: [1x1 RTW.TargetCharacteristics]
    CurrentHWDevice: 'Specified'

a.HitCache(1)
ans =
```

Tf1CFunctionEntry with properties:

```

    Implementation: [1x1 RTW.CImplementation]
    SlopesMustBeTheSame: 0
    BiasMustBeTheSame: 0
    AlgorithmParams: []
        ImplType: 'FCN_IMPL_MACRO'
    AdditionalHeaderFiles: {0x1 cell}
    AdditionalSourceFiles: {0x1 cell}
        OtherFiles: {0x1 cell}
    AdditionalIncludePaths: {0x1 cell}
    AdditionalSourcePaths: {0x1 cell}
        AdditionalLinkObjs: {0x1 cell}
    AdditionalLinkObjsPaths: {0x1 cell}
        AdditionalLinkFlags: {0x1 cell}
    AdditionalCompileFlags: {0x1 cell}
        SearchPaths: {0x1 cell}
    Key: 'NULL'
    Priority: 100
    ConceptualArgs: [1x1 RTW.Tf1ArgNumeric]
    EntryInfo: []
    GenCallback: ''
    GenFileName: ''
    SaturationMode: 'RTW_SATURATE_UNSPECIFIED'
    RoundingModes: {'RTW_ROUND_UNSPECIFIED'}
    TypeConversionMode: 'RTW_EXPLICIT_CONVERSION'
    AcceptExprInput: 0
    SideEffects: 0
    UsageCount: 3
    RecordedUsageCount: 0
    Description: ''
    StoreFcnReturnInLocalVar: 0
    TraceManager: [1x1 RTW.Tf1TraceManager]

```

Related Examples

- “Choose a Code Replacement Library”
- “Replace Code Generated from Simulink Models”
- “Generate a Code Generation Report”
- “Traceability in Code Generation Report”
- “Choose a SIL or PIL Approach”

- “Program Builds”
- “Build and Run a Program”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “Reports for Code Generation”
- “About Code Tracing”
- “About SIL and PIL Simulations”
- “Code Replacement Libraries”
- “Code Replacement Terminology”
- “Code Replacement Customization Limitations”

External Web Sites

- “Replacing Math Functions and Operators”

Deploy Code Replacement Library

When you are ready to package and deploy a custom code replacement library for others to use,

- 1 Move your code replacement table files to an area that is on the MATLAB search path and that is accessible to and shared by other users.
- 2 Move the `rtwTargetInfo.m` registration file, to an area that is on the MATLAB search path and that is accessible to and shared by other users. If you are deploying a library to a folder in a development environment that already contains a `rtwTargetInfo.m` file, copy the registration code from your code replacement library version of `rtwTargetInfo.m` and paste it into the shared version of that file.
- 3 Register the library customizations or restart MATLAB.
- 4 Verify that the libraries are available for configuring the code generator and that code replacements occur as expected.
- 5 Inform users that the libraries are available and provide direction on when and how to apply them.

Related Examples

- “Relocate Code to Another Development Environment”
- “Review and Test Code Replacements”
- “Develop a Code Replacement Library”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code Replacement Libraries”
- “Code Replacement Terminology”

Math Function Code Replacement

This example shows how to define a code replacement mapping for a math function. The example defines a mapping for the `sin` function programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

- 1 Define a function for the definition.

```
function hTable = crl_table_sinfcn2()
%CRL_TABLE_SINFCN2 - Define function entry for code replacement table.
```

- 2 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 3 Create an entry for the function mapping with a call to the `RTW.Tf1CFunctionEntry` function.

```
% Create entry for sin function replacement
fcn_entry = RTW.Tf1CFunctionEntry;
```

- 4 Set function entry parameters with a call to the `setTf1CFunctionEntryParameters` function.

```
setTf1CFunctionEntryParameters(fcn_entry, ...
                               'Key',          'sin', ...
                               'Priority',      30, ...
                               'ImplementationName', 'mySin', ...
                               'ImplementationHeaderFile', 'basicMath.h',...
                               'ImplementationSourceFile', 'basicMath.c');
```

- 5 Create conceptual arguments `y1` and `u1`. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call.

```
createAndAddConceptualArg(fcn_entry, 'RTW.Tf1ArgNumeric', ...
                          'Name',     'y1', ...
                          'IOType',   'RTW_IO_OUTPUT', ...
                          'DataTypeMode', 'double');

createAndAddConceptualArg(fcn_entry, 'RTW.Tf1ArgNumeric', ...
                          'Name',     'u1', ...
                          'IOType',   'RTW_IO_INPUT', ...
                          'DataTypeMode', 'double');
```

- 6 Copy the conceptual arguments to the implementation arguments. This example uses a call to the `copyConceptualArgsToImplementation` function to create and add implementation arguments to the entry by copying matching conceptual arguments.

```
copyConceptualArgsToImplementation(fcn_entry);
```

- 7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, fcn_entry);
```

Related Examples

- “Define Code Replacement Mappings”
- “Data Alignment for Code Replacement”
- “Algorithm-Based Code Replacement”
- “Reserved Identifiers and Code Replacement ”
- “Customize Matching and Replacement Process for Functions”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “Code You Can Replace From Simulink Models”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Memory Function Code Replacement

This example shows how to define a code replacement mapping for a memory function. The example defines a mapping for the `memcpy` function programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

- 1 Define a function for the definition.

```
function hTable = crl_table_memcpy()
```

- 2 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 3 Create an entry for the function mapping with a call to the `RTW.Tf1CFunctionEntry` function.

```
% Create entry for void* memcpy(void*, void*, size_t)
fcn_entry = RTW.Tf1CFunctionEntry;
```

- 4 Set function entry parameters with a call to the `setTf1CFunctionEntryParameters` function.

```
% Set SideEffects to 'true' for function returning void to prevent it from
% being optimized away.
setTf1CFunctionEntryParameters(fcn_entry, ...
    'Key', 'memcpy', ...
    'Priority', 90, ...
    'ImplementationName', 'memcpy_int', ...
    'ImplementationHeaderFile', 'memcpy_int.h', ...
    'SideEffects', true);
```

- 5 Create conceptual arguments `y1`, `u1`, `u2`, and `u3`. This example uses calls to the `getTf1ArgFromString` and `addConceptualArg` functions to create and add the arguments.

```
arg = getTf1ArgFromString(hTable, 'y1', 'void*');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(fcn_entry, arg);
```

```
arg = getTf1ArgFromString(hTable, 'u1', 'void*');
addConceptualArg(fcn_entry, arg);
```

```
arg = getTf1ArgFromString(hTable, 'u2', 'void*');
addConceptualArg(fcn_entry, arg);
```

```
arg = getTf1ArgFromString(hTable, 'u3', 'size_t');
addConceptualArg(fcn_entry, arg);
```

- 6 Copy the conceptual arguments to the implementation arguments. This example uses a call to the `copyConceptualArgsToImplementation` function to create

and add implementation arguments to the entry by copying matching conceptual arguments.

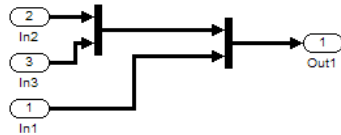
```
copyConceptualArgsToImplementation(fcn_entry);
```

- 7 Add the entry to a code replacement table with a call to the `addEntry` function.

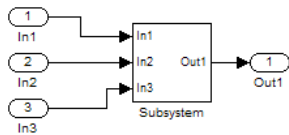
```
addEntry(hTable, fcn_entry);
```

To test this example, create model that uses the `memcpy` function for vector assignments. For example:

- 1 Use In, Out, and Mux blocks to create the following model. (Alternatively, open the example model `rtwdemo_crlmath` and copy the contents of `Subsystem1` to a new model.)



- 2 Select the diagram and use **Edit > Subsystem** to make it a subsystem.



- 3 Configure the subsystem with the following settings:
 - On the **Solver** pane, select a fixed-step solver.
 - On the **Optimization > Signals and Parameters** pane, select **Use memcpy for vector assignment** and set **Memcpy threshold (bytes)** to 64.
 - On the **Code Generation** pane, select an ERT-based system target file.
 - On the **Code Generation > Interface** pane, select the code replacement library that contains your memory function entry.
- 4 In the Model Explorer, configure the **Signal Attributes** for the `In1`, `In2`, and `In3` source blocks. For each, set **Port dimensions** to `[1, 100]`, and set **Data type** to `int32`. Apply the changes. Save the model.
- 5 Generate code and a code generation report.
- 6 Review the code replacements.

Related Examples

- “Define Code Replacement Mappings”
- “Review and Test Code Replacements”
- “Data Alignment for Code Replacement”
- “Reserved Identifiers and Code Replacement ”
- “Customize Matching and Replacement Process for Functions”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “Code You Can Replace From Simulink Models”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Nonfinite Function Code Replacement

This example shows how to define a code replacement mapping for nonfinite utility functions.

- 1 Define a function for the definition.

```
function hTable = crl_table_nonfinite()
```

- 2 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 3 Create entries for the function mappings. To minimize the size of this function, the example uses a local function, `locAddFcnEnt`, to group lines of code repeated for each entry. A call to the `RTW.Tf1CFunctionEntry` function creates an entry for the collection of local function entry definitions.

```

%% Create entries for nonfinite utility functions
% locAddFcnEnt(hTable, key, implName, out, in1, hdr)

locAddFcnEnt(hTable, 'getNaN', 'getNaN', 'double', 'void', 'nonfin.h');
locAddFcnEnt(hTable, 'getNaNF', 'getNaNF', 'single', 'void', 'nonfin.h');
locAddFcnEnt(hTable, 'getInf', 'getInf', 'double', 'void', 'nonfin.h');
locAddFcnEnt(hTable, 'getInfF', 'getInfF', 'single', 'void', 'nonfin.h');
locAddFcnEnt(hTable, 'getMinusInf', 'getMinusInf', 'double', 'void', 'nonfin.h');
locAddFcnEnt(hTable, 'getMinusInfF', 'getMinusInfF', 'single', 'void', 'nonfin.h');

%% Local Function
function locAddFcnEnt(hTable, key, implName, out, in1, hdr)
    if isempty(hTable)
        return;
    end

    fcn_entry = RTW.Tf1CFunctionEntry;
```

- 4 Set function entry parameters with a call to the `setTf1CFunctionEntryParameters` function.

```

setTf1CFunctionEntryParameters(fcn_entry, ...
    'Key', key, ...
    'Priority', 90, ...
    'ImplementationName', implName, ...
    'ImplementationHeaderFile', hdr);
```

- 5 Create conceptual arguments `y1` and `u1`. This example uses calls to the `getTf1ArgFromString` and `addConceptualArg` functions to create and add the arguments.

```

arg = getTf1ArgFromString(hTable, 'y1', out);
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(fcn_entry, arg);
```

```
arg = getTf1ArgFromString(hTable, 'u1', in1);
addConceptualArg(fcn_entry, arg);
```

- 6 Copy the conceptual arguments to the implementation arguments. This example uses a call to the `copyConceptualArgsToImplementation` function to create and add implementation arguments to the entry by copying matching conceptual arguments.

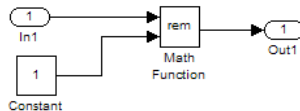
```
copyConceptualArgsToImplementation(fcn_entry);
```

- 7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, fcn_entry);
```

To test this example, create a model that uses a nonfinite function. For example:

- 1 Create a model that includes a Math Function block that is set to the `rem` function. For example:



- 2 Configure the model with the following settings:
 - On the **Solver** pane, select a fixed-step solver.
 - On the **Code Generation** pane, select an ERT-based system target file.
 - On the **Code Generation > Interface** pane, select the code replacement library that contains your memory function entry and select **Support: non-finite numbers**.
- 3 In the Model Explorer, configure the **Signal Attributes** for the In1 and Constant source blocks. For each source block, set **Data type** to `double`. Apply the changes. Save the model.
- 4 Generate code and a code generation report.
- 5 Review the code replacements.

Related Examples

- “Define Code Replacement Mappings”
- “Review and Test Code Replacements”
- “Data Alignment for Code Replacement”

- “Reserved Identifiers and Code Replacement ”
- “Customize Matching and Replacement Process for Functions”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “Code You Can Replace From Simulink Models”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Semaphore and Mutex Function Replacement

If you are authoring or maintaining a custom target that supports concurrent execution, and you want to provide custom implementations of semaphore or mutex operations, you create a code replacement table. The table must have four semaphore entries, four mutex entries, or both, and include the table in a custom code replacement library. (The semaphore or mutex entries are mutually dependent. You must provide them in complete sets of four.)

Note: A custom target that supports concurrent multitasking must set the target configuration parameter `ConcurrentExecutionCompliant`. For more information, see “Support Concurrent Execution of Multiple Tasks” in the Simulink Coder documentation.

During code generation for a multicore target environment, if the build process generates semaphore or mutex function calls for data transfer between tasks, use a custom library to replace the generated function calls with custom semaphore or mutex implementations that are optimal for your target environment. Using the Code Replacement Tool (`crtool`) or equivalent code replacement functions, you can:

- Configure code replacement table entries for custom semaphore or mutex functions. During system startup, execution of the code for data transfer between tasks, and system shutdown the generated code calls these functions.
- Configure DWork arguments that represent global data, which the semaphore or mutex functions access. A DWork pointer is passed to the model entry functions.

Generated mutex and semaphore code typically consists of these elements:

Code	Generated Code
Model initialization	Initialization function call that creates a mutex or semaphore function to control entry to a critical section of code.
Model step	<ul style="list-style-type: none"> • Before code for a data transfer between tasks enters the critical section, mutex lock or semaphore wait function calls reserve the critical section of code. • After code for a data transfer between tasks finishes executing the critical section, mutex unlock or

Code	Generated Code
	semaphore post function calls release the critical section of code.
Model termination	Optional destroy function call to explicitly delete the mutex or semaphore.

This example shows how to create code replacement table entries for a mutex replacement scenario. You configure a multicore target model for concurrent execution and for data transfer between tasks of differing rates, which are handled by Rate Transition blocks. In the generated code for the model, each Rate Transition block has a separate, unique mutex. Mutex lock and unlock operations within the Rate Transition block generated code share access to the same global data by using the unique mutex created for that Rate Transition block.

- 1 Open the Code Replacement Tool.
- 2 Create and open a new table.
- 3 Name the table `cr1_table_rt_mutex`.
- 4 Create an entry for a mutex initialization function replacement.
 - a Select **File > New entry > Semaphore entry** to open a new table entry for configuring a semaphore or mutex replacement.
 - b In the **Mapping Information** tab, use the **Function** parameter to select `Mutex Init`. Initial default values for the table entry appear. In the **Conceptual function** section, typically you can leave the argument settings at their defaults.
 - c In the **DWork attributes** section, the **Allocate DWork** option is selected. The dialog box provides a unique entry tag for the DWork argument `d1`.

The screenshot shows the 'DWork attributes' configuration pane. It contains the following elements:

- DWork attributes**: A section header.
- Allocate DWork**: A checked checkbox.
- DWork argument**: A section header.
- Entry tag (unique): : A text input field containing 'entry_25576'.
- DWork arguments**: A list box containing the argument 'd1'.
- Argument properties**: A section header.
- Data type: : A text input field containing 'void'.
- Pointer**: A checked checkbox.

On the **DWork attributes** pane, configure a DWork argument to the replacement function. The DWork argument supports sharing of a semaphore or mutex between code that creates the semaphore or mutex, code that requests and relinquishes access, and code that deletes the semaphore or mutex. In this example, the DWork argument for the `Mutex Init` function defines a unique entry tag, `entry_25576`. That function also defines DWork arguments for `Mutex Lock`, `Mutex Unlock`, and `Mutex Destroy`, which reference the entry tag to share the DWork data.

The only data type supported for the DWork **Data type** parameter is `void*`.

- d** In the **Replacement function** section, enter a function name in the **Name** field. This example uses `myMutexCreate`. In the list of **Function arguments**, leave the DWork argument `d1` data type as `void**`.

Replacement function

Function prototype

Name: C++ namespace:

Function returns void

Function arguments

y1(return arg)
d1

Argument properties

Data type: I/O type:

Const Pointer Pointer-Pointer

Function signature preview

```
void myMutexCreate (void** d1);
```

The C function signature preview is:

```
void myMutexCreate (void** d1);
```

- e In the **Replacement function** section, select **Function modifies internal or global state**. This option instructs the code generator not to optimize away the implementation function described by this entry because it accesses global memory values. Click **Apply**. Optionally, you can click **Validate entry** to validate the information entered in the **Mapping Information** tab.
- To create a sample table entry, configure the replacement function signature without the replacement function and its build information. If header and source files for these functions are available, select the **Build Information** table to specify them.
- f The `Mutex Init` table entry is complete. Optionally, you can save the table to a file, and inspect the MATLAB code created for the table definition so far.
- 5 Repeat the following sequence to create the table entries for the mutex lock, unlock, and destroy function replacements. Each table entry references the DWork unique tag entry, `entry_25576`, defined in the `Mutex Init` table entry.
- a Select **File > New entry > Semaphore entry**.

- b In the **Mapping Information** tab, use the **Function** parameter to select **Mutex Lock**, **Mutex Unlock**, or **Mutex Destroy**. Initial default values for the table entry appear. In the **Conceptual function** section, typically you can leave the argument settings at their defaults.
- c For a Rate Transition block mutex, the wait, post, and destroy functions operate on the DWork allocated at system startup by the mutex initialization function. In the **DWork attributes** section, verify that the **Allocate DWork** option is cleared. From the **DWork Allocator entry** drop-down list, select the entry tag matching the value in the **Mutex Init** table entry. In this example, the entry tag is `entry_25576`.

DWork attributes

Allocate DWork

DWork Allocator entry: entry_25576

- d In the **Replacement function** section, **Name** field, enter a function name. This example uses `myMutexLock`, `myMutexUnlock`, and `myMutexDelete`. In the list of **Function arguments**, leave the DWork argument `d1` data type as `void*`.

Replacement function

Function prototype

Name: C++ namespace:

Function returns void

Function arguments

y1(return arg)
d1

Argument properties

Data type: void I/O type: INPUT

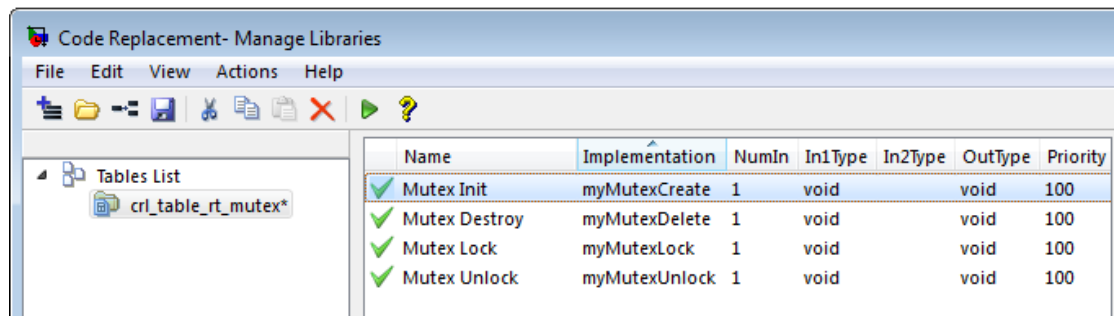
Const
 Pointer
 Pointer-Pointer

Alignment value: -1

Function signature preview

```
void myMutexLock ( void* d1);
```

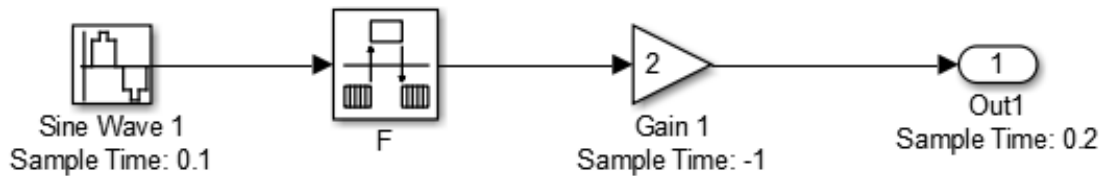
- e In the **Implementation attributes** section, select the option **Function modifies internal or global state**. This option instructs the code generator not to optimize away the implementation function described by this entry because it accesses global memory values.
 - f Optionally, supply replacement function build information on the **Build Information** tab.
 - g Click **Apply**. In the middle pane, right-click the table entry and select **Validate entry(s)**.
- 6 When you have added the table entries for **Mutex Lock**, **Mutex Unlock**, and **Mutex Destroy** to the entry for **Mutex Init**, the rate transition mutex replacement table is complete. In the leftmost pane, right-click the table name and select **Validate table**. Address errors and repeat the table validation.



- 7 Save the table to a MATLAB file in your working folder, for example, using **File > Save table**. The name of the saved file is the table name, `crl_table_rt_mutex`, with an `.m` extension. Optionally, you can open the saved file and examine the MATLAB code for the code replacement table definition.

To test this example, create a model that contains a rate transition for which the build process generates mutex function calls. Configure the model for a multicore target environment. For example:

- 1 Create this model.



2 Configure the model with the following settings:

- On the **Solver** pane, select a fixed-step solver.
- On the **Code Generation** pane, select an ERT-based system target file.
- On the **Code Generation > Interface** pane, select the code replacement library that contains your mutex entry.

Related Examples

- “Define Code Replacement Mappings”
- “Review and Test Code Replacements”
- “Data Alignment for Code Replacement”
- “Reserved Identifiers and Code Replacement ”
- “Customize Matching and Replacement Process for Functions”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “Code You Can Replace From Simulink Models”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Algorithm-Based Code Replacement

For some math function blocks, you can control code replacement based on the computation or approximation algorithm configured for that block. For example, you can configure:

- The Reciprocal Sqrt block to use the **Newton-Raphson** or **Exact** computation method.
- The Trigonometric Function block, with **Function** set to **sin**, **cos**, or **sincos**, to use the approximation method **CORDIC** or **None**.

You can define code replacement entries to replace these functions for one or all of the available computation methods. For example, you can define an entry to replace only Newton-Raphson instances of the `rSqrt` function.

To set the algorithm for a function in an entry definition, use the `EntryInfoAlgorithm` property in a call to the function `setTf1CFunctionEntryParameters`. The following table lists arguments for specifying the computation method to match during code generation.

Function	Argument
<code>rSqrt</code>	<ul style="list-style-type: none"> • 'RTW_DEFAULT' (match the default computation method, Exact) • 'RTW_NEWTON_RAPHSON' • 'RTW_UNSPECIFIED' (match any computation method)
<code>sin</code> <code>cos</code> <code>sincos</code>	<ul style="list-style-type: none"> • 'RTW_CORDIC' • 'RTW_DEFAULT' (match the default approximation method, None) • 'RTW_UNSPECIFIED' (match any approximation method)

For example, to replace only Newton-Raphson instances of the `rSqrt` function, create an entry as follows:

- 1 Define a function for the definition.

```
function hTable = crl_rsqr()
%CRL_TABLE_RSQRT - Define function entry for code replacement table.
```

- 2 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 3 Create an entry for the function mapping with a call to the `RTW.Tf1CFunctionEntry` function.

```
% Create entry for rsqrt function replacement
fcn_entry = RTW.Tf1CFunctionEntry;
```

- 4 Set function entry parameters with a call to the `setTf1CFunctionEntryParameters` function.

```
setTf1CFunctionEntryParameters(fcn_entry, ...
    'Key',                'rSqrt', ...
    'Priority',           80, ...
    'ImplementationName', 'rsqrt_newton', ...
    'ImplementationHeaderFile', 'rsqrt.h', ...
    'EntryInfoAlgorithm', 'RTW_NEWTON_RAPHSON');
```

- 5 Create conceptual arguments `y1` and `u1`. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call.

```
createAndAddConceptualArg(fcn_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'y1', ...
    'IOType',        'RTW_IO_OUTPUT', ...
    'DataTypeMode', 'double');

createAndAddConceptualArg(e, 'RTW.Tf1ArgNumeric', ...
    'Name',          'u1', ...
    'DataTypeMode', 'double');
```

- 6 Copy the conceptual arguments to the implementation arguments. This example uses a call to the `copyConceptualArgsToImplementation` function to create and add implementation arguments to the entry by copying matching conceptual arguments.

```
copyConceptualArgsToImplementation(fcn_entry);
```

- 7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, fcn_entry);
```

The generated code for a Newton-Raphson instance of the `rSqrt` function looks like the following code:

```
/* Model step function */
void mrsqrt_step(void)
{
```

```
/* Outport: '<Root>/Out1' incorporates:  
 * Inport: '<Root>/In1'  
 * Sqrt:   '<Root>/rSqrtBlk'  
 */  
mrsqrt_Y.Out1 = rsqrt_newton(mrsqrt_U.In1);  
}
```

Related Examples

- “Math Function Code Replacement”
- “Define Code Replacement Mappings”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “Code You Can Replace From Simulink Models”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Lookup Table Function Code Replacement

In this section...

“Lookup Table Algorithm Replacement” on page 21-106

“Lookup Table and Interpolation Algorithm Parameters” on page 21-106

“Lookup Table Function Signatures” on page 21-107

“Interactive Mapping with Code Replacement Tool” on page 21-111

“Programmatic Specification” on page 21-116

“Sample Code Replacement Definition for the lookup2D Function” on page 21-121

Lookup Table Algorithm Replacement

You can configure the algorithm for table lookup operations and index searches. Use the **Algorithm** tab of lookup table blocks. For example, you can specify the interpolation, extrapolation, and index search methods.

If the code generated for available algorithm options does not meet requirements for your application, you can create custom code replacement table entries to replace generated algorithm code. You can create the table entries programmatically or interactively by using the Code Replacement Tool.

For more information about using lookup table blocks, see “Nonlinearity” in the Simulink documentation.

Lookup Table and Interpolation Algorithm Parameters

Code Replacement Tool Prompt	Programmatic Name	Values*
Extrapolation method	ExtrapMethod	Clip Linear
Index search method	IndexSearchMethod	Evenly spaced points Linear search Binary search
Integer rounding mode	RndMeth	Ceiling Convergent

Code Replacement Tool Prompt	Programmatic Name	Values*
		Floor Nearest Round Simplest Zero
Interpolation method	InterpMethod	Flat Linear Cubic spline Nearest
Number of table dimensions	NumberOfTableDimensions	1 2 3 4 5
Remove protection against out-of-range input	RemoveProtectionInput	off on
Saturate on integer overflow	SaturateOnIntegerOverflow	off on
Use last breakpoint for input at or above upper limit	UseLastBreakpoint	off on
Use last table value for inputs at or above last breakpoint	UseLastTableValue	off on
Valid index input may reach last index	ValidIndexMayReachLast	off on

* When specifying parameter values programmatically, specify them as strings.

Lookup Table Function Signatures

To create code replacement table entries for a function corresponding to a lookup table algorithm, you must have information about the conceptual function signature and

relevant algorithm parameters. The following table provides the conceptual function signature information.

Conceptual Function Signature	Argument Summary
y1 = interp1D(u1, u2, u3, u4)	y1 – output u1 – index u2 – fraction u3 – table data u4 – table dimension length
y1 = interp2D(u1, u2, u3, u4, u5, u6, u7)	y1 – output u1, u3 – index u2, u4 – fraction u5 – table data u6, u7 – table dimension lengths
y1 = interp3D(u1, u2, u3, u4, u5, u6, u7, u8, u9, u10)	y1 – output u1, u3, u5 – index u2, u4, u6 – fraction u7 – table data u8, u9, u10 – table dimension lengths
y1 = interp4D(u1, u2, u3, u4, u5, u6, u7, u8, u9, u10, u11, u12, u13)	y1 – output u1, u3, u5, u7 – index u2, u4, u6, u8 – fraction u9 – table data u10, u11, u12, u13 – table dimension lengths
y1 = interp5D(u1, u2, u3, u4, u5, u6, u7, u8, u9, u10, u11, u12, u13, u14, u15, u16)	y1 – output u1, u3, u5, u7, u9 – index u2, u4, u6, u8, u10 – fraction u11 – table data u12, u13, u14, u15, u16 – table dimension lengths
y1 = interpND({ui, uf,... ut, un...})	y1 – output ui, uf is an index and fraction pair per dimension ut – table data un – table dimension lengths

Conceptual Function Signature	Argument Summary
y1 = lookup1D(u1, u2, u3, u4)	y1 – output u1 – input u2 – breakpoint data u3 – table data u4 – table dimension length
y1 = lookup2D(u1, u2, u3, u4, u5, u6, u7)	y1 – output u1, u2 – input u3, u4 – breakpoint data u5 – table data u6, u7 – table dimension lengths
y1 = lookup3D(u1, u2, u3, u4, u5, u6, u7, u8, u9, u10)	y1 – output u1, u2, u3 – input u4, u5, u6 – breakpoint data u7 – table data u8, u9, u10 – table dimension lengths
y1 = lookup4D(u1, u2, u3, u4, u5, u6, u7, u8, u9, u10, u11, u12, u13)	y1 – output u1, u2, u3, u4 – input u5, u6, u7, u8 – breakpoint data u9 – table data u10, u11, u12, u13 – table dimension lengths
y1 = lookup5D(u1, u2, u3, u4, u5, u6, u7, u8, u9, u10, u11, u12, u13, u14, u15, u16)	y1 – output u1, u2, u3, u4, u5 – input u6, u7, u8, u9, u10 – breakpoint data u11 – table data u12, u13, u14, u15, u16 – table dimension lengths
y1 = lookupND(un,..., ub,..., ut, un...)	y1 – output un, input per dimension ub, breakpoint per dimension ut – table data un – table dimension lengths

Conceptual Function Signature	Argument Summary
<code>y1 = lookupND_Direct(u1, u2,...ui, ui+1)</code>	<p>y1 – output</p> <p>u1...ui – input</p> <p>ui+1 – table data</p>
<code>y1, y2 = prelookup(u1, u2, u3)</code>	<p>y1 – index</p> <p>y2 – fraction</p> <p>u1 – input</p> <p>u2 – breakpoint data</p> <p>u3 – number of breakpoints</p>

The following table lists required and optional parameters that pertain to lookup table algorithms. When defining a table entry programmatically:

- Set values for required parameters to achieve code replacement.
- If you do not set a value for an optional parameter, the code replacement software applies don't care. The software ignores the parameter while searching for matches.

Functions	Required Parameters	Optional Parameters
interp1D interp2D interp3D interp4D interp5D interpND	<p>Interpolation method (InterpMethod)</p> <p>Extrapolation method (ExtrapMethod)</p>	<p>Remove protection against out-of-range input (RemoveProtectionInput)</p> <p>Valid index input may reach last index (ValidIndexMayReachLast)</p> <p>Integer rounding mode (RndMeth)</p> <p>Saturate on integer overflow (SaturateOnIntegerOverflow)</p>
lookup1D lookup2D lookup3D lookup4D lookup5D lookupND	<p>Interpolation method (InterpMethod)</p> <p>Extrapolation method (ExtrapMethod)</p>	<p>Index search method (IndexSearchMethod)</p> <p>Remove protection against out-of-range input (RemoveProtectionInput)</p> <p>Use last table value for inputs at or above last breakpoint (UseLastTableValue)</p> <p>Integer rounding mode (RndMeth)</p> <p>Saturate on integer overflow (SaturateOnIntegerOverflow)</p>

Functions	Required Parameters	Optional Parameters
lookupND_Direct	Number of table dimensions (NumberOfTableDimensions) Inputs select this object from table (InputSelectThisObjectFromTab	Integer rounding mode (RndMeth)
prelookup	Extrapolation method (ExtrapMethod)	Index search method (IndexSearchMethod) Use last breakpoint for input at or above upper limit (UseLastBreakpoint) Remove protection against out-of-range input (RemoveProtectionInput) Integer rounding mode (RndMeth)

Interactive Mapping with Code Replacement Tool

This example shows how to specify a code replacement table entry for a lookup table algorithm by using the Code Replacement Tool.

Open and Examine Example Replacement Function

Identify or create the C or C++ replacement function for the algorithm that you want to use in place of an algorithm provided by the Simulink software.

This example uses the following C replacement function source and header files that are in the folder `matlab/help/toolbox/ecoder/examples/code_replacement`:

- `myLookup1D.c`
- `myLookup1D.h`

Place a copy of these files in your working folder.

Open and examine the code in `myLookup1D.c`. Note the function signature. When you enter the implementation argument specification in the Code Replacement Tool, you must specify argument properties.

```
#include "myLookup1D.h"

real_T my_Lookup1d_Replace(real_T u0, const real_T *bp0, const real_T *table)
{
```

```

real_T y;
uint16_T frac;
uint32_T bpIdx;
uint32_T maxIndex=9U;

if (u0 <= bp0[0U]) {
    bpIdx = 0U;
    frac = 0U;
} else if (u0 < bp0[maxIndex]) {
    bpIdx = maxIndex >> 1U;
    while ((u0 < bp0[bpIdx]) && (bpIdx > 0U)) {
        bpIdx--;
    }

    while ((bpIdx < maxIndex - 1U) && (u0 >= bp0[bpIdx + 1U])) {
        bpIdx++;
    }

    frac = (uint16_T)((u0 - bp0[bpIdx]) / (bp0[bpIdx + 1U] -
        bp0[bpIdx]) * 32768.0);
} else {
    bpIdx = maxIndex;
    frac = 0U;
}

if (bpIdx == maxIndex) {
    y = table[bpIdx];
} else {
    y = (table[bpIdx + 1U] - table[bpIdx]) * ((real_T)frac * 3.0517578125E-5) +
        table[bpIdx];
}

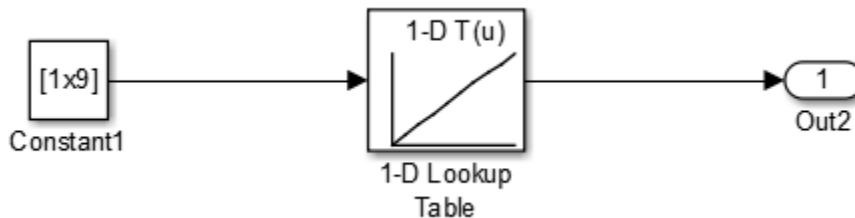
return y;
}

```

Open and Examine the Example Model

This example uses the model `matlab/help/toolbox/ecoder/examples/code_replacement/lookup1d.slx` to test your code replacement specification. Place a copy of the model in your working folder and name it `my_lookup1d.slx`.

Open and examine the model. Note input and output specifications and block parameter settings. To achieve a match, you must specify conceptual arguments based on how the 1D Lookup Table block is configured in the example model.



Create Code Replacement Table

- 1 At the command prompt, enter `crtool` to open the Code Replacement Tool.
- 2 Add a new table, select that table, and add a new function entry.
- 3 On the **Mapping Information** tab, select **Custom** for the **Function** parameter.
- 4 Look up the call signature and algorithm parameter information for the lookup function that you want to update with an algorithm replacement. See “Lookup Table Function Signatures” on page 21-107.

For this example, you replace the algorithm for the conceptual function associated with the 1-D Lookup Table block. The signature for that function is:

```
y1 = lookup1D(u1, u2, u3, u4)
```

Arguments `u1`, `u2`, `u3`, `u4` represent input, breakpoint data, table data, and table dimension length, respectively. The function returns output to `y1`.

Set the algorithm parameters **Extrapolation method** and **Interpolation method**. Optionally, you can specify values for:

- **Index search method**
- **Remove protection against out-of-range input in generated code**
- **Support tunable table size in code generation**
- **Use last table value for inputs at or above last breakpoint**

If the code generator finds a match of the conceptual function signature and algorithm parameter values, an algorithm replacement occurs.

- 5 To the right of the **Function** drop-down list, in the function-name text box, enter the name of the Simulink lookup table function. For this example, type the name `lookup1D`. Type the name exactly as it appears in the documented signature, including character casing. Press **Enter**.
- 6 Set the algorithm parameters to match the settings of the 1-D Lookup Table block in the example model to trigger a match.

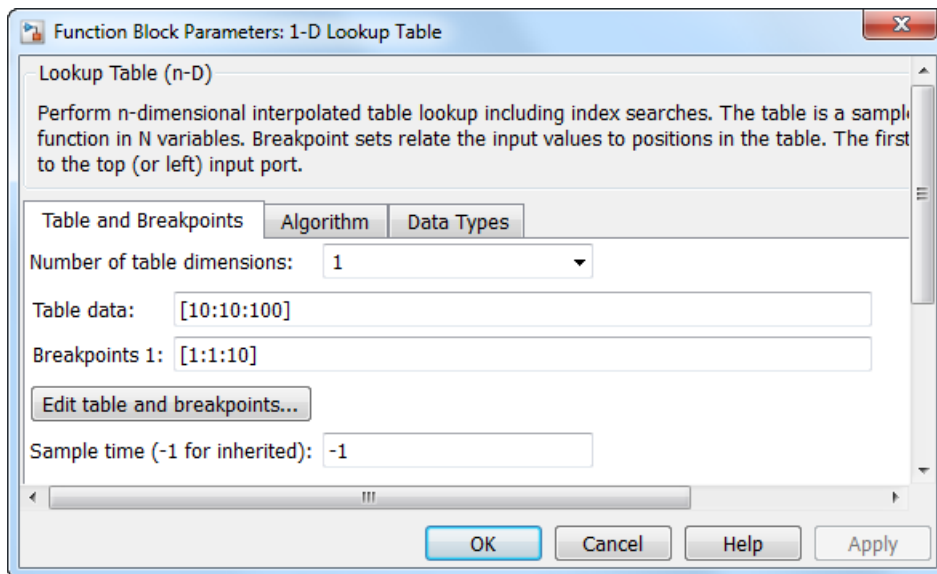
Set	To
Interpolation method	Linear
Extrapolation method	Clip
Index search method	Linear search

Set	To
Remove protection against out-of-range input	off
Support tunable table size	off
Use last table value for inputs at or above last breakpoint	on

- 7 Specify the conceptual arguments. Under the **Conceptual arguments** list box, click + to add the arguments that are in the documented function signature. The `lookup1D` function takes one output argument and three input arguments. Click + four times.

The tool creates an output argument `y1` and four input arguments `u1`, `u2`, `u3`, and `u4`. By default, the four arguments are scalars of type `double`.

You can adjust the conceptual argument properties. For this example, you do not make changes for `y1` and `u1`. However, as the block parameter dialog box for the example model shows, you must adjust the argument properties for the breakpoint and table data arguments.



Adjust the conceptual argument properties by using the following table. Then, click **Apply**.

Signature Argument Name	Conceptual Argument Name	Data type	I/O type	Argument type	Lower range	Upper range
y	y1	double	OUTPUT	Scalar	Not applicable	Not applicable
u1	u1	double	INPUT	Scalar	Not applicable	Not applicable
bp1	u2	double	INPUT	Matrix	[0 0]	[Inf Inf]
table	u3	double	INPUT	Matrix	[0 0]	[Inf Inf]
tdl	u4	uint32	INPUT	Scalar	Not applicable	Not applicable

- 8** Enter information for the replacement function prototype. The prototype for the example function is:

```
real_T my_Lookup1D_Repl(real_T u0, const real_T *bp0, const real_T *table, uint32_T tdl)
```

In the **Replacement function > Function prototype** section, type the function name `my_Lookup1D_Repl` in the **Name** text box.

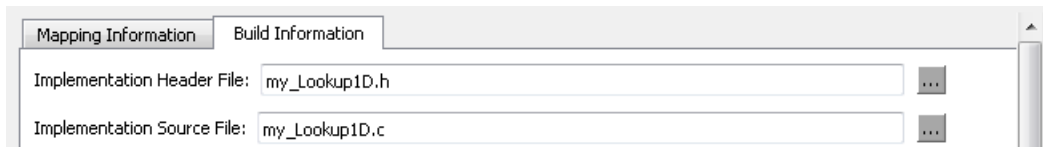
- 9** Specify the arguments for the replacement function. Under the **Function arguments** list box, click **+** four times to add four implementation arguments.

You might need to adjust the function argument properties. Make changes for `y1` and `u1`. However, as the replacement function signature shows, you must adjust the argument properties for the breakpoint, table data, and table dimension length arguments. For `u2` (breakpoints) and `u3` (table), select the **Const** check box. For `u4`, set **Data type** to `uint32`.

- 10** Set relevant implementation attributes. Use the default settings.
- 11** Validate the entry. If the tool reports errors, fix them, and retry the validation. Repeat until the tool does not report errors.
- 12** Save the code replacement table in your working folder as `my_lookup_replacement_table.m`.

Specify Build Information

On the **Build Information** tab, specify information relevant to generating C or C++ code and building an executable from the model. Enter `myLookup1D.h` for **Implementation Header File** and `myLookup1D.c` for **Implementation Source File**.



If you copied the example files to a folder other than the working folder containing the test model, `lookup1d.slx`, specify the source and header file paths. Otherwise, leave the other **Build Information** parameters set to default values. Click **Apply**.

Test the Entry

To test this example, use the example model `matlab/help/toolbox/ecoder/examples/code_replacement/lookup1d.slx`.

Configure the model with the following settings:

- On the **Solver** pane, select a fixed-step solver.
- On the **Code Generation** pane, select an ERT-based system target file.
- On the **Code Generation > Interface** pane, select the code replacement library that contains your memory function entry.

Programmatic Specification

This example shows how to programmatically specify code replacement table entries for lookup table functions.

Open and Examine Example Replacement Function

Identify or create the C or C++ replacement function for the algorithm that you want to use in place of an algorithm provided by the Simulink software.

This example uses the following C replacement function source and header files that are in the folder `matlab/help/toolbox/ecoder/examples/code_replacement`:

- `myLookup1D.c`

- myLookup1D.h

Place a copy of these files in your working folder.

Open and examine the code in myLookup1D.c. Note the function signature. When you enter the implementation argument specification in the Code Replacement Tool, you must specify argument properties.

```
#include "myLookup1D.h"

real_T my_Lookup1d_Replace(real_T u0, const real_T *bp0, const real_T *table)
{
    real_T y;
    uint16_T frac;
    uint32_T bpIdx;
    uint32_T maxIndex=9U;

    if (u0 <= bp0[0U]) {
        bpIdx = 0U;
        frac = 0U;
    } else if (u0 < bp0[maxIndex]) {
        bpIdx = maxIndex >> 1U;
        while ((u0 < bp0[bpIdx]) && (bpIdx > 0U)) {
            bpIdx--;
        }

        while ((bpIdx < maxIndex - 1U) && (u0 >= bp0[bpIdx + 1U])) {
            bpIdx++;
        }

        frac = (uint16_T)((u0 - bp0[bpIdx]) / (bp0[bpIdx + 1U] -
            bp0[bpIdx]) * 32768.0);
    } else {
        bpIdx = maxIndex;
        frac = 0U;
    }

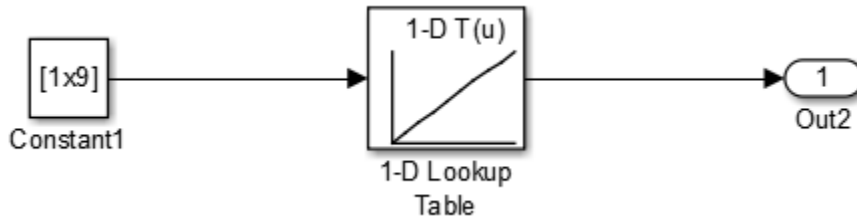
    if (bpIdx == maxIndex) {
        y = table[bpIdx];
    } else {
        y = (table[bpIdx + 1U] - table[bpIdx]) * ((real_T)frac * 3.0517578125E-5) +
            table[bpIdx];
    }

    return y;
}
```

Open and Examine the Example Model

This example uses the model matlab/help/toolbox/ecoder/examples/code_replacement/lookup1d.slx to test your code replacement specification. Place a copy of the model in your working folder and name it my_lookup1d.slx.

Open and examine the model. Note input and output specifications and block parameter settings. To achieve a match, you must specify conceptual arguments based on how the 1D Lookup Table block is configured in the example model.



Review Lookup Function Signature

Look up the call signature and algorithm parameter information for the lookup function that you want to update with an algorithm replacement. See “Lookup Table Function Signatures” on page 21-107.

Replace the algorithm for the function associated with the 1–D Lookup Table block. The signature for that function is:

$$y1 = \text{lookup1D}(u1, u2, u3, u4)$$

Arguments $u1$, $u2$, $u3$, and $u4$ represent input, breakpoint data, table data, and table dimension length, respectively. The function returns output to $y1$.

Set the algorithm parameters `ExtrapMethod` and `InterpMethod`. Optionally, you can specify values for:

- `IndexSearchMethod`
- `RemoveProtectionInput`
- `RndMeth`
- `SaturateOnIntegerOverflow`
- `UseLastTableValue`

If the code generator finds a match of the function signature and algorithm parameter settings, an algorithm replacement occurs.

Create Code Replacement Entry

Create a code replacement table file, as a MATLAB function, that describes the lookup table function code replacement table entries. Place a copy of the file `matlab/help/toolbox/ecoder/examples/code_replacement/Lookup1D_CRLtable.m` in

your working folder. This file defines a code replacement table for the C function `my_Lookup1D_Repl`.

Open `Lookup1D_CRLtable.m` and examine the definition.

- 1 Define a function for the definition.

```
function hLib = myLookup1DCRL_table
```

- 2 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hLib = RTW.Tf1Table;
```

- 3 Create an entry for the function mapping with a call to the `RTW.Tf1CFunctionEntry` function.

```
hEnt = RTW.Tf1CFunctionEntry;
```

- 4 Set function entry parameters with a call to the `setTf1CFunctionEntryParameters` function. The function key, implementation name, and header and source files specified in the function call identify the Simulink lookup table function name, `lookup1D` and the function name, header file, and source file for the replacement function `my_Lookup1D_Repl`. Specify the Simulink lookup table function name exactly as it appears in the documented signature, including character casing (see “Lookup Table Function Signatures” on page 21-107). If you copied the example files to a folder other than the working folder that contains the test model, `lookup1d.slx`, specify the source and header file paths.

```
setTf1CFunctionEntryParameters(hEnt, ...
    'Key', 'lookup1D', ...
    'Priority', 100, ...
    'ImplementationName', 'my_Lookup1D_Repl', ...
    'ImplementationHeaderFile', 'myLookup1D.h', ...
    'ImplementationSourceFile', 'myLookup1D.c', ...
    'GenCallback', 'RTW.copyFileToBuildDir');
```

- 5 Create conceptual arguments and add them to the entry. This example uses calls to the `getTf1ArgFromString` and `addConceptualArg` functions to create and add the arguments.

The example defines five conceptual arguments for the `lookup1D` function, one output argument `y1` and four input arguments `u1`, `u2`, `u3`, and `u4`. Arguments `y1` and `u1` are defined as scalar `double` data. Arguments `u2` and `u3` represent `bp1` and `table` in the signature and are defined as 1x10 matrices of `double` data. Argument `u4` represents `td1` and is defined as scalar of `uint32` data. This definition triggers a match with the example model.

```

arg = hEnt.getTf1ArgFromString('y1','double');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEnt, arg);

arg = hEnt.getTf1ArgFromString('u1','double');
addConceptualArg(hEnt, arg);

arg = RTW.Tf1ArgMatrix('u2', 'RTW_IO_INPUT', 'double');
arg.DimRange = [0 0; Inf Inf];
addConceptualArg(hEnt, arg);

arg = RTW.Tf1ArgMatrix('u3', 'RTW_IO_INPUT', 'double');
arg.DimRange = [0 0; Inf Inf];
addConceptualArg(hEnt, arg);

arg = hEnt.getTf1ArgFromString('u4','uint32');
addConceptualArg(hEnt, arg);

```

- 6 Set the algorithm properties for Lookup1D with calls to the function `addAlgorithmProperty`. The calls to `addAlgorithmProperty` set the algorithm parameters as listed in the following table. These settings trigger a match with the example model.

Set	To
InterpMethod	Linear
ExtrapMethod	Clip
IndexSearchMethod	Linear search
RemoveProtectionInput	Off
UseLastTableValue	On

```

addAlgorithmProperty(hEnt, 'ExtrapMethod', 'Clip');
addAlgorithmProperty(hEnt, 'InterpMethod', 'Linear');
addAlgorithmProperty(hEnt, 'IndexSearchMethod', 'Linear search');
addAlgorithmProperty(hEnt, 'UseLastTableValue', 'on');

```

- 7 Create the implementation arguments and add them to the entry. This example uses calls to the `getTf1ArgFromString` function to create five implementation arguments that map to arguments in the replacement function prototype: one output argument `y1` and four input arguments `u1`, `u2`, `u3`, and `u4`. The convenience methods `setReturn` and `addArgument` specify whether an argument is a return value or argument and adds the argument to the entry's array of implementation arguments.

```

arg = hEnt.getTf1ArgFromString('y1','double');
arg.IOType = 'RTW_IO_OUTPUT';

```

```

hEnt.Implementation.setReturn(arg);

arg = hEnt.getTf1ArgFromString('u1', 'double');
hEnt.Implementation.addArgument(arg);

arg = hEnt.getTf1ArgFromString('u2', 'double*');
hEnt.Implementation.addArgument(arg);

arg = hEnt.getTf1ArgFromString('u3', 'double*');
hEnt.Implementation.addArgument(arg);

arg = hEnt.getTf1ArgFromString('u4', 'uint32');
hEnt.Implementation.addArgument(arg);

```

- 8 Add the entry to a code replacement table with a call to the `addEntry` function.

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

Test the Entry

To test this example, use the example model `matlab/help/toolbox/ecoder/examples/code_replacement/lookup1d.slx`.

Configure the model with the following settings:

- On the **Solver** pane, select a fixed-step solver.
- On the **Code Generation** pane, select an ERT-based system target file.
- On the **Code Generation > Interface** pane, select the code replacement library that contains your memory function entry.

Sample Code Replacement Definition for the `lookup2D` Function

The following code shows a replacement definition for the `lookup2D` function.

```

function hLib = my_2dlookup_replacement_table

hLib = RTW.Tf1Table;

hEnt = RTW.Tf1CFunctionEntry;
setTf1CFunctionEntryParameters(hEnt, ...
    'Key', 'lookup2D', ...
    'Priority', 100, ...
    'ImplementationName', 'custom_lookup2d', ...

```

```
        'ImplementationHeaderFile', 'custom_lookup2d.h', ...
        'ImplementationSourceFile', 'custom_lookup2d.c', ...
        'GenCallback',             'RTW.copyFileToBuildDir');

% Conceptual Args

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

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

arg = hEnt.getTflArgFromString('u2','double');
addConceptualArg(hEnt, arg);

arg = RTW.TflArgMatrix('u3', 'RTW_IO_INPUT', 'double');
arg.DimRange = [1 1; 10 1];
addConceptualArg(hEnt, arg);

arg = RTW.TflArgMatrix('u4', 'RTW_IO_INPUT', 'double');
arg.DimRange = [1 1; 10 1];
addConceptualArg(hEnt, arg);

arg = RTW.TflArgMatrix('u5', 'RTW_IO_INPUT', 'double');
arg.DimRange = [1 1; 10 1];
addConceptualArg(hEnt, arg);

arg = hEnt.getTflArgFromString('u6','uint32');
addConceptualArg(hEnt, arg);

arg = hEnt.getTflArgFromString('u7','uint32');
addConceptualArg(hEnt, arg);

% Algorithm Parameters

addAlgorithmProperty(hEnt, 'ExtrapMethod','Clip');
addAlgorithmProperty(hEnt, 'IndexSearchMethod','Linear search');
addAlgorithmProperty(hEnt, 'InterpMethod','Linear');
addAlgorithmProperty(hEnt, 'RemoveProtectionInput','off');
addAlgorithmProperty(hEnt, 'UseLastTableValue','on');

% Implementation Args
```

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

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

arg = hEnt.getTflArgFromString('u2','double');
hEnt.Implementation.addArgument(arg);

arg = hEnt.getTflArgFromString('u3','double*');
arg.Type.BaseType.ReadOnly = true;
hEnt.Implementation.addArgument(arg);

arg = hEnt.getTflArgFromString('u4','double*');
arg.Type.BaseType.ReadOnly = true;
hEnt.Implementation.addArgument(arg);

arg = hEnt.getTflArgFromString('u5','double*');
arg.Type.BaseType.ReadOnly = true;
hEnt.Implementation.addArgument(arg);

arg = hEnt.getTflArgFromString('u6','uint32');
hEnt.Implementation.addArgument(arg);

arg = hEnt.getTflArgFromString('u7','uint32');
hEnt.Implementation.addArgument(arg);

arg = hEnt.getTflArgFromString('u6','uint32');
hEnt.Implementation.addArguments(arg);

arg = hEnt.getTflArgFromString('u7','uint32');
hEnt.Implementation.addArguments(arg);

hLib.addEntry(hEnt);
```

Related Examples

- “Define Code Replacement Mappings”
- “Review and Test Code Replacements”
- “Data Alignment for Code Replacement”
- “Reserved Identifiers and Code Replacement ”
- “Customize Matching and Replacement Process for Functions”

- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “Code You Can Replace From Simulink Models”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Data Alignment for Code Replacement

In this section...

“Code Replacement Data Alignment” on page 21-125

“Specify Data Alignment Requirements for Function Arguments” on page 21-125

“Provide Data Alignment Specifications for Compilers” on page 21-127

“Basic Example of Code Replacement Data Alignment” on page 21-132

Code Replacement Data Alignment

Code replacement libraries can align data objects passed into a replacement function to a specified boundary. You can take advantage of function implementations that require aligned data to optimize application performance. To configure data alignment for a function implementation:

- 1 Specify the data alignment requirements in a code replacement entry. Specify alignment separately for each implementation function argument or collectively for all function arguments. See “Specify Data Alignment Requirements for Function Arguments” on page 21-125.
- 2 Specify the data alignment capabilities and syntax for one or more compilers. Include the alignment specifications in a library registration entry in the `rtwTargetInfo.m` file. See “Provide Data Alignment Specifications for Compilers” on page 21-127.
- 3 Register the library containing the table entry and alignment specification object.
- 4 Configure the code generator to use the code replacement library and generate code. Observe the results.

For examples, see “Basic Example of Code Replacement Data Alignment” on page 21-132 and the “Data Alignment for Function Implementations” section of the “Replacing Math Functions and Operators” example page.

Specify Data Alignment Requirements for Function Arguments

To specify the data alignment requirement for an argument in a code replacement entry:

- If you are defining a replacement function in a code replacement table registration file, create an argument descriptor object (`RTW.ArgumentDescriptor`). Use its

AlignmentBoundary property to specify the required alignment boundary and assign the object to the argument's **Descriptor** property.

- If you are defining a replacement function using the Code Replacement Tool, on the **Mapping Information** tab, in the **Argument properties** section for the replacement function, enter a value for the **Alignment value** parameter.

The screenshot shows the 'Replacement function' dialog box. It is divided into several sections:

- Function prototype:**
 - Name:
 - C++ namespace:
 - Function returns void
- Function arguments:** A list box containing 'y1(return arg)' and 'u1'. There are up and down arrow buttons next to the list.
- Argument properties:**
 - Data type:
 - I/O type:
 - Const
 - Pointer
 - Complex
 - Alignment value:

The **AlignmentBoundary** property (or **Alignment value** parameter) specifies the alignment boundary for data passed to a function argument, in number of bytes. The **AlignmentBoundary** property is valid only for addressable objects, including matrix and pointer arguments. It is not applicable for value arguments. Valid values are:

- -1 (default) — If the data is a `Simulink.Bus`, `Simulink.Signal`, or `Simulink.Parameter` object, specifies that the code generator determines an optimal alignment based on usage. Otherwise, specifies that there is not an alignment requirement for this argument.
- Positive integer that is a power of 2, not exceeding 128 — Specifies number of bytes in the boundary. The starting memory address for the data allocated for the function argument is a multiple of the specified value. If you specify an alignment boundary that is less than the natural alignment of the argument data type, the alignment directive is emitted in the generated code, but is silently ignored by the target compiler.

The following code specifies the **AlignmentBoundary** for an argument as 16 bytes.

```
hLib = RTW.Tf1Table;
entry = RTW.Tf1COperationEntry;
```



```

arg = getTflArgFromString(hLib, 'u1', 'single*');
desc = RTW.ArgumentDescriptor;
desc.AlignmentBoundary = 16;
arg.Descriptor = desc;
entry.Implementation.addArgument(arg);

```

The equivalent alignment boundary specification in the Code Replacement Tool dialog box is in this figure.

The figure shows a dialog box titled "Argument properties". It contains the following controls:

- "Data type:" dropdown menu set to "single".
- "I/O type:" dropdown menu set to "INPUT".
- Three checkboxes: "Const" (unchecked), "Pointer" (checked), and "Complex" (unchecked).
- "Alignment value:" text input field containing the number "16".

Note: If your model imports `Simulink.Bus`, `Simulink.Parameter`, or `Simulink.Signal` objects, specify an alignment boundary in the object properties, using the **Alignment** property. For more information, see “`Simulink.Bus`”, “`Simulink.Parameter`”, and “`Simulink.Signal`”.

Provide Data Alignment Specifications for Compilers

To support data alignment in generated code, describe the data alignment capabilities and syntax for your compilers in the code replacement library registration. Provide one or more alignment specifications for each compiler in a library registry entry.

To describe the data alignment capabilities and syntax for a compiler:

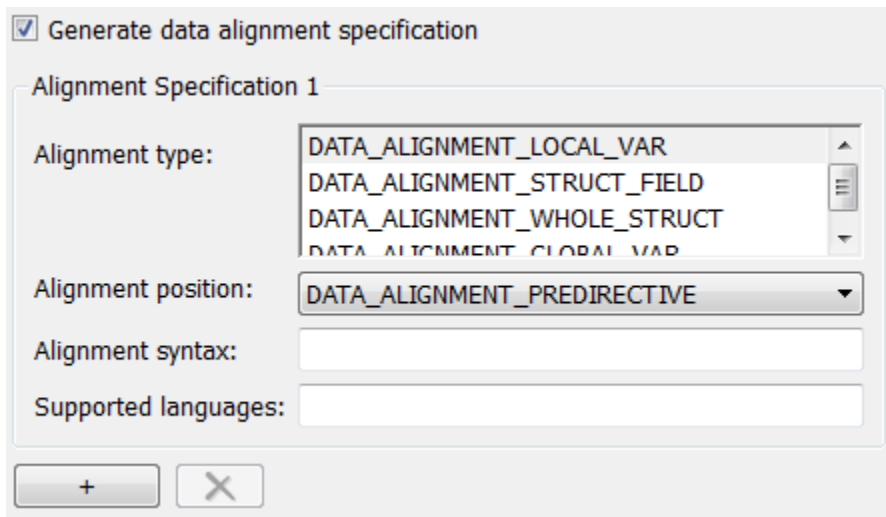
- If you are defining a code replacement library registration entry in a `rtwTargetInfo.m` customization file, add one or more `AlignmentSpecification` objects to an `RTW.DataAlignment` object. Attach the `RTW.DataAlignment` object to the `TargetCharacteristics` object of the registry entry.

The `RTW.DataAlignment` object also has the property `DefaultMallocAlignment`, which specifies the default alignment boundary, in bytes, that the compiler uses for dynamically allocated memory. If the code generator uses dynamic memory allocation for a data object involved in a code replacement, this value determines

if the dynamically allocated memory satisfies the alignment requirement of the replacement. If not, the code generator does not use the replacement. The default value for `DefaultMallocAlignment` is -1, indicating that the default alignment boundary used for dynamically allocated memory is unknown. In this case, the code generator uses the natural alignment of the data type to determine whether to allow a replacement.

Additionally, you can specify the alignment boundary for complex types by using the `addComplexTypeAlignment` function.

- If you are generating a customization file function using the Code Replacement Tool, fill out the following fields for each compiler.



Click the plus (+) symbol to add additional compiler specifications.

For each data alignment specification, provide the following information.

Alignment-Specification Property	Dialog Box Parameter	Description
AlignmentType	Alignment type	Cell array of predefined enumerated strings, specifying which types of alignment this specification supports. <ul style="list-style-type: none"> • <code>DATA_ALIGNMENT_LOCAL_VAR</code> — Local variables.

Alignment-Specification Property	Dialog Box Parameter	Description
		<ul style="list-style-type: none">• DATA_ALIGNMENT_GLOBAL_VAR — Global variables.• DATA_ALIGNMENT_STRUCT_FIELD — Individual structure fields.• DATA_ALIGNMENT_WHOLE_STRUCT — Whole structure, with padding (individual structure field alignment, if specified, is favored and takes precedence over whole structure alignment). <p>Each alignment specification must specify at least DATA_ALIGNMENT_GLOBAL_VAR and DATA_ALIGNMENT_STRUCT_FIELD.</p>

Alignment-Specification Property	Dialog Box Parameter	Description
AlignmentPosition	Alignment position	<p>Predefined enumerated string specifying the position in which you must place the compiler alignment directive for alignment type <code>DATA_ALIGNMENT_WHOLE_STRUCT</code>:</p> <ul style="list-style-type: none"> • <code>DATA_ALIGNMENT_PREDIRECTIVE</code> — The alignment directive is emitted before <code>struct st_tag{...}</code>, as part of the type definition statement (for example, MSVC). • <code>DATA_ALIGNMENT_POSTDIRECTIVE</code> — The alignment directive is emitted after <code>struct st_tag{...}</code>, as part of the type definition statement (for example, gcc). • <code>DATA_ALIGNMENT_PRECEDING_STATEMENT</code> — The alignment directive is emitted as a stand alone statement immediately preceding the definition of the structure type. A semicolon (;) must terminate the registered alignment syntax. • <code>DATA_ALIGNMENT_FOLLOWING_STATEMENT</code> — The alignment directive is emitted as a stand alone statement immediately following the definition of the structure type. A semicolon (;) must terminate the registered alignment syntax. <p>For alignment types other than <code>DATA_ALIGNMENT_WHOLE_STRUCT</code>, code generation uses alignment position <code>DATA_ALIGNMENT_PREDIRECTIVE</code>.</p>

Alignment-Specification Property	Dialog Box Parameter	Description
AlignmentSyntax-Template	Alignment syntax	<p>Specifies the alignment directive string that the compiler supports. The string is registered as a syntax template that has placeholders in it. These placeholders are supported:</p> <ul style="list-style-type: none"> • %n — Replaced by the alignment boundary for the replacement function argument. • %s — Replaced by the symbol that will be aligned, usually the identifier of a variable. <p>For example, for the gcc compiler, you can specify <code>__attribute__((aligned(%n)))</code>, or for the MSVC compiler, <code>__declspec(align(%n))</code>.</p>
SupportedLanguage	Supported languages	<p>Cell array specifying the languages to which this alignment specification applies, among c and c++. Sometimes alignment syntax and position differ between languages for a compiler.</p> <p>.</p>

Here is a data alignment specification for the GCC compiler:

```

da = RTW.DataAlignment;

as = RTW.AlignmentSpecification;
as.AlignmentType = {'DATA_ALIGNMENT_LOCAL_VAR', ...
                  'DATA_ALIGNMENT_STRUCT_FIELD', ...
                  'DATA_ALIGNMENT_GLOBAL_VAR'};
as.AlignmentSyntaxTemplate = '__attribute__((aligned(%n)))';
as.AlignmentPosition = 'DATA_ALIGNMENT_PREDIRECTIVE';
as.SupportedLanguages = {'c', 'c++'};
da.addAlignmentSpecification(as);

tc = RTW.TargetCharacteristics;
tc.DataAlignment = da;

```

Here is the corresponding specification in the **Generate customization** dialog box of the Code Replacement Tool.

Generate data alignment specification

Alignment Specification 1

Alignment type: DATA_ALIGNMENT_LOCAL_VAR
DATA_ALIGNMENT_STRUCT_FIELD
DATA_ALIGNMENT_WHOLE_STRUCT
DATA_ALIGNMENT_GLOBAL_VAR

Alignment position: DATA_ALIGNMENT_PREDIRECTIVE

Alignment syntax: __attribute__((aligned(%n)))

Supported languages: c, c++

Basic Example of Code Replacement Data Alignment

A simple example of the complete workflow for data alignment specified for code replacement is:

- 1 Create and save the following code replacement table definition file, `cr1_table_mmul_4x4_single_align.m`. This table defines a replacement entry for the `*` (multiplication) operator, the `single` data type, and input dimensions `[4,4]`. The entry also specifies a data alignment boundary of 16 bytes for each replacement function argument. The entry expresses the requirement that the starting memory address for the data allocated for the function arguments during code generation is a multiple of 16.

```
function hLib = cr1_table_mmul_4x4_single_align
%CRL_TABLE_MMUL_4x4_SINGLE_ALIGN - Describe matrix operator entry with data alignment

hLib = RTW.Tf1Table;
entry = RTW.Tf1COperationEntry;
setTf1COperationEntryParameters(entry, ...
    'Key', 'RTW_OP_MUL', ...
    'Priority', 90, ...
    'ImplementationName', 'matrix_mul_4x4_s');

% conceptual arguments
createAndAddConceptualArg(entry, 'RTW.Tf1ArgMatrix', ...
    'Name', 'y1', ...
    'IOType', 'RTW_IO_OUTPUT', ...
    'BaseType', 'single', ...
    'DimRange', [4 4]);
```

```

createAndAddConceptualArg(entry, 'RTW.Tf1ArgMatrix',...
                             'Name',      'u1', ...
                             'BaseType',  'single', ...
                             'DimRange',  [4 4]);

createAndAddConceptualArg(entry, 'RTW.Tf1ArgMatrix',...
                             'Name',      'u2', ...
                             'BaseType',  'single', ...
                             'DimRange',  [4 4]);

% implementation arguments
arg = getTf1ArgFromString(hLib, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
entry.Implementation.setReturn(arg);

arg = getTf1ArgFromString(hLib, 'y1', 'single*');
desc = RTW.ArgumentDescriptor;
desc.AlignmentBoundary = 16;
arg.Descriptor = desc;
entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hLib, 'u1', 'single*');
desc = RTW.ArgumentDescriptor;
desc.AlignmentBoundary = 16;
arg.Descriptor = desc;
entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hLib, 'u2', 'single*');
desc = RTW.ArgumentDescriptor;
desc.AlignmentBoundary = 16;
arg.Descriptor = desc;
entry.Implementation.addArgument(arg);

hLib.addEntry(entry);

```

- 2 Create and save the following registration file, `rtwTargetInfo.m`. If you want to compile the code generated in this example, first modify the `AlignmentSyntaxTemplate` property for the compiler that you use. For example, for the MSVC compiler, replace the gcc template string `__attribute__((aligned(%n)))` with `__declspec(align(%n))`.

```

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);

% 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
```

- 3 To register your library with code generator without having to restart MATLAB, enter this command:

```
RTW.TargetRegistry.getInstance('reset');
```
- 4 Configure the code generator to use your code replacement library.
- 5 Generate code and a code generation report.
- 6 Review the code replacements. For example, check whether a multiplication operation is replaced with a `matrix_mul_4x4_s` function call. In `mmalign.h`, check whether the gcc alignment directive `__attribute__((aligned(16)))` is generated to align the function variables.

Related Examples

- “Define Code Replacement Mappings”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “Code You Can Replace From Simulink Models”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Replace MATLAB Functions with Custom Code Using `coder.replace`

The `coder.replace` function provides the ability to replace a specified MATLAB function with a code replacement function in generated code. Use `coder.replace` in MATLAB code from which you want to generate C code using:

- MATLAB Coder
- MATLAB code in a Simulink MATLAB Function block

You can replace MATLAB functions that have:

- Single or multiple inputs
- Single or multiple outputs
- Scalar and matrix inputs and outputs

Supported types include:

- `single`, `double` (complex and noncomplex)
- `int8`, `uint8` (complex and noncomplex)
- `int16`, `uint16` (complex and noncomplex)
- `int32`, `uint32` (complex and noncomplex)
- Fixed-point integers
- Mixed types (different type on each input)

Related Examples

- “Define Code Replacement Mappings”
- “Register Code Replacement Mappings”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “Code You Can Replace From Simulink Models”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Replace MATLAB Functions Specified in MATLAB Function Blocks

This example shows how to use code replacement to replace a **MATLAB** function specified in a MATLAB Function block.

- 1 Open the `ex_replace` model. At the command prompt, enter:

```
addpath(fullfile(docroot,'toolbox','ecoder','examples'))
ex_replace
```

- 2 View the MATLAB Function Block code. In the model, double-click the MATLAB Function block to view the code in the MATLAB editor.

```
function y = customFcn(u1, u2) %#codegen
% This block supports MATLAB for code generation.

% Replace this MATLAB function with CRL replacement function and if no
% CRL replacement is found, generate an error during code generation.
coder.replace('-errorifnoreplacement');

assert(isa(u1,'int32'));
assert(isa(u2,'int32'));

y = power(u1,u2);
```

The `coder.replace('-errorifnoreplacement')` statement instructs the code generator to replace this MATLAB function with a code replacement library function and generate an error if it does not find a code replacement entry match.

- 3 Define a function for the definition.

```
function hTable = crl_table_coderreplace()
```

- 4 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 5 Create an entry for the function mapping with a call to the `RTW.Tf1CFunctionEntry` function.

```
hEnt = RTW.Tf1CFunctionEntry;
```

- 6 Set function entry parameters with a call to the `setTf1CFunctionEntryParameters` function.

```
setTf1CFunctionEntryParameters(hEnt, ...
    'Key', ...
    'customFcn', ...
```

```

    'Priority',          100, ...
    'ImplementationName', 'scalarFcnReplacement', ...
    'ImplementationHeaderFile', 'MyMath.h', ...
    'ImplementationSourceFile', 'MyMath.c')

```

- 7 Create conceptual arguments `y1`, `u1` and `u1`. This example uses calls to the `getTflArgFromString` and `addConceptualArg` functions to create and add the arguments.

```

arg = getTflArgFromString(hEnt, 'y1', 'int32');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEnt, arg);

```

```

arg = getTflArgFromString(hEnt, 'u1', 'int32');
addConceptualArg(hEnt, arg);

```

```

arg = getTflArgFromString(hEnt, 'u2', 'int32');
addConceptualArg(hEnt, arg);

```

- 8 Create the implementation arguments and add them to the entry. This example uses calls to the `getTflArgFromString` function to create implementation arguments that map to arguments in the replacement function prototype: output argument `void`, input arguments `u1` and `u2`, and output argument `y1`. The convenience methods `setReturn` and `addArgument` specify whether an argument is a return value or argument and adds the argument to the entry's array of implementation arguments.

```

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

```

```

arg = getTflArgFromString(hEnt, 'u1', 'int32');
hEnt.Implementation.addArgument(arg);

```

```

arg = getTflArgFromString(hEnt, 'u2', 'int32');
hEnt.Implementation.addArgument(arg);

```

```

arg = getTflArgFromString(hEnt, 'y1', 'int32*');
arg.IOType = 'RTW_IO_OUTPUT';
hEnt.Implementation.addArgument(arg);

```

- 9 Add the entry to a code replacement table with a call to the `addEntry` function.

```

addEntry(hLib, hEnt);

```

To test the example:

- 1 Register the table that contains the entry in a code replacement library.

- 2 Create files `MyMath.c` and `MyMath.h` that define the replacement function, `scalarFcnReplacement`, which has two `int32` inputs and one `int32` output.

`MyMath.c`

```
#include "MyMath.h"

void scalarFcnReplacement(int32_T u1, int32_T u2, int32_T* y1 ) {
    *y1 = u1^u2;
}
```

`MyMath.h`

```
#ifndef _ScalarMath_h
#define _ScalarMath_h

#include "rtwtypes.h"

#ifdef __cplusplus
extern "C" {
#endif

extern void scalarFcnReplacement(int32_T u1, int32_T u2, int32_T* y1);

#ifdef __cplusplus
}
#endif

#endif
```

- 3 Open the `ex_replace` model.
- 4 Configure the code generator to use the code replacement library and to include the Code Replacements Report in the code generation report.
- 5 Generate the replacement code and a code generation report.
- 6 Review the code replacements. In the code generation report, view the generated code for `ex_replace.c`.

```
void ex_replace_step(void)
{
    int32_T y;
    scalarFcnReplacement(ex_replace_U.In1, ex_replace_U.In2, &y);
    ex_replace_Y.Out1 = y;
}
```

Related Examples

- “Define Code Replacement Mappings”
- “Register Code Replacement Mappings”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “Code You Can Replace From Simulink Models”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Reserved Identifiers and Code Replacement

The code generator and C programming language use, internally, reserved keywords for code generation. Do not use reserved keywords as identifiers or function names. Reserved keywords for code generation include many code replacement library identifiers, the majority of which are function names, such as `acos`.

To view a list of reserved identifiers for the code replacement library that you use to generate code, specify the name of the library in a call to the function `RTW.TargetRegistry.getInstance.getTf1ReservedIdentifiers`. For example:

```
cr1_ids = RTW.TargetRegistry.getInstance.getTf1ReservedIdentifiers('GNU99 (GNU)')
```

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.

To register additional code replacement reserved identifiers, use the `setReservedIdentifiers` function. This function registers specified reserved identifiers to be associated with a code replacement table.

You can register up to four reserved identifier structures in a code replacement table. You can associate one set of reserved identifiers with a code replacement library, while the other three (if present) must be associated with ANSI C. The following example shows a reserved identifier structure that specifies two identifiers and the associated header file.

```
d{1}.LibraryName = 'ANSI_C';  
d{1}.HeaderInfos{1}.HeaderName = 'math.h';  
d{1}.HeaderInfos{1}.ReservedIds = {'y0', 'y1'};
```

The code generator adds the identifiers to the list of reserved identifiers and honors them during the build procedure.

Related Examples

- “Define Code Replacement Mappings”
- “Register Code Replacement Mappings”
- “Review and Test Code Replacements”
- “Develop a Code Replacement Library”

- “Quick Start Library Development”

More About

- “What Is Code Replacement Customization?”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Customize Matching and Replacement Process for Functions

During the build process, the code generator uses:

- Preset match criteria to identify functions and operators for which application-specific implementations replace default implementations.
- Preset replacement function signatures.

However, preset match criteria and preset replacement function signatures might not completely meet your function and operator replacement needs. For example,

- You want to replace an operator with a particular fixed-point implementation function only when fraction lengths are within a particular range.
- When a match is made, you want to modify your replacement function signature based on compile-time information, such as passing fraction-length values into the function.

When you need to add extra logic into the code replacement matching and replacement process, create custom code replacement table entries. With custom entries you can specify additional match criteria and modify the replacement function signature to meet application needs.

To create a custom code replacement entry:

- 1** Create a custom code replacement entry class, derived from `RTW.Tf1CFunctionEntryML` (for function replacement) or `RTW.Tf1COperationEntryML` (for operator replacement).
- 2** In your derived class, implement a `do_match` method with a fixed preset signature as a MATLAB function. In your `do_match` method, provide either or both of the following customizations that instantiate the class:
 - a** Add additional match criteria that the base class does not provide. The base class provides a match based on argument number, argument name, signedness, word size, slope (if not specified with wildcards), bias (if not specified with wildcards), math modes such as saturation and rounding, and operator or function key. Accept a match only when additional size or range conditions are met.
 - b** Modify the implementation signature by adding additional arguments or setting constant input argument values. You can inject a constant value, such as an input scaling value, as an additional argument to the replacement function.

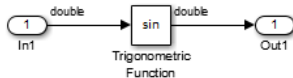
- 3 Create code replacement entries that instantiate your custom entry class.
- 4 Register a library containing the code replacement table that includes your entries.

During code generation, the code replacement matching process first tries to match function or operator call sites with the base class of your derived entry class. If the process finds a match, the software calls your `do_match` method to execute your additional match logic (if any) and your replacement function customizations (if any).

Customize Code Matching and Replacement for Functions

This example shows how to use custom code replacement table entries to refine the matching and replacement logic for functions. Modify a sine function replacement only if the integer size on the current target platform is 32 bits, such that the implementation function passes in a degrees-versus-radians flag as an input argument.

- 1 To exercise the custom code replacement table entries that you create in this example, create an ERT-based model with a sine function block, such as this model.



In the import block parameters, set the signal **Data type** to **double**. If the target platform selected for your model on the **Hardware Implementation** pane in the Configuration Parameters dialog box supports an integer size other than 32, either temporarily select a target platform with a 32-bit integer size, or modify the code to match the integer size of your target platform.

- 2 Create a class folder using the name of your derived class, such as `@Tf1CustomFunctionEntry`. Verify that the class folder is on the MATLAB search path or in your current working folder.
- 3 In the class folder, create and save the following class definition file, `Tf1CustomFunctionEntry.m`. This file defines the class `Tf1CustomFunctionEntry`, which is derived from the base class `RTW.Tf1FunctionEntryML`.

The derived class defines a `do_match` method. In the `do_match` method signature:

- `ent` is the return handle, which is returned either as empty (indicating that the match failed) or as a `Tf1FunctionEntry` handle.

- `hThis` is the handle to this object.
- `hCSO` is a handle to an object created by the code generator for querying the library for a replacement.
- The remaining arguments are the number of bits for various data types of the current target.

The `do_match` method adds required additional match criteria not provided by the base class and makes required modifications to the implementation signature. In this case, the `do_match` method must match only `targetBitPerInt`, representing the number of bits in the `C int` data type for the current target, to the value 32. If a match is made, the method sets the return handle and creates and adds an input argument, representing whether units are expressed as degrees or radians, to the replacement function signature.

Note: Alternatively, you can create and add the additional implementation function argument for passing a units flag in each code replacement table definition file that instantiates this class. In that case, this class definition code does not create the argument. That code only sets the argument value. For an example of creating and adding additional implementation function arguments in a table definition file, see “Customize Matching and Replacement Process for Operators” on page 21-180.

```
classdef Tf1CustomFunctionEntry < RTW.Tf1CFunctionEntryML
    methods
        function ent = do_match(hThis, ...
            hCSO, ... %#ok
            targetBitPerChar, ... %#ok
            targetBitPerShort, ... %#ok
            targetBitPerInt, ... %#ok
            targetBitPerLong) %#ok
            % DO_MATCH - Create a custom match function. The base class
            % checks the types of the arguments prior to calling this
            % method. This will check additional data and perhaps modify
            % the implementation function.

            ent = []; % default the return to empty, indicating the match failed.

            % Match sine function only if the target int size is 32 bits
            if targetBitPerInt == 32
                % Need to modify the default implementation, starting from a copy
                % of the standard Tf1CFunctionEntry.
                ent = RTW.Tf1CFunctionEntry(hThis);

                % If the target int size is 32 bits, the implementation function
                % takes an additional input flag argument indicating degrees vs.
```

```

% radians. The additional argument can be created and added either
% in the CRL table definition file that instantiates this class, or
% here in the class definition, as follows:
createAndAddImplementationArg(ent, 'RTW.Tf1ArgNumericConstant', ...
                                'Name',      'u2', ...
                                'IsSigned',   true, ...
                                'WordLength', 32, ...
                                'FractionLength', 0, ...
                                'Value',      1);
    end
end
end
end

```

Exit the class folder and return to the previous working folder.

- 4 Create and save the following code replacement table definition file, `crl_table_custom_sinfcn_double.m`. This file defines a code replacement table containing a function table entry for sine with `double` input and output. This entry instantiates the derived class from the previous step, `Tf1CustomFunctionEntry`.

```

function hTable = crl_table_custom_sinfcn_double

hTable = RTW.Tf1Table;

%% Add Tf1CustomFunctionEntry
fcn_entry = Tf1CustomFunctionEntry;
setTf1CFunctionEntryParameters(fcn_entry, ...
    'Key',      'sin', ...
    'Priority', 30, ...
    'ImplementationName', 'mySin', ...
    'ImplementationHeaderFile', 'mySin.h', ...
    'ImplementationSourceFile', 'mySin.c');

createAndAddConceptualArg(fcn_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',      'y1', ...
    'IOType',    'RTW_IO_OUTPUT', ...
    'DataTypeMode', 'double');

createAndAddConceptualArg(fcn_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',      'u1', ...
    'IOType',    'RTW_IO_INPUT', ...
    'DataTypeMode', 'double');

% Tf1CustomFunctionEntry class do_match method will create and add
% an implementation function argument during code generation if
% the supported integer size on the current target is 32 bits.
copyConceptualArgsToImplementation(fcn_entry);

addEntry(hTable, fcn_entry);

```

- 5 Optionally, perform a quick check of the validity of the function entry by invoking the table definition file at the command prompt (`>> tbl =`

`cr1_table_custom_sinfcn_double`) and by viewing it in the Code Replacement Viewer (`>> RTW.viewTfl(cr1_table_custom_sinfcn_double)`). For more information about validating code replacement tables, see “Review and Test Code Replacements” on page 21-74.

Related Examples

- “Define Code Replacement Mappings”
- “Register Code Replacement Mappings”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code Replacement Match and Replacement Process”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Scalar Operator Code Replacement

This example shows you how to define a code replacement mapping for a scalar operator. The example defines a mapping for the + (addition) operator programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

- 1 Define a function for the definition.

```
function hTable = crl_table_add_uint8
```

- 2 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 3 Create an entry for the operator mapping with a call to the `RTW.Tf1COperationEntry` function.

```
% Create operation entry
op_entry = RTW.Tf1COperationEntry;
```

- 4 Set function entry parameters with a call to the `setTf1COperationEntryParameters` function.

```
% Define addition operation of built-in uint8 data type
% Saturation on, Rounding unspecified
setTf1COperationEntryParameters(op_entry, ...
    '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');
```

- 5 Create conceptual arguments `y1`, `u1`, and `u2`. This example uses calls to the `getTf1ArgFromString` and `addConceptualArg` functions to create and add the arguments.

```
arg = getTf1ArgFromString(hTable, 'y1', 'uint8');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(op_entry, arg);
```

```
arg = getTf1ArgFromString(hTable, 'u1', 'uint8');
addConceptualArg(op_entry, arg);
```

```
arg = getTf1ArgFromString(hTable, 'u2', 'uint8');
addConceptualArg(op_entry, arg);
```

- 6 Copy the conceptual arguments to the implementation arguments. This example uses a call to the `copyConceptualArgsToImplementation` function to create

and add implementation arguments to the entry by copying matching conceptual arguments.

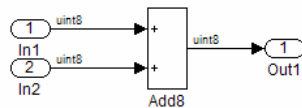
```
copyConceptualArgsToImplementation(op_entry);
```

- 7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

To test this example, create a model that uses an Add block.

- 1 Create a model that includes an Add block, such as this model.



- 2 Configure the model with the following settings:
 - On the **Solver** pane, select a fixed-step solver.
 - On the **Code Generation** pane, select an ERT-based system target file.
 - On the **Code Generation > Interface** pane, select the code replacement library that contains your addition operation entry.
- 3 Generate code and a code generation report.
- 4 Review the code replacements.

Related Examples

- “Define Code Replacement Mappings”
- “Review and Test Code Replacements”
- “Addition and Subtraction Operator Code Replacement”
- “Data Alignment for Code Replacement”
- “Remap Operator Output to Function Input”
- “Customize Matching and Replacement Process for Operators”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “What Is Code Replacement?”

- “What Is Code Replacement Customization?”
- “Code You Can Replace From Simulink Models”
- “Code Replacement Match and Replacement Process”
- “Code Replacement Libraries”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Addition and Subtraction Operator Code Replacement

In this section...

“Algorithm Options” on page 21-152

“Interactive Specification with Code Replacement Tool” on page 21-153

“Programmatic Specification” on page 21-153

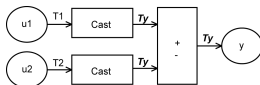
“Algorithm Classification” on page 21-153

“Limitations” on page 21-155

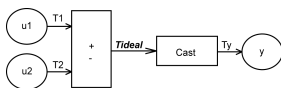
Algorithm Options

When creating a code replacement table entry for an addition or subtraction operator, first determine the type of algorithm that your library function implements.

- **Cast-before-operation (CBO), default** — Prior to performing the addition or subtraction operation, the algorithm type casts input values to the output type. If the output data 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.



- **Cast-after-operation (CAO)** — The algorithm computes the ideal result of the addition or subtraction operation of the two inputs. The algorithm then type casts 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 operand 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.



Interactive Specification with Code Replacement Tool

When you use the Code Replacement Tool to create a code replacement table entry for an addition or subtraction operation, the tool displays an **Algorithm** menu. Use that menu to specify the `Cast before operation` or `Cast after operation` algorithm for that entry.

Programmatic Specification

Create a code replacement table file, as a MATLAB function, that describes the addition or subtraction code replacement table entry. In the call to `setTf1COperationEntryParameters`, set at least these parameters:

- `Key` to `RTW_OP_ADD` or `RTW_OP_MINUS`
- `ImplementationName` to the name of your replacement function
- `EntryInfoAlgorithm` to `RTW_CAST_BFORE_OP` (cast-before-operation) or `RTW_CAST_AFTER_OP` (cast-after-operation)

This example sets parameters for a code replacement operator entry for a cast-after-operation implementation of a `uint8` addition.

```
op_entry = RTW.Tf1COperationEntry;
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_ADD', ...
    'EntryInfoAlgorithm', 'RTW_CAST_AFTER_OP', ...
    'ImplementationName', 'u8_add_u8_u8');
```

For more information, see `setTf1COperationEntryParameters`.

Algorithm Classification

During code generation, the code generator examines addition and subtraction operations, including adjacent type cast operations, to determine the type of algorithm to compute the expression result. Based on the data types in the expression and the type of the accumulator (type used to hold the result of the addition or subtraction operation), the code generator uses these rules.

- Floating-point types only

Input 1 Data Type	Input 2 Data Type	Accumulator Data Type	Output Data Type	Classification
double	double	double	double	CBO, CAO

Input 1 Data Type	Input 2 Data Type	Accumulator Data Type	Output Data Type	Classification
double	double	double	single	—
double	double	single	double	—
double	double	single	single	CBO
double	single	double	double	CBO, CAO
double	single	double	single	—
double	single	single	double	—
double	single	single	single	CBO
single	single	single	single	CBO, CAO
single	single	single	double	—
single	single	double	single	—
single	single	double	double	CBO, CAO

- Floating-point and fixed-point types on the immediate addition or subtraction operation

Algorithm	Conditions
CBO	One of the following is true: <ul style="list-style-type: none"> • Operation type is double. • Operation type is single and input types are single or fixed-point.
CAO	Operation type is a superset of input types—that is, output type can represent values of input types without loss of data.

- Fixed-point types only

Algorithm	Conditions
CBO	At least one of the following is true: <ul style="list-style-type: none"> • Accumlator type equals output type ($T_{acc} == T_{out}$). • Output type is a superset of input types ($T_{acc} \geq \{T_{in1}, T_{in2}\}$) and accumulator type is a superset of output type ($T_{acc} \geq T_{out}$). • Operation does not incur range or precision loss.

Algorithm	Conditions
CAO	Net bias is zero and the data types in the expression have equal slope adjustment factors. For more information on net bias, see “Addition” or “Subtraction” in “Fixed-Point Operator Code Replacement” (for MATLAB code) or “Fixed-Point Operator Code Replacement” (for Simulink models).

In many cases, the numerical result of a CBO operation is equal to that of a CAO operation. For example, if the input and output types are such that the operation produces the ideal result, as in the case of `int8 + int8 -> int16`. To maximize the probability of code replacement occurring in such cases, set the algorithm to cast-after-operation.

Limitations

- The code generator does not replace operations with nonzero net bias.
- When classifying an operation as a CAO operation, the code generator includes the adjacent casts in the expression when the expression involves only fixed-point types. Otherwise, the code generator classifies and replaces only the immediate addition or subtraction operation. Casts that the code generator excludes from the classification appear in the generated code.
- To enable the code generator to include multiple cast operations, which follow an addition or subtraction of fixed-point data, in the classification of an expression, the rounding mode must be `simplest` or `floor`. Consider the expression `y=(cast A) (cast B) (u1+u2)`. If the rounding mode of `(cast A)`, `(cast B)`, and the addition operator (+) are set to `simplest` or `floor`, the code generator takes into account `(cast A)` and `(cast B)` when classifying the expression and performing the replacement only.

Related Examples

- “Define Code Replacement Mappings”
- “Fixed-Point Operator Code Replacement”
- “Data Alignment for Code Replacement”
- “Remap Operator Output to Function Input”
- “Customize Matching and Replacement Process for Operators”

- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code You Can Replace From Simulink Models”
- “Code Replacement Match and Replacement Process”
- “Code Replacement Libraries”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”
- `rtwdemo_crl_cbo_cao`

Small Matrix Operation to Processor Code Replacement

This example shows how to define code replacement mappings that replace nonscalar small matrix operations with processor-specific intrinsic functions. The example defines a table containing two matrix operator replacement entries for the + (addition) operator and the `double` data type. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

- 1 Define a function for the definition.

```
function hTable = crl_table_matrix_add_double
```

- 2 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 3 Create the entry for the first operator mapping with a call to the `RTW.Tf1COperationEntry` function.

```
% Create table entry for matrix_sum_2x2_double
op_entry = RTW.Tf1COperationEntry;
```

- 4 Set operator entry parameters with a call to the `setTf1COperationEntryParameters` function. The code generator ignores saturation and rounding modes for floating-point nonscalar addition and subtraction. For code replacement entries for nonscalar addition and subtraction operations that do not involve fixed-point data, in the call to `setTf1COperationEntryParameters`, specify `'RTW_SATURATE_UNSPECIFIED'` for the `SaturationMode` property and `{'RTW_ROUND_UNSPECIFIED'}` for `RoundingModes`.

```
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_ADD', ...
    'Priority', 30, ...
    'SaturationMode', 'RTW_WRAP_ON_OVERFLOW', ...
    'ImplementationName', 'matrix_sum_2x2_double', ...
    'ImplementationHeaderFile', 'MatrixMath.h', ...
    'ImplementationSourceFile', 'MatrixMath.c', ...
    'ImplementationHeaderPath', LibPath, ...
    'ImplementationSourcePath', LibPath, ...
    'AdditionalIncludePaths', {LibPath}, ...
    'GenCallback', 'RTW.copyFileToBuildDir', ...
    'SideEffects', true);
```

- 5 Create conceptual arguments `y1`, `u1`, and `u2`. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. To specify a matrix argument in the function call, use the argument

class `RTW.Tf1ArgMatrix`. Specify the base type and the dimensions for which the argument is valid. The first table entry specifies [2 2] and the second table entry specifies [3 3].

```
% Specify operands and result
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name', 'y1', ...
    'IOType', 'RTW_IO_OUTPUT', ...
    'BaseType', 'double', ...
    'DimRange', [2 2]);
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name', 'u1', ...
    'BaseType', 'double', ...
    'DimRange', [2 2]);
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name', 'u2', ...
    'BaseType', 'double', ...
    'DimRange', [2 2]);
```

- 6 Create the implementation arguments. This example uses calls to the `getTf1ArgFromString` to create the arguments. The convenience methods `setReturn` and `addArgument` specify whether an argument is a return value or argument and adds the argument to the entry's array of implementation arguments.

```
arg = getTf1ArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.setReturn(arg);

arg = getTf1ArgFromString(hTable, 'u1', ['double' '*']);
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'u2', ['double' '*']);
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'y1', ['double' '*']);
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.addArgument(arg);
```

- 7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

- 8 Create the entry for the second operator mapping.

```
% Create table entry for matrix_sum_3x3_double
op_entry = RTW.Tf1COperationEntry;
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_ADD', ...
    'Priority', 30, ...
    'SaturationMode', 'RTW_WRAP_ON_OVERFLOW', ...
    'ImplementationName', 'matrix_sum_3x3_double', ...
    'ImplementationHeaderFile', 'MatrixMath.h', ...
```



```

    'ImplementationSourceFile', 'MatrixMath.c', ...
    'ImplementationHeaderPath', LibPath, ...
    'ImplementationSourcePath', LibPath, ...
    'AdditionalIncludePaths', {LibPath}, ...
    'GenCallback', 'RTW.copyFileToBuildDir', ...
    'SideEffects', true);

% Specify operands and result
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name', 'y1', ...
    'IOType', 'RTW_IO_OUTPUT', ...
    'BaseType', 'double', ...
    'DimRange', [3 3]);
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name', 'u1', ...
    'BaseType', 'double', ...
    'DimRange', [3 3]);
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name', 'u2', ...
    'BaseType', 'double', ...
    'DimRange', [3 3]);

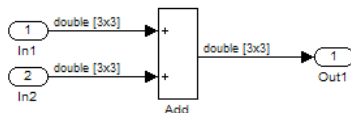
% Specify replacement function signature
arg = getTf1ArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.setReturn(arg);
arg = getTf1ArgFromString(hTable, 'u1', ['double' '*' ]);
op_entry.Implementation.addArgument(arg);
arg = getTf1ArgFromString(hTable, 'u2', ['double' '*' ]);
op_entry.Implementation.addArgument(arg);
arg = getTf1ArgFromString(hTable, 'y1', ['double' '*' ]);
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.addArgument(arg);

addEntry(hTable, op_entry);

```

To test this example:

- 1 Create a model that includes an Add block.



- 2 Configure the model with the following settings:
 - On the **Solver** pane, select a fixed-step, discrete solver with a fixed-step size such as 0.1.
 - On the **Code Generation** pane, select an ERT-based system target file.
 - On the **Code Generation > Interface** pane, select the code replacement library that contains your addition operation entry.
- 3 In the Model Explorer, configure the **Signal Attributes** for the In1 and In2 source blocks. For each source block, set **Port dimensions** to [3,3], and set **Data type** to **double**. Apply the changes. Save the model.
- 4 Generate code and a code generation report.
- 5 Review the code replacements. The code generator replaces the + operator with `matrix_sum_3x3_double` in the generated code.
- 6 Reconfigure **Port dimensions** for In1 and In2 to [2 2], regenerate code. Observe that code containing the + operator is replaced with `matrix_sum_2x2_double`.

Related Examples

- “Define Code Replacement Mappings”
- “Review and Test Code Replacements”
- “Matrix Multiplication Operation to MathWorks BLAS Code Replacement”
- “Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement”
- “Data Alignment for Code Replacement”
- “Remap Operator Output to Function Input”
- “Customize Matching and Replacement Process for Operators”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code You Can Replace From Simulink Models”
- “Code Replacement Match and Replacement Process”

- “Code Replacement Libraries”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Matrix Multiplication Operation to MathWorks BLAS Code Replacement

This example shows how to define code replacement mappings that replace nonscalar multiplication operations with Basic Linear Algebra Subroutine (BLAS) multiplication functions `xgemm` and `xgemv`. The example defines code replacement entries that map floating-point matrix/matrix and matrix/vector multiplication operations to MathWorks BLAS library multiplication functions `dgemm` and `dgemv`. The example defines the function mappings programmatically. Alternatively, you can use the Code Replacement Tool to define the same mappings.

BLAS libraries support matrix/matrix multiplication in the form of

$$C = a(\text{op}(A) * \text{op}(B)) + bC. \text{ op}(X) \text{ means } X, \text{ transposition of } X, \text{ or Hermitian}$$

transposition of X . However, code replacement libraries support only the limited case of

$$C = \text{op}(A) * \text{op}(B) \text{ (} a = 1.0, b = 0.0 \text{)}. \text{ Correspondingly, although BLAS libraries support}$$

matrix/vector multiplication in the form of $y = a(\text{op}(A) * x) + by$, code replacement

libraries support only the limited case of $y = \text{op}(A) * x$ ($a = 1.0, b = 0.0$).

- 1 Define a function for the definition.

```
function hTable = crl_table_tmwblas_mmult_double
```

- 2 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 3 Define the path for the BLAS function library. If your replacement functions are on the MATLAB search path or are in your working folder, you can skip this step.

```
% Define library path for Windows or UNIX
arch = computer('arch');
if ~ispc
    LibPath = fullfile('$MATLAB_ROOT', 'bin', arch);
else
    % Use Stateflow to get the compiler info
    compilerInfo = sf('Private','compilerman','get_compiler_info');
    compilerName = compilerInfo.compilerName;
    if strcmp(compilerName, 'msvc90') || ...
        strcmp(compilerName, 'msvc80') || ...
        strcmp(compilerName, 'msvc71') || ...
        strcmp(compilerName, 'msvc60'), ...
        compilerName = 'microsoft';
    end
    LibPath = fullfile('$MATLAB_ROOT', 'extern', 'lib', arch, compilerName);
end
```

- 4 Create an entry for the first mapping with a call to the `RTW.Tf1BlasEntryGenerator` function.

```
% Create table entry for dgemm32
op_entry = RTW.Tf1BlasEntryGenerator;
```

- 5 Set operator entry parameters with a call to the `setTf1CFunctionEntryParameters` function. The function call sets matrix multiplication operator entry properties. The code generator ignores saturation and rounding modes for floating-point nonscalar addition and subtraction. For code replacement entries for nonscalar addition and subtraction operations that do not involve fixed-point data, in the call to `setTf1CFunctionEntryParameters`, specify `'RTW_SATURATE_UNSPECIFIED'` for the `SaturationMode` property and `{'RTW_ROUND_UNSPECIFIED'}` for `RoundingModes`.

```
if ispc
    libExt = 'lib';
elseif ismac
    libExt = 'dylib';
else
    libExt = 'so';
end
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_MUL', ...
    'Priority', 100, ...
    'ImplementationName', 'dgemm32', ...
    'ImplementationHeaderFile', 'blascompat32.h', ...
    'ImplementationHeaderPath', fullfile('${MATLAB_ROOT}','extern','include'), ...
    'AdditionalLinkObjs', {'libmwblascompat32.' libExt}, ...
    'AdditionalLinkObjsPaths', {LibPath}, ...
    'SideEffects', true);
```

- 6 Create conceptual arguments `y1`, `u1`, and `u2`. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. To specify a matrix argument in the function call, use the argument class `RTW.Tf1ArgMatrix` and specify the base type and the dimensions for which the argument is valid. This type of table entry supports 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. The conceptual output argument for the `dgemm32` entry for matrix/matrix multiplication replacement specifies dimensions `[2 2; inf inf]`, while the conceptual output argument for the `dgemv32` entry for matrix/vector multiplication replacement specifies dimensions `[2 1; inf 1]`.

```
% Specify operands and result
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name', 'y1', ...
    'IOType', 'RTW_IO_OUTPUT', ...
```

```

        'BaseType',    'double', ...
        'DimRange',   [2 2; inf inf]);
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
        'Name',       'u1', ...
        'BaseType',   'double', ...
        'DimRange',   [2 2; inf inf]);
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
        'Name',       'u2', ...
        'BaseType',   'double', ...
        'DimRange',   [1 1; inf inf]);

```

- 7 Create the implementation arguments. This example uses calls to the `getTf1ArgFromString` and `RTW.Tf1ArgCharConstant` functions to create the arguments. The example code configures special implementation arguments that are required for `dgemm` and `dgemv` function replacements. The convenience methods `setReturn` and `addArgument` specify whether an argument is a return value or argument and adds the argument to the entry's array of implementation arguments.

```

% Using RTW.Tf1BlasEntryGenerator for xgemm requires the following
% implementation signature:
%
% void f(char* TRANSA, char* TRANSB, int* M, int* N, int* K,
%        type* ALPHA, type* u1, int* LDA, type* u2, int* LDB,
%        type* BETA, type* y, int* LDC)
%
% When a match occurs, the code generator computes the
% values for M, N, K, LDA, LDB, and LDC and inserts them into the
% generated code. TRANSA and TRANSB are set to 'N'.

% Specify replacement function signature

arg = getTf1ArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.setReturn(arg);

arg = RTW.Tf1ArgCharConstant('TRANSA');
% Possible values for PassByType property are
% RTW_PASSBY_AUTO, RTW_PASSBY_POINTER,
% RTW_PASSBY_VOID_POINTER, RTW_PASSBY_BASE_POINTER
arg.PassByType = 'RTW_PASSBY_POINTER';
op_entry.Implementation.addArgument(arg);

arg = RTW.Tf1ArgCharConstant('TRANSB');
arg.PassByType = 'RTW_PASSBY_POINTER';
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'M', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'N', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;

```

```

op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'K', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'ALPHA', 'double', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'u1', ['double' '*']);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'LDA', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'u2', ['double' '*']);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'LDB', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'BETA', 'double', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'y1', ['double' '*']);
arg.IOType = 'RTW_IO_OUTPUT';
arg.PassByType = 'RTW_PASSBY_POINTER';
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'LDC', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

```

- 8 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

- 9 Create the entry for the second mapping.

```

% Create table entry for dgemv32
op_entry = RTW.Tf1BlasEntryGenerator;
if ispc

```

```

    libExt = 'lib';
elseif ismac
    libExt = 'dylib';
else
    libExt = 'so';
end
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_MUL', ...
    'Priority', 100, ...
    'ImplementationName', 'dgemv32', ...
    'ImplementationHeaderFile', 'blascompat32.h', ...
    'ImplementationHeaderPath', fullfile('${MATLAB_ROOT}','extern','include'), ...
    'AdditionalLinkObjs', {'libmwblascompat32.' libExt}], ...
    'AdditionalLinkObjsPaths', {LibPath},...
    'SideEffects', true);

% Specify operands and result
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name', 'y1', ...
    'IOType', 'RTW_IO_OUTPUT', ...
    'BaseType', 'double', ...
    'DimRange', [2 1; inf 1]);
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name', 'u1', ...
    'BaseType', 'double', ...
    'DimRange', [2 2; inf inf]);
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix',...
    'Name', 'u2', ...
    'BaseType', 'double', ...
    'DimRange', [1 1; inf 1]);

% Using RTW.Tf1BlasEntryGenerator for xgemv requires the following
% implementation signature:
%
% void f(char* TRANS, int* M, int* N,
%       type* ALPHA, type* u1, int* LDA, type* u2, int* INCX,
%       type* BETA, type* y, int* INCY)
%
% Upon a match, the CRL entry will compute the
% values for M, N, LDA, INCX, and INCY, and insert them into the
% generated code. TRANS will be set to 'N'.

% Specify replacement function signature

arg = getTf1ArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.setReturn(arg);

arg = RTW.Tf1ArgCharConstant('TRANS');
arg.PassByType = 'RTW_PASSBY_POINTER';
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'M', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;

```



```
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'N', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'ALPHA', 'double', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'u1', ['double' '*']);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'LDA', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'u2', ['double' '*']);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'INCX', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'BETA', 'double', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

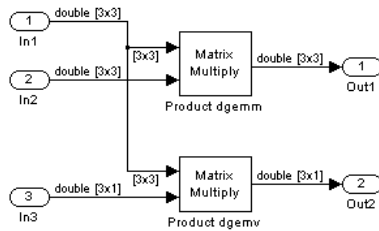
arg = getTf1ArgFromString(hTable, 'y1', ['double' '*']);
arg.IOType = 'RTW_IO_OUTPUT';
arg.PassByType = 'RTW_PASSBY_POINTER';
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'INCY', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

addEntry(hTable, op_entry);
```

To test this example:

- 1 Create a model that includes two Product blocks.



- 2 For each Product block, set the block parameter **Multiplication** to the value `Matrix(*)`.
- 3 Configure the model with the following settings:
 - On the **Solver** pane, select a fixed-step, discrete solver with a fixed-step size such as 0.1.
 - On the **Code Generation** pane, select an ERT-based system target file.
 - On the **Code Generation > Interface** pane, select the code replacement library that contains your addition operation entry.
- 4 In the Model Explorer, configure the **Signal Attributes** for the In1, In2, and In3 source blocks. For In1 and In2, set **Port dimensions** to [3 3] and set the **Data type** to double. For In3, set **Port dimensions** to [3 1] and set the **Data type** to double.
- 5 Generate code and a code generation report.
- 6 Review the code replacements.

Related Examples

- “Define Code Replacement Mappings”
- “Review and Test Code Replacements”
- “Small Matrix Operation to Processor Code Replacement”
- “Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement”
- “Data Alignment for Code Replacement”
- “Remap Operator Output to Function Input”
- “Customize Matching and Replacement Process for Operators”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code You Can Replace From Simulink Models”
- “Code Replacement Match and Replacement Process”
- “Code Replacement Libraries”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement

This example shows how to define code replacement mappings that replace nonscalar multiplication operations with ANSI/ISO[®] C BLAS multiplication functions `xgemm` and `xgemv`. The example defines code replacement entries that map floating-point matrix/matrix and matrix/vector multiplication operations to ANSI/ISO C BLAS library multiplication functions `dgemm` and `dgemv`. The example defines the function mappings programmatically. Alternatively, you can use the Code Replacement Tool to define the same mappings.

BLAS libraries support matrix/matrix multiplication in the form of $C = a(\text{op}(A) * \text{op}(B)) + bC$. `op(X)` means X, transposition of X, or Hermitian transposition of X. However, code replacement libraries support only the limited case of $C = \text{op}(A) * \text{op}(B)$ ($a = 1.0, b = 0.0$). Correspondingly, although BLAS libraries support matrix/vector multiplication in the form of $y = a(\text{op}(A) * x) + by$, code replacement libraries support only the limited case of $y = \text{op}(A) * x$ ($a = 1.0, b = 0.0$).

- 1 Define a function for the definition.

```
function hTable = crl_table_cblas_mmult_double
```

- 2 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 3 Define the path for the CBLAS function library. For example:

```
LibPath = fullfile(matlabroot, 'toolbox', 'rtw', 'rtwdemos', 'crl_demo');
```

- 4 Create an entry for the first mapping with a call to the `RTW.Tf1BlasEntryGenerator` function.

```
% Create table entry for cblas_dgemm
op_entry = RTW.Tf1CblasEntryGenerator;
```

- 5 Set operator entry parameters with a call to the `setTf1COperationEntryParameters` function. The function call sets matrix multiplication operator entry properties. The code generator ignores saturation and rounding modes for floating-point nonscalar addition and subtraction.

```
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_MUL', ...
```

```

'Priority',          100, ...
'ImplementationName', 'cblas_dgemm', ...
'ImplementationHeaderFile', 'cblas.h', ...
'ImplementationHeaderPath', LibPath, ...
'AdditionalIncludePaths', {LibPath}, ...
'GenCallback',      'RTW.copyFileToBuildDir', ...
'SideEffects',      true);

```

- 6 Create conceptual arguments `y1`, `u1`, and `u2`. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. To specify a matrix argument in the function call, use the argument class `RTW.Tf1ArgMatrix` and specify the base type and the dimensions for which the argument is valid. This type of table entry supports 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. The conceptual output argument for the `dgemm32` entry for matrix/matrix multiplication replacement specifies dimensions `[2 2; inf inf]`. The conceptual output argument for the `dgemv32` entry for matrix/vector multiplication replacement specifies dimensions `[2 1; inf 1]`.

```

% Specify operands and result
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name', 'y1', ...
    'IOType', 'RTW_IO_OUTPUT', ...
    'BaseType', 'double', ...
    'DimRange', [2 2; inf inf]);
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name', 'u1', ...
    'BaseType', 'double', ...
    'DimRange', [2 2; inf inf]);
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name', 'u2', ...
    'BaseType', 'double', ...
    'DimRange', [1 1; inf inf]);

```

- 7 Create the implementation arguments. This example uses calls to the `getTf1ArgFromString` function to create the arguments. The example code configures special implementation arguments that are required for `dgemm` and `dgemv` function replacements. The convenience methods `setReturn` and `addArgument` specify whether an argument is a return value or argument and adds the argument to the entry's array of implementation arguments.

```

% Using RTW.Tf1CBlasEntryGenerator for xgemm requires the following
% implementation signature:
%
% void f(enum ORDER, enum TRANSA, enum TRANSB, int M, int N, int K,
%         type ALPHA, type* u1, int LDA, type* u2, int LDB,
%         type BETA, type* y, int LDC)
%
% Since CRLs do not have the ability to specify enums, you must

```

```
% use integer. (This will cause problems with C++ code generation,  
% so for C++, use a wrapper function to cast each int to the  
% corresponding enumeration type.)  
%  
% When a match occurs, the code generator computes the  
% values for M, N, K, LDA, LDB, and LDC and insert them into the  
% generated code.  
  
% Specify replacement function signature  
  
arg = getTf1ArgFromString(hTable, 'y2', 'void');  
arg.IOType = 'RTW_IO_OUTPUT';  
op_entry.Implementation.setReturn(arg);  
  
arg = getTf1ArgFromString(hTable, 'ORDER', 'integer', 102);  
%arg.Type.ReadOnly=true;  
op_entry.Implementation.addArgument(arg);  
  
arg = getTf1ArgFromString(hTable, 'TRANSA', 'integer', 111);  
%arg.Type.ReadOnly=true;  
op_entry.Implementation.addArgument(arg);  
  
arg = getTf1ArgFromString(hTable, 'TRANSB', 'integer', 111);  
%arg.Type.ReadOnly=true;  
op_entry.Implementation.addArgument(arg);  
  
arg = getTf1ArgFromString(hTable, 'M', 'integer', 0);  
op_entry.Implementation.addArgument(arg);  
  
arg = getTf1ArgFromString(hTable, 'N', 'integer', 0);  
op_entry.Implementation.addArgument(arg);  
  
arg = getTf1ArgFromString(hTable, 'K', 'integer', 0);  
op_entry.Implementation.addArgument(arg);  
  
arg = getTf1ArgFromString(hTable, 'ALPHA', 'double', 1);  
op_entry.Implementation.addArgument(arg);  
  
arg = getTf1ArgFromString(hTable, 'u1', ['double' '*']);  
op_entry.Implementation.addArgument(arg);  
  
arg = getTf1ArgFromString(hTable, 'LDA', 'integer', 0);  
op_entry.Implementation.addArgument(arg);  
  
arg = getTf1ArgFromString(hTable, 'u2', ['double' '*']);  
op_entry.Implementation.addArgument(arg);  
  
arg = getTf1ArgFromString(hTable, 'LDB', 'integer', 0);  
op_entry.Implementation.addArgument(arg);  
  
arg = getTf1ArgFromString(hTable, 'BETA', 'double', 0);  
op_entry.Implementation.addArgument(arg);  
  
arg = getTf1ArgFromString(hTable, 'y1', ['double' '*']);  
arg.IOType = 'RTW_IO_OUTPUT';
```

```

op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'LDC', 'integer', 0);
op_entry.Implementation.addArgument(arg);

```

- 8 Add the entry to a code replacement table with a call to the addEntryfunction.

```
addEntry(hTable, op_entry);
```

- 9 Create the entry for the second mapping.

```

% Create table entry for cblas_dgemv
op_entry = RTW.Tf1CBlasEntryGenerator;
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW.OP_MUL', ...
    'Priority', 100, ...
    'ImplementationName', 'cblas_dgemv', ...
    'ImplementationHeaderFile', 'cblas.h', ...
    'ImplementationHeaderPath', LibPath, ...
    'AdditionalIncludePaths', {LibPath}, ...
    'GenCallback', 'RTW.copyFileToBuildDir', ...
    'SideEffects', true);

% Specify operands and result
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name', 'y1', ...
    'IOType', 'RTW_IO_OUTPUT', ...
    'BaseType', 'double', ...
    'DimRange', [2 1; inf 1]);
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name', 'u1', ...
    'BaseType', 'double', ...
    'DimRange', [2 2; inf inf]);
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name', 'u2', ...
    'BaseType', 'double', ...
    'DimRange', [1 1; inf 1]);

% Using RTW.Tf1CBlasEntryGenerator for xgemv requires the following
% implementation signature:
%
% void f(enum ORDER, enum TRANSA, int M, int N,
%       type ALPHA, type* u1, int LDA, type* u2, int INCX,
%       type BETA, type* y, int INCY)
%
% Since CRLs do not have the ability to specify enums, you must
% use integer. (This will cause problems with C++ code generation,
% so for C++, use a wrapper function to cast each int to the
% corresponding enumeration type.)
%
% Upon a match, the CRL entry will compute the
% values for M, N, LDA, INCX, and INCY and insert them into the
% generated code.

% Specify replacement function signature

```

```
arg = getTf1ArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.setReturn(arg);

arg = getTf1ArgFromString(hTable, 'ORDER', 'integer', 102);
%arg.Type.ReadOnly=true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'TRANSA', 'integer', 111);
%arg.Type.ReadOnly=true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'M', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'N', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'ALPHA', 'double', 1);
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'u1', ['double' '*']);
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'LDA', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'u2', ['double' '*']);
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'INCX', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'BETA', 'double', 0);
op_entry.Implementation.addArgument(arg);

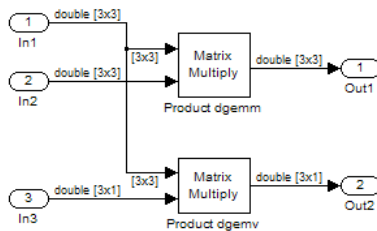
arg = getTf1ArgFromString(hTable, 'y1', ['double' '*']);
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'INCY', 'integer', 0);
op_entry.Implementation.addArgument(arg);

addEntry(hTable, op_entry);
```

To test this example:

- 1 Create a model that includes two Product blocks.



- 2 Configure the model with the following settings:
 - On the **Solver** pane, select a fixed-step, discrete solver with a fixed-step size such as 0.1.
 - On the **Code Generation** pane, select an ERT-based system target file.
 - On the **Code Generation > Interface** pane, select the code replacement library that contains your addition operation entry.
- 3 For each Product block, set the block parameter **Multiplication** to the value `Matrix(*)`.
- 4 In the Model Explorer, configure the **Signal Attributes** for the In1, In2, and In3 source blocks. For In1 and In2, set **Port dimensions** to [3 3]. Set the **Data type** to double. For In3, set **Port dimensions** to [3 1]. Set the **Data type** to double.
- 5 Generate code and a code generation report.
- 6 Review the code replacements.

Related Examples

- “Define Code Replacement Mappings”
- “Review and Test Code Replacements”
- “Small Matrix Operation to Processor Code Replacement”
- “Matrix Multiplication Operation to MathWorks BLAS Code Replacement”
- “Data Alignment for Code Replacement”
- “Remap Operator Output to Function Input”
- “Customize Matching and Replacement Process for Operators”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code You Can Replace From Simulink Models”
- “Code Replacement Match and Replacement Process”
- “Code Replacement Libraries”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Remap Operator Output to Function Input

If your generated code must meet a specific coding pattern or you want more flexibility, for example, to further improve performance, you can remap operator outputs to input positions in an implementation function argument list.

Note: Remapping outputs to implementation function inputs is supported only for operator replacement.

For example, for a sum operation, the code generator produces code similar to:

```
add8_Y.Out1 = u8_add_u8_u8(add8_U.In1, add8_U.In2);
```

If you remap the output to the first input, the code generator produces code similar to:

```
u8_add_u8_u8(&add8_Y.Out1;, add8_U.In1, add8_U.In2);
```

The following table definition file for a sum operation remaps operator output `y1` as the first function input argument.

- 1 Define a function for the definition.

```
function hTable = crl_table_add_uint8
```

- 2 In the call to the `setTf1COperationEntryParameters` function for the operator replacement, set the `SideEffects` property to `true`.

- 3 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 4 Create an entry for the operator mapping with a call to the `RTW.Tf1COperationEntry` function.

```
% Create operation entry
op_entry = RTW.Tf1COperationEntry;
```

- 5 Set function entry parameters with a call to the `setTf1CFunctionEntryParameters` function. In the function call, set the property `SideEffects` to `true`.

```
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_ADD', ...
    'Priority', 90, ...
    'ImplementationName', 'u8_add_u8_u8', ...
```

```

        'ImplementationHeaderFile', 'u8_add_u8_u8.h', ...
        'ImplementationSourceFile', 'u8_add_u8_u8.c', ...
        'SideEffects', true );

```

- 6 Create conceptual arguments y1, u1, and u2. This example uses calls to the `getTf1ArgFromString` and `addConceptualArg` functions to create and add the arguments.

```

arg = getTf1ArgFromString(hTable, 'y1', 'uint8');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(op_entry, arg);

arg = getTf1ArgFromString(hTable, 'u1', 'uint8');
addConceptualArg(op_entry, arg );

arg = getTf1ArgFromString(hTable, 'u2', 'uint8');
addConceptualArg(op_entry, arg );

```

- 7 Create the implementation arguments. This example uses calls to the `getTf1ArgFromString` function to create the arguments. When defining the implementation function return argument, create a new `void` output argument, for example, y2. When defining the implementation function argument for the conceptual output argument (y1), set the operator output argument as an additional input argument. Mark its `IOType` as output. Make its type a pointer type. The convenience methods `setReturn` and `addArgument` specify whether an argument is a return value or argument and adds the argument to the entry's array of implementation arguments.

```

% Create new void output y2
arg = getTf1ArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.setReturn(arg);

% Set y1 as first input arg, mark IOType as output, and use pointer type
arg=getTf1ArgFromString(hTable, 'y1', 'uint8*');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.addArgument(arg);

arg=getTf1ArgFromString(hTable, 'u1', 'uint8');
op_entry.Implementation.addArgument(arg);

arg=getTf1ArgFromString(hTable, 'u2', 'uint8');
op_entry.Implementation.addArgument(arg);

```

- 8 Add the entry to a code replacement table with a call to the `addEntry` function.

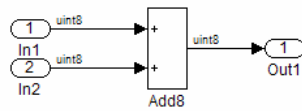
```

addEntry(hTable, op_entry);

```

To test this example:

- 1 Create a model that includes an Add block.



- 2 Configure the model with the following settings:
 - On the **Solver** pane, select a fixed-step solver.
 - On the **Code Generation** pane, select an ERT-based system target file.
 - On the **Code Generation > Interface** pane, select the code replacement library that contains your addition operation entry.
 - On the **Optimization** pane, set **Signals and Parameters > Optimize global data access** to Use global to hold temporary results to reduce data copies in the generated code.
- 3 Generate code and a code generation report.
- 4 Review the code replacements.

Related Examples

- “Define Code Replacement Mappings”
- “Review and Test Code Replacements”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code You Can Replace From Simulink Models”
- “Code Replacement Libraries”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Customize Matching and Replacement Process for Operators

Create the Entry

This example shows how to create custom code replacement entries that add extra logic to the code replacement matching and replacement process. Custom entries allow you to specify additional match criteria or modify the replacement function signature to meet your application needs.

- You might want to replace an operator with a particular fixed-point implementation function only when fraction lengths are within a particular range.
- When a match occurs, you might want to modify your replacement function signature based on compile-time information, such as passing fraction-length values into the function.

The example modifies a fixed-point addition replacement such that the implementation function passes in the fraction lengths of the input and output data types as arguments.

Create Class Folder for Entry

Create a class folder using the name of your derived class, such as `@Tf1CustomOperationEntry`. Verify that the class folder is on the MATLAB search path or in your current working folder.

Create Derived Class that Defines `do_match` Method

In the class folder, create and save the following class definition file, `Tf1CustomOperationEntry.m`. This file defines the class `Tf1CustomOperationEntry`, which is derived from the base class `RTW.Tf1COperationEntryML`.

The derived class defines a `do_match` method. In the `do_match` signature:

- `ent` is the return handle, which is returned either as empty (indicating that the match failed) or as a `Tf1COperationEntry` handle.
- `hThis` is the handle to this object.
- `hCSO` is a handle to an object created by the code generator for querying the library for a replacement.
- The remaining arguments are the number of bits for various data types of the current target.

The `do_match` method adds required additional match criteria that the base class does not provide. The method makes required modifications to the implementation signature. In this case, the `do_match` method can rely on the base class for checking word size and signedness. `do_match` must match only the number of conceptual arguments to the value 3 (two inputs and one output) and the bias for each argument to the value 0. If the code generator finds a match, `do_match` sets the return handle, removes slope and bias wildcards from the conceptual arguments (the match is for specific slope and bias values), and writes fraction-length values for the inputs and output into replacement function arguments 3, 4, and 5.

You can create and add the three additional implementation function arguments for passing fraction lengths in the class definition or in each code replacement entry definition that instantiates this class. This example creates the arguments, adds them to a code replacement table definition file, and sets them to specific values in the class definition code.

```
classdef TflCustomOperationEntry < RTW.TflCOperationEntryML
    methods
        function ent = do_match(hThis, ...
            hCSO, ... %#ok
            targetBitPerChar, ... %#ok
            targetBitPerShort, ... %#ok
            targetBitPerInt, ... %#ok
            targetBitPerLong) %#ok
            % DO_MATCH - Create a custom match function. The base class
            % checks the types of the arguments prior to calling this
            % method. This will check additional data and perhaps modify
            % the implementation function.

            % The base class checks word size and signedness. Slopes and biases
            % have been wildcarded, so the only additional checking to do is
            % to check that the biases are zero and that there are only three
            % conceptual arguments (one output, two inputs)

            ent = []; % default the return to empty, indicating the match failed

            if length(hCSO.ConceptualArgs) == 3 && ...
                hCSO.ConceptualArgs(1).Type.Bias == 0 && ...
                hCSO.ConceptualArgs(2).Type.Bias == 0 && ...
                hCSO.ConceptualArgs(3).Type.Bias == 0

                % Modify the default implementation. Since this is a
                % generator entry, a concrete entry is created using this entry
                % as a template. The type of entry being created is a standard
                % TflCOperationEntry. Using the standard operation entry
                % provides required information, and you do not need
                % a custom match function.
                ent = RTW.TflCOperationEntry(hThis);

                % Since this entry is modifying the implementation for specific
```

```
% fraction-length values (arguments 3, 4, and 5), the conceptual argument
% wildcards must be removed (the wildcards were inherited from the
% generator when it was used as a template for the concrete entry).
% This concrete entry is now for a specific slope and bias.
% hCS0 holds the slope and bias values (created by the code generator).
for idx=1:3
    ent.ConceptualArgs(idx).CheckSlope = true;
    ent.ConceptualArgs(idx).CheckBias = true;

    % Set the specific Slope and Biases
    ent.ConceptualArgs(idx).Type.Slope = hCS0.ConceptualArgs(idx).Type.Slope;
    ent.ConceptualArgs(idx).Type.Bias = 0;
end

% Set the fraction-length values in the implementation function.
ent.Implementation.Arguments(3).Value = ...
    -1.0*hCS0.ConceptualArgs(2).Type.FixedExponent;
ent.Implementation.Arguments(4).Value = ...
    -1.0*hCS0.ConceptualArgs(3).Type.FixedExponent;
ent.Implementation.Arguments(5).Value = ...
    -1.0*hCS0.ConceptualArgs(1).Type.FixedExponent;
end
end
end
end
```

Create Code Replacement Entry

Create code replacement entries that instantiate your custom entry class. For this example, create and save a code replacement table that contains a single operator entry, an entry generator for unsigned 32-bit fixed-point addition operations, with arbitrary fraction-length values on the inputs and the output. This entry instantiates the derived class from the previous step.

If you want to replace all word sizes and signedness attributes (not just 32-bit and unsigned), you can use the same derived class, but not the same entry, because you cannot wildcard the `WordLength` and `ISSigned` arguments. For example, to support `uint8`, `int8`, `uint16`, `int16`, and `int32`, you must add five other distinct entries. Similarly, to use different implementation functions for saturation and rounding modes other than overflow and round to floor, you must add entries for those match permutations.

This table entry creates and adds three implementation arguments to hold the fraction-length values for the inputs and output. Alternatively, the entry can omit those argument definitions. Instead the `do_match` method of the derived class `Tf1CustomOperationEntry` can create and add the three implementation arguments. When the number of additional implementation arguments required can vary based on compile-time information, use the alternative approach.

- 1 In your working folder, create an entry definition file.
- 2 Define a function for the definition.

```
function hTable = crl_table_custom_add_ufix32
```

- 3 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 4 Create an entry for the custom operator mapping with a call to the `RTW.Tf1CustomOperationEntry` function.

```
%% Add Tf1CustomOperationEntry
op_entry = Tf1CustomOperationEntry;
```

- 5 Set function entry parameters with a call to the `setTf1COperationEntryParameters` function.

```
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_ADD', ...
    'Priority', 30, ...
    'SaturationMode', 'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes', {'RTW_ROUND_FLOOR'}, ...
    'ImplementationName', 'myFixptAdd', ...
    'ImplementationHeaderFile', 'myFixptAdd.h', ...
    'ImplementationSourceFile', 'myFixptAdd.c');
```

- 6 Create conceptual arguments `y1`, `u1`, and `u2`. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call.

```
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'y1', ...
    'IOType', 'RTW_IO_OUTPUT', ...
    'CheckSlope', false, ...
    'CheckBias', false, ...
    'DataType', 'Fixed', ...
    'Scaling', 'BinaryPoint', ...
    'IsSigned', false, ...
    'WordLength', 32);
```

```
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'u1', ...
    'IOType', 'RTW_IO_INPUT', ...
    'CheckSlope', false, ...
    'CheckBias', false, ...
    'DataType', 'Fixed', ...
    'Scaling', 'BinaryPoint', ...
    'IsSigned', false, ...
    'WordLength', 32);
```

```
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'u2', ...
```

```

    'IOType',      'RTW_IO_INPUT', ...
    'CheckSlope', false, ...
    'CheckBias',  false, ...
    'DataType',   'Fixed', ...
    'Scaling',    'BinaryPoint', ...
    'IsSigned',   false, ...
    'WordLength', 32);

```

- 7 Create the implementation arguments. This example uses calls to the `createAndSetCImplementationReturn` and `createAndAddImplementationArg` functions to create and add implementation arguments to the entry.

```

% Specify replacement function signature
createAndSetCImplementationReturn(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',      'y1', ...
    'IOType',    'RTW_IO_OUTPUT', ...
    'IsSigned',  false, ...
    'WordLength', 32, ...
    'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',      'u1', ...
    'IOType',    'RTW_IO_INPUT', ...
    'IsSigned',  false, ...
    'WordLength', 32, ...
    'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',      'u2', ...
    'IOType',    'RTW_IO_INPUT', ...
    'IsSigned',  false, ...
    'WordLength', 32, ...
    'FractionLength', 0);

% Add 3 fraction-length args. Actual values are set during code generation.
createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumericConstant', ...
    'Name',      'f1_in1', ...
    'IOType',    'RTW_IO_INPUT', ...
    'IsSigned',  false, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value',     0);

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumericConstant', ...
    'Name',      'f1_in2', ...
    'IOType',    'RTW_IO_INPUT', ...
    'IsSigned',  false, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value',     0);

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumericConstant', ...
    'Name',      'f1_out', ...
    'IOType',    'RTW_IO_INPUT', ...

```

```

    'IsSigned', false, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value', 0);
    
```

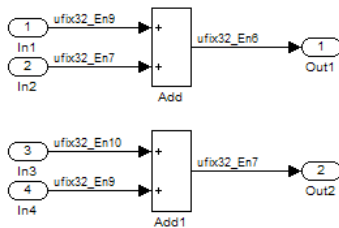
- 8 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

Test the Entry

To test the custom code replacement entry:

- 1 Create a model that includes one or more unsigned 32-bit fixed-point addition operations.



- 2 In the block parameters for the Add blocks, set **Integer rounding mode** to **Floor** and select **Saturate on integer overflow**.
- 3 Configure the model with the following settings:
 - On the **Solver** pane, select a fixed-step, discrete solver.
 - On the **Code Generation** pane, select an ERT-based system target file.
 - On the **Code Generation > Interface** pane, select the code replacement library that contains your custom operation entry.

Apply the changes. Save the model.

- 4 Generate code and a code generation report.
- 5 Review the code replacements. `myFixptAdd` replaces the default implementation code for the unsigned 32-bit fixed-point addition operation. The three additional fraction-length arguments are present.

```

/* Model step function */
void ufix32_add_step(void)
{
    
```

```
/* Output: '<Root>/Out1' incorporates:
 * Inport: '<Root>/In1'
 * Inport: '<Root>/In2'
 * Sum: '<Root>/Add1'
 */
ufix32_add_Y.Out1 = myFixptAdd(ufix32_add_U.In1, ufix32_add_U.In2, 9U, 7U, 6U);

/* Output: '<Root>/Out2' incorporates:
 * Inport: '<Root>/In3'
 * Inport: '<Root>/In4'
 * Sum: '<Root>/Add1'
 */
ufix32_add_Y.Out2 = myFixptAdd(ufix32_add_U.In3, ufix32_add_U.In4, 10U, 9U, 7U);
}
```

Related Examples

- “Define Code Replacement Mappings”
- “Review and Test Code Replacements”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “Code Replacement Match and Replacement Process”
- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code You Can Replace From Simulink Models”
- “Code Replacement Libraries”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Fixed-Point Operator Code Replacement

In this section...

“Fixed-Point Operator Entries” on page 21-187

“Fixed-Point Numbers and Arithmetic” on page 21-191

“Addition” on page 21-191

“Subtraction” on page 21-192

“Multiplication” on page 21-192

“Division” on page 21-193

“Data Type Conversion (Cast)” on page 21-194

“Shift” on page 21-194

Fixed-Point Operator Entries

If you have a Fixed-Point Designer license, you can define fixed-point operator code replacement entries to match:

- A binary-point-only scaling combination on the operator inputs and output.
- A slope bias scaling combination on the operator inputs and output.
- Relative scaling or net slope between multiplication or division operator inputs and output. Use one of these methods to map a range of slope and bias values to a replacement function for multiplication or division.
- Equal slope and zero net bias across addition or subtraction operator inputs and output. Use this method to disregard specific slope and bias values and map relative slope and bias values to a replacement function for addition or subtraction.

The following table maps common ways to match fixed-point operator code replacement entries with the associated fixed-point parameters that you specify in a code replacement table definition file.

Match	Create entry	Minimally specify parameters
A specific binary-point-only scaling combination on the operator inputs and output.	RTW.Tf1COperationEntry	createAndAddConceptualArg function: <ul style="list-style-type: none"> • CheckSlope: Specify the value true.

Match	Create entry	Minimally specify parameters
		<ul style="list-style-type: none"> • CheckBias: Specify the value <code>true</code>. • DataTypeMode (or DataType/Scaling equivalent): Specify fixed-point binary-point-only scaling. • FractionLength: Specify a fraction length (for example, 3).
<p>A specific slope bias scaling combination on the operator inputs and output.</p>	<p>RTW.Tf1COperationEntry</p>	<p>createAndAddConceptualArg function:</p> <ul style="list-style-type: none"> • CheckSlope: Specify the value <code>true</code>. • CheckBias: Specify the value <code>true</code>. • DataTypeMode (or DataType/Scaling equivalent): Specify fixed-point [slope bias] scaling. • Slope (or SlopeAdjustmentFactor/FixedExponent equivalent): Specify a slope value (for example, 15). • Bias: Specify a bias value (for example, 2).

Match	Create entry	Minimally specify parameters
<p>Net slope between operator inputs and output (multiplication and division).</p>	<p>RTW.Tf1COperationEntry-Generator_NetSlope</p>	<p>setTf1COperationEntryParameters function:</p> <ul style="list-style-type: none"> • NetSlopeAdjustmentFactor: Specify the slope adjustment factor (F) part of the net slope, $F2^E$ (for example, 1.0). • NetFixedExponent: Specify the fixed exponent (E) part of the net slope, $F2^E$ (for example, -3.0). <p>createAndAddConceptualArg function:</p> <ul style="list-style-type: none"> • CheckSlope: Specify the value false. • CheckBias: Specify the value false. • DataType: Specify the value 'Fixed'.

Match	Create entry	Minimally specify parameters
<p>Relative scaling between operator inputs and output (multiplication and division).</p>	<p>RTW.Tf1COperationEntry-Generator</p>	<p>setTf1COperationEntryParameters function:</p> <ul style="list-style-type: none"> • RelativeScalingFactorF: Specify the slope adjustment factor (F) part of the relative scaling factor, $F2^E$ (for example, 1.0). • RelativeScalingFactorE: Specify the fixed exponent (E) part of the relative scaling factor, $F2^E$ (for example, -3.0). <p>createAndAddConceptualArg function:</p> <ul style="list-style-type: none"> • CheckSlope: Specify the value false. • CheckBias: Specify the value false. • DataType: Specify the value 'Fixed'.
<p>Equal slope and zero net bias across operator inputs and output (addition and subtraction).</p>	<p>RTW.Tf1COperationEntry-Generator</p>	<p>setTf1COperationEntryParameters function:</p> <ul style="list-style-type: none"> • SlopesMustBeTheSame: Specify the value true. • MustHaveZeroNetBias: Specify the value true. <p>createAndAddConceptualArg function:</p> <ul style="list-style-type: none"> • CheckSlope: Specify the value false. • CheckBias: Specify the value false.

Fixed-Point Numbers and Arithmetic

Fixed-point numbers use integers and integer arithmetic to represent real numbers and arithmetic with the following encoding scheme:

$$V = \tilde{V} = SQ + B$$

- V is an arbitrarily precise real-world value.
- \tilde{V} is the approximate real-world value that results from fixed-point representation.
- Q is an integer that encodes \tilde{V} , referred to as the *quantized integer*.
- S is a coefficient of Q , referred to as the *slope*.
- B is an additive correction, referred to as the *bias*.

The general equation for an operation between fixed-point operands is:

$$(S_0 Q_0 + B_0) = (S_1 Q_1 + B_1) < op > (S_2 Q_2 + B_2)$$

The objective of fixed-point operator replacement is to replace an operator that accepts and returns fixed-point or integer inputs and output with a function that accepts and returns built-in C numeric data types. The following sections provide additional programming information for each supported operator.

Addition

The operation $V_0 = V_1 + V_2$ implies that

$$Q_0 = \left(\frac{S_1}{S_0} \right) Q_1 + \left(\frac{S_2}{S_0} \right) Q_2 + \left(\frac{B_1 + B_2 - B_0}{S_0} \right)$$

If an addition replacement function is defined such that the scaling on the operands and sum are equal and the net bias

$$\left(\frac{B_1 + B_2 - B_0}{S_0} \right)$$

is zero (for example, a function `s8_add_s8_s8` that adds two signed 8-bit values and produces a signed 8-bit result), then the operator entry must set the operator entry parameters `SlopesMustBeTheSame` and `MustHaveZeroNetBias` to `true`. (For parameter descriptions, see the reference page for the function `setTf1COperationEntryParameters`.)

Subtraction

The operation $V_0 = V_1 - V_2$ implies that

$$Q_0 = \left(\frac{S_1}{S_0} \right) Q_1 - \left(\frac{S_2}{S_0} \right) Q_2 + \left(\frac{B_1 - B_2 - B_0}{S_0} \right)$$

If a subtraction replacement function is defined such that the scaling on the operands and difference are equal and the net bias

$$\left(\frac{B_1 - B_2 - B_0}{S_0} \right)$$

is zero (for example, a function `s8_sub_s8_s8` that subtracts two signed 8-bit values and produces a signed 8-bit result), then the operator entry must set the operator entry parameters `SlopesMustBeTheSame` and `MustHaveZeroNetBias` to `true`. (For parameter descriptions, see the reference page for the function `setTf1COperationEntryParameters`.)

Multiplication

There are different ways to specify multiplication replacements. The most direct way is to specify an exact match of the input and output types. This is feasible if a model contains only a few (known) slope and bias combinations. Use the `Tf1COperationEntry` class

and specify the exact values of slope and bias on each argument. For scenarios where there are numerous slope/bias combinations, it is not feasible to specify each value with a different entry. Use a net slope entry or create a custom entry.

The operation $V_0 = V_1 * V_2$ implies, for binary-point-only scaling, that

$$S_0 Q_0 = (S_1 Q_1)(S_2 Q_2)$$

$$Q_0 = \left(\frac{S_1 S_2}{S_0} \right) Q_1 Q_2$$

$$Q_0 = S_n Q_1 Q_2$$

where S_n is the net slope.

It is common to replace all multiplication operations that have a net slope of 1.0 with a function that performs C-style multiplication. For example, to replace all signed 8-bit multiplications that have a net scaling of 1.0 with the `s8_mul_s8_u8` replacement function, the operator entry must define a net slope factor, $F2^E$. You specify the values for F and E using operator entry parameters `NetSlopeAdjustmentFactor` and `NetFixedExponent`. (For parameter descriptions, see the reference page for the function `setTf1CoperationEntryParameters`.) For the `s8_mul_s8_u8` function, set `NetSlopeAdjustmentFactor` to 1 and `NetFixedExponent` to 0.0.

Note: When an operator entry specifies `NetSlopeAdjustmentFactor` and `NetFixedExponent`, matching entries must have arguments with zero bias.

Division

There are different ways to specify division replacements. The most direct way is to specify an exact match of the input and output types. This is feasible if a model contains only a few (known) slope and bias combinations. For this, use the `Tf1CoperationEntry` class and specify the exact values of slope and bias on each argument. For scenarios where there are numerous slope/bias combinations, it is not feasible to specify each value with a different entry. For this, use a net slope entry or create a custom entry (see “Customize Matching and Replacement Process for Functions”).

The operation $V_0 = (V_1 / V_2)$ implies, for binary-point-only scaling, that

$$S_0 Q_0 = \left(\frac{S_1 Q_1}{S_2 Q_2} \right)$$
$$Q_0 = S_n \left(\frac{Q_1}{Q_2} \right)$$

where S_n is the net slope.

It is common to replace all division operations that have a net slope of 1.0 with a function that performs C-style division. For example, to replace all signed 8-bit divisions that have a net scaling of 1.0 with the `s8_mul_s8_u8_` replacement function, the operator entry must define a net slope factor, $F2^E$. You specify the values for F and E using operator entry parameters `NetSlopeAdjustmentFactor` and `NetFixedExponent`. (For parameter descriptions, see the reference page for the function `setTf1COperationEntryParameters`.) For the `s16_netslope0p5_div_s16_s16` function, you would set `NetSlopeAdjustmentFactor` to 1 and `NetFixedExponent` to 0.0.

Note: When an operator entry specifies `NetSlopeAdjustmentFactor` and `NetFixedExponent`, matching entries must have arguments with zero bias.

Data Type Conversion (Cast)

The data type conversion operation $V_0 = V_1$ implies, for binary-point-only scaling, that

$$Q_0 = \left(\frac{S_1}{S_0} \right) Q_1$$
$$Q_0 = S_n Q_1$$

where S_n is the net slope.

Shift

The shift left or shift right operation $V_0 = (V_1 / 2^n)$ implies, for binary-point-only scaling, that

$$S_0 Q_0 = \left(\frac{S_1 Q_1}{2^n} \right)$$

$$Q_0 = \left(\frac{S_1}{S_0} \right) + \left(\frac{Q_1}{2^n} \right)$$

$$Q_0 = S_n \left(\frac{Q_1}{2^n} \right)$$

where S_n is the net slope.

Related Examples

- “Define Code Replacement Mappings”
- “Binary-Point-Only Scaling Code Replacement”
- “Slope Bias Scaling Code Replacement”
- “Net Slope Scaling Code Replacement”
- “Equal Slope and Zero Net Bias Code Replacement”
- “Data Type Conversions (Casts) and Operator Code Replacement”
- “Shift Left Operations and Code Replacement”
- “Data Alignment for Code Replacement”
- “Remap Operator Output to Function Input”
- “Customize Matching and Replacement Process for Operators”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code You Can Replace From Simulink Models”
- “Code Replacement Match and Replacement Process”
- “Code Replacement Libraries”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Binary-Point-Only Scaling Code Replacement

You can define code replacement entries for operations on fixed-point data types such that they match a binary-point-only scaling combination on operator inputs and output. These binary-point-only scaling entries can map the specified binary-point-scaling combination to a replacement function for addition, subtraction, multiplication, or division.

This example creates a code replacement entry for multiplication of fixed-point data types. You specify arguments using binary-point-only scaling. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

- 1 Define a function for the definition. For example:

```
function hTable = crl_table_fixed_binptscale
```

- 2 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 3 Create the entry for the operator mapping with a call to the `RTW.Tf1COperationEntry` function.

```
op_entry = RTW.Tf1COperationEntry;
```

- 4 Set operator entry parameters with a call to the `setTf1COperationEntryParameters` function. The parameters specify the type of operation as multiplication, the saturation mode as saturate on overflow, rounding modes as unspecified, and the name of the replacement function as `s32_mul_s16_s16_binarypoint`.

```
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_MUL', ...
    'Priority', 90, ...
    'SaturationMode', 'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes', {'RTW_ROUND_UNSPECIFIED'}, ...
    'ImplementationName', 's32_mul_s16_s16_binarypoint', ...
    'ImplementationHeaderFile', 's32_mul_s16_s16_binarypoint.h', ...
    'ImplementationSourceFile', 's32_mul_s16_s16_binarypoint.c');
```

- 5 Create conceptual arguments `y1`, `u1`, and `u2`. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. Each argument specifies that the data type is fixed-point, the mode is binary-point-only scaling, and its derived slope and bias values must exactly match the call-site slope and bias values. The output argument is 32 bits, signed, with a

fraction length of 28. The input arguments are 16 bits, signed, with fraction lengths of 15 and 13.

```
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric',...
    'Name',          'y1', ...
    'IOType',        'RTW_IO_OUTPUT', ...
    'CheckSlope',    true, ...
    'CheckBias',     true, ...
    'DataTypeMode',  'Fixed-point: binary point scaling', ...
    'IsSigned',      true, ...
    'WordLength',    32, ...
    'FractionLength', 28);

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'u1', ...
    'IOType',        'RTW_IO_INPUT', ...
    'CheckSlope',    true, ...
    'CheckBias',     true, ...
    'DataTypeMode',  'Fixed-point: binary point scaling', ...
    'IsSigned',      true, ...
    'WordLength',    16, ...
    'FractionLength', 15);

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'u2', ...
    'IOType',        'RTW_IO_INPUT', ...
    'CheckSlope',    true, ...
    'CheckBias',     true, ...
    'DataTypeMode',  'Fixed-point: binary point scaling', ...
    'IsSigned',      true, ...
    'WordLength',    16, ...
    'FractionLength', 13);
```

- 6 Create the implementation arguments. This example uses calls to the `createAndSetCImplementationReturn` and `createAndAddImplementationArg` functions to create and add implementation arguments to the entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types). In this case, the output argument is 32 bits and signed (`int32`). The input arguments are 16 bits and signed (`int16`).

```
createAndSetCImplementationReturn(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'y1', ...
    'IOType',        'RTW_IO_OUTPUT', ...
    'IsSigned',      true, ...
    'WordLength',    32, ...
    'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric',...
    'Name',          'u1', ...
    'IOType',        'RTW_IO_INPUT', ...
    'IsSigned',      true, ...
    'WordLength',    16, ...
    'FractionLength', 0);
```



```

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric',...
    'Name',          'u2', ...
    'IOType',        'RTW_IO_INPUT', ...
    'IsSigned',      true, ...
    'WordLength',    16, ...
    'FractionLength', 0);

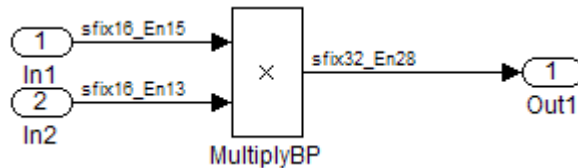
```

- 7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

To test this example:

- 1 Create a model.



- 2 For this model:

- Set the Inport 1 **Data type** to `fixdt(1,16,15)`.
- Set the Inport 2 **Data type** to `fixdt(1,16,13)`.
- In the Product block:
 - Set **Output data type** to `fixdt(1,32,28)`.
 - Select the option **Saturate on integer overflow**.

- 3 Configure the model with the following settings:

- On the **Solver** pane, select a fixed-step, discrete solver.
- On the **Code Generation** pane, select an ERT-based system target file.
- On the **Code Generation > Interface** pane, select the code replacement library that contains your addition operation entry.

- 4 Generate code and a code generation report.

- 5 Review the code replacements.

Related Examples

- “Define Code Replacement Mappings”

- “Review and Test Code Replacements”
- “Fixed-Point Operator Code Replacement”
- “Slope Bias Scaling Code Replacement”
- “Net Slope Scaling Code Replacement”
- “Equal Slope and Zero Net Bias Code Replacement”
- “Data Type Conversions (Casts) and Operator Code Replacement”
- “Shift Left Operations and Code Replacement”
- “Data Alignment for Code Replacement”
- “Remap Operator Output to Function Input”
- “Customize Matching and Replacement Process for Operators”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code You Can Replace From Simulink Models”
- “Code Replacement Match and Replacement Process”
- “Code Replacement Libraries”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Slope Bias Scaling Code Replacement

You can define code replacement for operations on fixed-point data types as matching a slope bias scaling combination on the operator inputs and output. The slope bias scaling entries can map the specified slope bias combination to a replacement function for addition, subtraction, multiplication, or division.

This example creates a code replacement entry for division of fixed-point data types. You specify arguments using slope bias scaling. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

- 1 Define a function for the definition. For example:

```
function hTable = crl_table_fixed_s16divslopebias
```

- 2 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 3 Create the entry for the operator mapping with a call to the `RTW.Tf1COperationEntry` function.

```
op_entry = RTW.Tf1COperationEntry;
```

- 4 Set operator entry parameters with a call to the `setTf1COperationEntryParameters` function. The parameters specify the type of operation as division, the saturation mode as saturate on overflow, rounding modes as round to ceiling, and the name of the replacement function as `s16_div_s16_s16_slopebias`.

```
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_DIV', ...
    'Priority', 90, ...
    'SaturationMode', 'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes', {'RTW_ROUND_CEILING'}, ...
    'ImplementationName', 's16_div_s16_s16_slopebias', ...
    'ImplementationHeaderFile', 's16_div_s16_s16_slopebias.h', ...
    'ImplementationSourceFile', 's16_div_s16_s16_slopebias.c');
```

- 5 Create conceptual arguments `y1`, `u1`, and `u2`. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. Each argument specifies that the data type is fixed-point, the mode is slope bias scaling, and its specified slope and bias values must exactly match the call-site slope and bias values. The output argument and input arguments are 16 bits, signed, each with specific slope bias specifications.

```

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    '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);

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'u1', ...
    'IOType',       'RTW_IO_INPUT', ...
    'CheckSlope',   true, ...
    'CheckBias',    true, ...
    'DataTypeMode', 'Fixed-point: slope and bias scaling', ...
    'IsSigned',     true, ...
    'WordLength',   16, ...
    'Slope',        15, ...
    'Bias',         2);

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'u2', ...
    'IOType',       'RTW_IO_INPUT', ...
    'CheckSlope',   true, ...
    'CheckBias',    true, ...
    'DataTypeMode', 'Fixed-point: slope and bias scaling', ...
    'IsSigned',     true, ...
    'WordLength',   16, ...
    'Slope',        13, ...
    'Bias',         5);

```

- 6 Create the implementation arguments. This example uses calls to the `createAndSetCImplementationReturn` and `createAndAddImplementationArg` functions to create and add implementation arguments to the entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types). In this case, the output and input arguments are 16 bits and signed (`int16`).

```

createAndSetCImplementationReturn(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'y1', ...
    'IOType',       'RTW_IO_OUTPUT', ...
    'IsSigned',     true, ...
    'WordLength',   16, ...
    'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'u1', ...
    'IOType',       'RTW_IO_INPUT', ...
    'IsSigned',     true, ...
    'WordLength',   16, ...
    'FractionLength', 0);

```

```

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric',...
    'Name',          'u2', ...
    'IOType',       'RTW_IO_INPUT', ...
    'IsSigned',     true, ...
    'WordLength',  16, ...
    'FractionLength', 0);

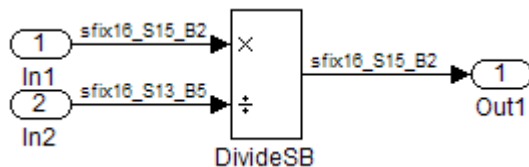
```

- 7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

To test this example:

- 1 Create a model.



- 2 For this model:

- Set the Inport 1 **Data type** to `fixdt(1,16,15,2)`.
- Set the Inport 2 **Data type** to `fixdt(1,16,13,5)`.
- In the Divide block:
 - Set **Output data type** to `Inherit: Inherit via back propagation`.
 - Set **Integer rounding mode** to `Ceiling`.
 - Select the option **Saturate on integer overflow**.

- 3 Configure the model with the following settings:

- On the **Solver** pane, select a fixed-step, discrete solver.
- On the **Code Generation** pane, select an ERT-based system target file.
- On the **Code Generation > Interface** pane, select the code replacement library that contains your addition operation entry.

- 4 Generate code and a code generation report.

- 5 Review the code replacements.

Related Examples

- “Define Code Replacement Mappings”
- “Review and Test Code Replacements”
- “Fixed-Point Operator Code Replacement”
- “Binary-Point-Only Scaling Code Replacement”
- “Net Slope Scaling Code Replacement”
- “Equal Slope and Zero Net Bias Code Replacement”
- “Data Type Conversions (Casts) and Operator Code Replacement”
- “Shift Left Operations and Code Replacement”
- “Data Alignment for Code Replacement”
- “Remap Operator Output to Function Input”
- “Customize Matching and Replacement Process for Operators”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code You Can Replace From Simulink Models”
- “Code Replacement Match and Replacement Process”
- “Code Replacement Libraries”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Net Slope Scaling Code Replacement

In this section...

“Multiplication and Division with Saturation” on page 21-205

“Multiplication and Division with Rounding Mode and Additional Implementation Arguments” on page 21-208

Multiplication and Division with Saturation

You can define code replacement entries for operations on fixed-point data types as matching net slope between operator inputs and output. The net slope entries can map a range of slope and bias values to a replacement function for multiplication or division.

This example creates a code replacement entry for division of fixed-point data types, using wrap on overflow saturation mode and a net slope. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

- 1 Define a function for the definition. For example:

```
function hTable = crl_table_fixed_netslopesaturate
```

- 2 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 3 Create the entry for the operator mapping with a call to the `RTW.Tf1COperationEntryGenerator_Netslope` function, which provides access to the fixed-point parameters `NetSlopeAdjustmentFactor` and `NetFixedExponent`.

```
wv = [16,32];
for iy = 1:2
    for inum = 1:2
        for iden = 1:2
            hTable = getDivOpEntry(hTable, ...
                fixdt(1,wv(iy)),fixdt(1,wv(inum)),fixdt(1,wv(iden)));
        end
    end
end
```

```
%-----
function hTable = getDivOpEntry(hTable, dty, dtnum, dtdden)
%-----
```

```
% Create an entry for division of fixed-point data types where
% arguments are specified using Slope and Bias scaling
% Saturation on, Rounding unspecified
```

```
funcStr = sprintf('user_div_%s_%s_%s',...
    typeStrFunc(dty),...
    typeStrFunc(dtnum),...
    typeStrFunc(dtdden));
```

```
op_entry = RTW.Tf1COperationEntryGenerator_NetSlope;
```

- 4 Set operator entry parameters with a call to the `setTf1COperationEntryParameters` function. The parameters specify the type of operation as division, the saturation mode as wrap on overflow, rounding modes as unspecified, and the name of the replacement function as `user_div_*`. `NetSlopeAdjustmentFactor` and `NetFixedExponent` specify the F and E parts of the net slope $F2^E$.

```
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_DIV', ...
    'Priority', 90, ...
    'SaturationMode', 'RTW_WRAP_ON_OVERFLOW',...
    'RoundingModes', {'RTW_ROUND_UNSPECIFIED'},...
    'NetSlopeAdjustmentFactor', 1.0, ...
    'NetFixedExponent', 0.0, ...
    'ImplementationName', funcStr, ...
    'ImplementationHeaderFile', [funcStr, '.h'], ...
    'ImplementationSourceFile', [funcStr, '.c']);
```

- 5 Create conceptual arguments `y1`, `u1`, and `u2`. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. Specify each argument as fixed-point and signed. Also, for each argument, specify that code replacement request processing does *not* check for an exact match to the call-site slope and bias values.

```
createAndAddConceptualArg(op_entry, ...
    'RTW.Tf1ArgNumeric', ...
    'Name', 'y1',...
    'IOType', 'RTW_IO_OUTPUT',...
    'CheckSlope', false,...
    'CheckBias', false,...
    'DataTypeMode', 'Fixed-point: slope and bias scaling',...
    'IsSigned', dtypes.Signed,...
    'WordLength', dtypes.WordLength,...
    'Bias', 0);
```

```
createAndAddConceptualArg(op_entry, ...
    'RTW.Tf1ArgNumeric', ...
    'Name', 'u1', ...
    'IOType', 'RTW_IO_INPUT',...
    'CheckSlope', false,...
    'CheckBias', false,...
```



```

    'DataTypeMode', 'Fixed-point: slope and bias scaling',...
    'IsSigned',    dtnum.Signed,...
    'WordLength',  dtnum.WordLength,...
    'Bias',        0);

createAndAddConceptualArg(op_entry, ...
    'RTW.Tf1ArgNumeric', ...
    'Name',        'u2', ...
    'IOType',      'RTW_IO_INPUT',...
    'CheckSlope',  false,...
    'CheckBias',   false,...
    'DataTypeMode', 'Fixed-point: slope and bias scaling',...
    'IsSigned',    dtden.Signed,...
    'WordLength',  dtden.WordLength,...
    'Bias',        0);

```

- 6** Create the implementation arguments. This example uses calls to the `getTf1ArgFromString` function to create the arguments. Implementation arguments must describe fundamental numeric data types (not fixed-point data types). The convenience methods `setReturn` and `addArgument` specify whether an argument is a return value or argument. These methods add the argument to the entry array of implementation arguments.

```

arg = getTf1ArgFromString(hTable, 'y1', typeStrBase(dty));
op_entry.Implementation.setReturn(arg);

arg = getTf1ArgFromString(hTable, 'u1', typeStrBase(dtnum));
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'u2', typeStrBase(dtden));
op_entry.Implementation.addArgument(arg);

```

- 7** Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

- 8** Define functions that determine the data type word length.

```

%-----
function str = typeStrFunc(dt)
%-----

if dt.Signed
    sstr = 's';
else
    sstr = 'u';
end
str = sprintf('%s%d', sstr, dt.WordLength);

%-----
function str = typeStrBase(dt)
%-----

if dt.Signed

```

```

    sstr = ;
else
    sstr = 'u';
end
str = sprintf('%sint%d',sstr,dt.WordLength);

```

Multiplication and Division with Rounding Mode and Additional Implementation Arguments

You can define code replacement entries for multiplication and division operations on fixed-point data types such that they match the net slope between operator inputs and output. The net slope entries can map a range of slope and bias values to a replacement function for multiplication or division.

This example creates a code replacement entry for division of fixed-point data types, using the ceiling rounding mode and a net slope scaling factor. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

- 1 Define a function for the definition. For example:

```
function hTable = crl_table_fixed_netsloperound
```

- 2 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 3 Create the entry for the operator mapping with a call to the `RTW.Tf1COperationEntryGenerator_Netslope` function, which provides access to the fixed-point parameters `NetSlopeAdjustmentFactor` and `NetFixedExponent`.

```
op_entry = RTW.Tf1COperationEntryGenerator_NetSlope;
```

- 4 Set operator entry parameters with a call to the `setTf1COperationEntryParameters` function. The parameters specify the type of operation as division, the saturation mode as saturation off, rounding modes as round to ceiling, and the name of the replacement function as `s16_div_s16_s16`. `NetSlopeAdjustmentFactor` and `NetFixedExponent` specify the F and E parts of the relative scaling factor $F2^E$.

```

setTf1COperationEntryParameters(op_entry, ...
    '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');

```

- 5 Create conceptual arguments `y1`, `u1`, and `u2`. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. Specify each argument as fixed-point, 16 bits, and signed. Also, for each argument, specify that code replacement request processing does *not* check for an exact match to the call-site slope and bias values.

```

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'y1', ...
    'IOType', 'RTW_IO_OUTPUT', ...
    'CheckSlope', false, ...
    'CheckBias', false, ...
    'DataType', 'Fixed', ...
    'IsSigned', true, ...
    'WordLength', 16);

```

```

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'u1', ...
    'IOType', 'RTW_IO_INPUT', ...
    'CheckSlope', false, ...
    'CheckBias', false, ...
    'DataType', 'Fixed', ...
    'IsSigned', true, ...
    'WordLength', 16);

```

```

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'u2', ...
    'IOType', 'RTW_IO_INPUT', ...
    'CheckSlope', false, ...
    'CheckBias', false, ...
    'DataType', 'Fixed', ...
    'IsSigned', true, ...
    'WordLength', 16);

```

- 6 Create the implementation arguments. This example uses calls to the `createAndSetCImplementationReturn` and `createAndAddImplementationArg` functions to create and add implementation arguments to the entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types). In this case, the output and input arguments are 16 bits and signed (`int16`).

```

createAndSetCImplementationReturn(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'y1', ...
    'IOType', 'RTW_IO_OUTPUT', ...
    'IsSigned', true, ...
    'WordLength', 16, ...
    'FractionLength', 0);

```

```

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric',...
                              'Name',      'u1', ...
                              'IOType',    'RTW_IO_INPUT', ...
                              'IsSigned',  true, ...
                              'WordLength', 16, ...
                              'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric',...
                              'Name',      'u2', ...
                              'IOType',    'RTW_IO_INPUT', ...
                              'IsSigned',  true, ...
                              'WordLength', 16, ...
                              'FractionLength', 0);

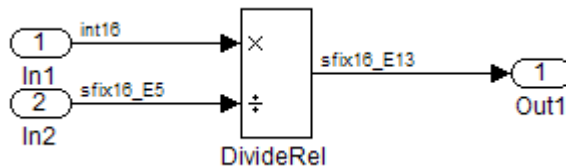
```

- 7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

To test this example:

- 1 Create a model.



- 2 For this model:

- Set the Inport 1 **Data type** to `int16`.
- Set the Inport 2 **Data type** to `fixdt(1,16,-5)`.
- In the Divide block:
 - Set **Output data type** to `fixdt(1,16,-13)`.
 - Set **Integer rounding mode** to `Ceiling`.

- 3 Configure the model with the following settings:

- On the **Solver** pane, select a fixed-step, discrete solver.
- On the **Code Generation** pane, select an ERT-based system target file.
- On the **Code Generation > Interface** pane, select the code replacement library that contains your addition operation entry.

- 4 Generate code and a code generation report.

- 5 Review the code replacements.

Related Examples

- “Define Code Replacement Mappings”
- “Review and Test Code Replacements”
- “Fixed-Point Operator Code Replacement”
- “Binary-Point-Only Scaling Code Replacement”
- “Slope Bias Scaling Code Replacement”
- “Equal Slope and Zero Net Bias Code Replacement”
- “Data Type Conversions (Casts) and Operator Code Replacement”
- “Shift Left Operations and Code Replacement”
- “Data Alignment for Code Replacement”
- “Remap Operator Output to Function Input”
- “Customize Matching and Replacement Process for Operators”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code You Can Replace From Simulink Models”
- “Code Replacement Match and Replacement Process”
- “Code Replacement Libraries”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Equal Slope and Zero Net Bias Code Replacement

You can define code replacement entries for addition or subtraction of fixed-point data types such that they match relative slope and bias values (equal slope and zero net bias) across operator inputs and output. These entries allow you to disregard slope and bias values. Map relative slope and bias values to a replacement function for addition or subtraction.

This example creates a code replacement entry for addition of fixed-point data types. Slopes must be equal and net bias must be zero across the operator inputs and output. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

- 1 Define a function for the definition.

```
function hTable = crl_table_fixed_slopeseq_netbiaszero
```

- 2 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 3 Create the entry for the operator mapping with a call to the `RTW.Tf1COperationEntryGenerator` function, which provides access to the fixed-point parameters `SlopesMustBeTheSame` and `MustHaveZeroNetBias`.

```
op_entry = RTW.Tf1COperationEntryGenerator;
```

- 4 Set operator entry parameters with a call to the `setTf1COperationEntryParameters` function. The parameters specify the type of operation as addition, the saturation mode as saturation off, rounding modes as unspecified, and the name of the replacement function as `u16_add_SameSlopeZeroBias`. `SlopesMustBeTheSame` and `MustHaveZeroNetBias` are set to `true`, indicating that slopes must be equal and net bias must be zero across the addition inputs and output.

```
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_ADD', ...
    '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');
```

- 5 Create conceptual arguments y_1 , u_1 , and u_2 . This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. Each argument is specified as 16 bits and unsigned. Each argument specifies that code replacement request processing does *not* check for an exact match to the call-site slope and bias values.

```
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'y1', ...
    'IOType',        'RTW_IO_OUTPUT', ...
    'CheckSlope',    false, ...
    'CheckBias',     false, ...
    'IsSigned',      false, ...
    'WordLength',    16);
```

```
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'u1', ...
    'IOType',        'RTW_IO_INPUT', ...
    'CheckSlope',    false, ...
    'CheckBias',     false, ...
    'IsSigned',      false, ...
    'WordLength',    16);
```

```
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'u2', ...
    'IOType',        'RTW_IO_INPUT', ...
    'CheckSlope',    false, ...
    'CheckBias',     false, ...
    'IsSigned',      false, ...
    'WordLength',    16);
```

- 6 Create the implementation arguments. This example uses calls to the `createAndSetCImplementationReturn` and `createAndAddImplementationArg` functions to create and add implementation arguments to the entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types). In this case, the output and input arguments are 16 bits and unsigned (`uint16`).

```
createAndSetCImplementationReturn(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'y1', ...
    'IOType',        'RTW_IO_OUTPUT', ...
    'IsSigned',      false, ...
    'WordLength',    16, ...
    'FractionLength', 0);
```

```
createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric',...
    'Name',          'u1', ...
    'IOType',        'RTW_IO_INPUT', ...
    'IsSigned',      false, ...
    'WordLength',    16, ...
    'FractionLength', 0);
```

```
createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric',...
```

```

    'Name',          'u2', ...
    'IOType',       'RTW_IO_INPUT', ...
    'IsSigned',     false, ...
    'WordLength',   16, ...
    'FractionLength', 0);

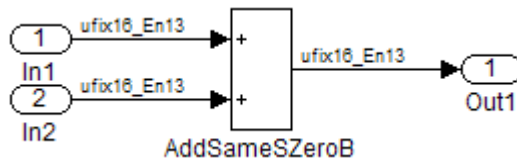
```

- 7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

To test this example:

- 1 Create a model.



- 2 For this model:

- Set the Inport 1 **Data type** to `fixdt(0,16,13)`.
- Set the Inport 2 **Data type** to `fixdt(0,16,13)`.
- In the Add block:
 - Verify that **Output data type** is set to its default, `Inherit via internal rule`.
 - Set **Integer rounding mode** to `Zero`.

- 3 Configure the model with the following settings:

- On the **Solver** pane, select a fixed-step, discrete solver.
- On the **Code Generation** pane, select an ERT-based system target file.
- On the **Code Generation > Interface** pane, select the code replacement library that contains your addition operation entry.

- 4 Generate code and a code generation report.

- 5 Review the code replacements.

Related Examples

- “Define Code Replacement Mappings”

- “Review and Test Code Replacements”
- “Fixed-Point Operator Code Replacement”
- “Binary-Point-Only Scaling Code Replacement”
- “Slope Bias Scaling Code Replacement”
- “Net Slope Scaling Code Replacement”
- “Data Type Conversions (Casts) and Operator Code Replacement”
- “Shift Left Operations and Code Replacement”
- “Data Alignment for Code Replacement”
- “Remap Operator Output to Function Input”
- “Customize Matching and Replacement Process for Operators”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code You Can Replace From Simulink Models”
- “Code Replacement Match and Replacement Process”
- “Code Replacement Libraries”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Data Type Conversions (Casts) and Operator Code Replacement

In this section...

“Casts From `int32` To `int16`” on page 21-216

“Casts Using Net Slope” on page 21-217

Casts From `int32` To `int16`

You can use code replacement entries to replace code that the code generator produces for data type conversion (cast) operations.

This example creates a code replacement entry that replaces `int32` to `int16` data type conversion (cast) operations. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

- 1 Define a function for the definition.

```
function hTable = crl_table_cast_int32_to_int16
```

- 2 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 3 Create the entry for the operator mapping with a call to the `RTW.Tf1COperationEntry` function.

```
op_entry = RTW.Tf1COperationEntry;
```

- 4 Set operator entry parameters with a call to the `setTf1COperationEntryParameters` function. The parameters specify the type of operation as cast, the saturation mode as saturate on overflow, rounding modes as toward negative infinity, and the name of the replacement function as `my_sat_cast`.

```
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_CAST', ...
    'Priority', 50, ...
    'ImplementationName', 'my_sat_cast', ...
    'SaturationMode', 'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes', {'RTW_ROUND_FLOOR'}, ...
    'ImplementationHeaderFile', 'some_hdr.h', ...
    'ImplementationSourceFile', 'some_hdr.c');
```

- 5 Create the `int16` argument as conceptual argument `y1` and the implementation return value. This example uses calls to the `getTf1ArgFromString` and

`addConceptualArg` functions to create the conceptual argument and add it to the entry. Convenience method `setReturn` specifies the argument as the implementation return value.

```
arg = getTf1ArgFromString(hTable, 'y1', 'int16');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(op_entry, arg);
op_entry.Implementation.setReturn(arg);
```

- 6 Create the `int32` argument as conceptual and implementation argument `u1`. This example uses calls to the `getTf1ArgFromString` and `addConceptualArg` functions to create the conceptual argument and add it to the entry. Convenience method `addArgument` specifies the argument as implementation input argument.

```
arg = getTf1ArgFromString(hTable, 'u1', 'int32');
addConceptualArg(op_entry, arg);
op_entry.Implementation.addArgument(arg);
```

- 7 Add the entry to a code replacement table with a call to the `addEntry` function.

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

Casts Using Net Slope

You can use code replacement entries to replace code that the code generator produces for data type conversion (cast) operations.

This example creates a code replacement entry to replace data type conversions (casts) of fixed-point data types by using a net slope. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

- 1 Define a function for the definition.

```
function hTable = cr1_table_cast_fixpt_net_slope
```

- 2 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 3 Create the entry for the operator mapping with a call to the `RTW.Tf1COperationEntryGenerator_Netslope` function, which provides access to the fixed-point parameters `NetSlopeAdjustmentFactor` and `NetFixedExponent`

```
op_entry = RTW.Tf1COperationEntryGenerator_NetSlope;
```

- 4 Set operator entry parameters with a call to the `setTf1COperationEntryParameters` function. The parameters specify the type of operation as cast, the saturation mode as saturate on overflow, rounding modes as toward negative infinity, and the name of the replacement function as `my_fxp_cast`. `NetSlopeAdjustmentFactor` and `NetFixedExponent` specify the F and E parts of the net slope $F2^E$.

```
InFL = 2;
InWL = 16;
InSgn = true;
OutFL = 4;
OutWL = 32;
OutSgn = true;
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_CAST', ...
    'Priority', 50, ...
    'SaturationMode', 'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes', {'RTW_ROUND_FLOOR'}, ...
    'NetSlopeAdjustmentFactor', 1.0, ...
    'NetFixedExponent', (OutFL - InFL), ...
    'ImplementationName', 'my_fxp_cast', ...
    'ImplementationHeaderFile', 'some_hdr.h', ...
    'ImplementationSourceFile', 'some_hdr.c');
```

- 5 Create conceptual arguments `y1` and `u1`. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. Each argument is specified as fixed-point and signed. Each argument specifies that code replacement request processing does *not* check for an exact match to the call-site slope and bias values.

```
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'y1', ...
    'IOType', 'RTW_IO_OUTPUT', ...
    'CheckSlope', false, ...
    'CheckBias', false, ...
    'DataTypeMode', 'Fixed-point: binary point scaling', ...
    'IsSigned', OutSgn, ...
    'WordLength', OutWL, ...
    'FractionLength', OutFL);

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'u1', ...
    'IOType', 'RTW_IO_INPUT', ...
    'CheckSlope', false, ...
    'CheckBias', false, ...
    'DataTypeMode', 'Fixed-point: binary point scaling', ...
    'IsSigned', InSgn, ...
    'WordLength', InWL, ...
    'FractionLength', InFL);
```

- 6 Create the implementation arguments. This example uses calls to the `createAndSetCImplementationReturn` and `createAndAddImplementationArg` functions to create and add implementation arguments to the entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types).

```
createAndSetCImplementationReturn(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'y1', ...
    'IOType',        'RTW_IO_OUTPUT', ...
    'IsSigned',      OutSgn, ...
    'WordLength',    OutWL, ...
    'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'u1', ...
    'IOType',        'RTW_IO_INPUT', ...
    'IsSigned',      InSgn, ...
    'WordLength',    InWL, ...
    'FractionLength', 0);
```

- 7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

Related Examples

- “Define Code Replacement Mappings”
- “Fixed-Point Operator Code Replacement”
- “Binary-Point-Only Scaling Code Replacement”
- “Slope Bias Scaling Code Replacement”
- “Net Slope Scaling Code Replacement”
- “Equal Slope and Zero Net Bias Code Replacement”
- “Shift Left Operations and Code Replacement”
- “Data Alignment for Code Replacement”
- “Remap Operator Output to Function Input”
- “Customize Matching and Replacement Process for Operators”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “What Is Code Replacement?”

- “What Is Code Replacement Customization?”
- “Code You Can Replace From Simulink Models”
- “Code Replacement Match and Replacement Process”
- “Code Replacement Libraries”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Shift Left Operations and Code Replacement

In this section...

“Shift Lefts for `int16` Data” on page 21-221

“Shift Lefts Using Net Slope” on page 21-222

Shift Lefts for `int16` Data

You can use code replacement entries to replace code that the code generator produces for shift (`<<`) operations.

This example creates a code replacement entry to replace shift left operations for `int16` data. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

- 1 Define a function for the definition.

```
function hTable = crl_table_shift_left_int16
```

- 2 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 3 Create the entry for the operator mapping with a call to the `RTW.Tf1COperationEntry` function.

```
op_entry = RTW.Tf1COperationEntry;
```

- 4 Set operator entry parameters with a call to the `setTf1COperationEntryParameters` function. The parameters specify the type of operation as shift left and the name of the replacement function as `my_shift_left`.

```
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_SL', ...
    'Priority', 50, ...
    'ImplementationName', 'my_shift_left', ...
    'ImplementationHeaderFile', 'some_hdr.h', ...
    'ImplementationSourceFile', 'some_hdr.c');
```

- 5 Create the `int16` argument as conceptual argument `y1` and the implementation return value. This example uses calls to the `getTf1ArgFromString` and `addConceptualArg` functions to create the conceptual argument and add it to the entry. Convenience method `setReturn` specifies the argument as the implementation return value.

```
arg = getTf1ArgFromString(hTable, 'y1', 'int16');
```

```
arg.IOType = 'RTW_IO_OUTPUT';  
addConceptualArg(op_entry, arg);  
op_entry.Implementation.setReturn(arg);
```

- 6 Create the `int16` argument as conceptual and implementation argument `u1`. This example uses calls to the `getTf1ArgFromString` and `addConceptualArg` functions to create the conceptual argument and add it to the entry. Convenience method `addArgument` specifies the argument as an implementation input argument.

```
arg = getTf1ArgFromString(hTable, 'u1', 'int16');  
addConceptualArg(op_entry, arg);  
op_entry.Implementation.addArgument(arg);
```

- 7 Create the `int8` argument as conceptual and implementation argument `u2`. This example uses calls to the `getTf1ArgFromString` and `addConceptualArg` functions to create the conceptual argument and add it to the entry. This argument specifies the number of bits to shift the previous input argument. Because the argument type is not relevant, the example disables type checking by setting the `CheckType` property to `false`. Convenience method `addArgument` specifies the argument as implementation input argument.

```
arg = getTf1ArgFromString(hTable, 'u2', 'int8');  
arg.CheckType = false;  
addConceptualArg(op_entry, arg);  
op_entry.Implementation.addArgument(arg);
```

- The function `getTf1ArgFromString` is called to create an `int8` input argument. This argument is added to the operator entry both as the third conceptual argument and the second implementation input argument.
- Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

Shift Lefts Using Net Slope

You can use code replacement entries to replace code that the code generator produces for shift (`<<`) operations.

This example creates a code replacement entry to replace shift left operations for fixed-point data using a net slope. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

- 1 Define a function for the definition.

```
function hTable = cr1_table_shift_left_fixpt_net_slope
```


- 2 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 3 Create the entry for the operator mapping with a call to the `RTW.Tf1COperationEntryGenerator_Netslope` function. This function provides access to the fixed-point parameters `NetSlopeAdjustmentFactor` and `NetFixedExponent`.

```
op_entry = RTW.Tf1COperationEntryGenerator_NetSlope;
```

- 4 Set operator entry parameters with a call to the `setTf1COperationEntryParameters` function. The parameters specify the type of operation as shift left, the saturation mode as saturate on overflow, rounding modes as toward negative infinity, and the name of the replacement function as `my_fxp_shift_left`. `NetSlopeAdjustmentFactor` and `NetFixedExponent` specify the F and E parts of the net slope $F2^E$.

```
InFL = 2;
InWL = 16;
InSgn = true;
OutFL = 4;
OutWL = 32;
OutSgn = true;
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_SL', ...
    'Priority', 50, ...
    'SaturationMode', 'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes', {'RTW_ROUND_FLOOR'}, ...
    'NetSlopeAdjustmentFactor', 1.0, ...
    'NetFixedExponent', (OutFL - InFL), ...
    'ImplementationName', 'my_fxp_shift_left', ...
    'ImplementationHeaderFile', 'some_hdr.h', ...
    'ImplementationSourceFile', 'some_hdr.c');
```

- 5 Create conceptual arguments `y1` and `u1`. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. Each argument is specified as fixed-point and signed. Each argument specifies that code replacement request processing does *not* check for an exact match to the call-site slope and bias values.

```
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'y1', ...
    'IOType', 'RTW_IO_OUTPUT', ...
    'CheckSlope', false, ...
    'CheckBias', false, ...
    'DataTypeMode', 'Fixed-point: binary point scaling', ...
    'IsSigned', OutSgn, ...
    'WordLength', OutWL, ...
    'FractionLength', OutFL);
```

```

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'u1', ...
    'IOType', 'RTW_IO_INPUT', ...
    'CheckSlope', false, ...
    'CheckBias', false, ...
    'DataTypeMode', 'Fixed-point: binary point scaling', ...
    'IsSigned', InSgn, ...
    'WordLength', InWL, ...
    'FractionLength', InFL);

```

- 6 Create the implementation arguments. This example uses calls to the `createAndSetCImplementationReturn` and `createAndAddImplementationArg` functions to create and add implementation arguments to the entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types).

```

createAndSetCImplementationReturn(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'y1', ...
    'IOType', 'RTW_IO_OUTPUT', ...
    'IsSigned', OutSgn, ...
    'WordLength', OutWL, ...
    'FractionLength', 0);

```

```

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'u1', ...
    'IOType', 'RTW_IO_INPUT', ...
    'IsSigned', InSgn, ...
    'WordLength', InWL, ...
    'FractionLength', 0);

```

- 7 Create the `int8` argument as conceptual and implementation argument `u2`. This example uses calls to the `getTf1ArgFromString` and `addConceptualArg` functions to create the conceptual argument and add it to the entry. This argument specifies the number of bits to shift the previous input argument. Because the argument type is not relevant, type checking is disabled by setting the `CheckType` property to `false`. Convenience method `addArgument` specifies the argument as implementation input argument.

```

arg = getTf1ArgFromString(hTable, 'u2', 'uint8');
arg.CheckType = false;
addConceptualArg(op_entry, arg);
op_entry.Implementation.addArgument(arg);

```

- 8 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

Related Examples

- “Define Code Replacement Mappings”

- “Fixed-Point Operator Code Replacement”
- “Binary-Point-Only Scaling Code Replacement”
- “Slope Bias Scaling Code Replacement”
- “Net Slope Scaling Code Replacement”
- “Equal Slope and Zero Net Bias Code Replacement”
- “Data Type Conversions (Casts) and Operator Code Replacement”
- “Data Alignment for Code Replacement”
- “Remap Operator Output to Function Input”
- “Customize Matching and Replacement Process for Operators”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code You Can Replace From Simulink Models”
- “Code Replacement Match and Replacement Process”
- “Code Replacement Libraries”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Code Replacement Customization for MATLAB Code

- “What Is Code Replacement Customization?” on page 22-3
- “Code You Can Replace from MATLAB Code” on page 22-4
- “Code Replacement Match and Replacement Process” on page 22-14
- “Code Replacement Customization Limitations” on page 22-15
- “Develop a Code Replacement Library” on page 22-16
- “Quick Start Library Development” on page 22-17
- “Identify Code Replacement Requirements” on page 22-27
- “Prepare for Code Replacement Library Development” on page 22-30
- “Define Code Replacement Mappings” on page 22-31
- “Specify Build Information for Replacement Code” on page 22-47
- “Register Code Replacement Mappings” on page 22-55
- “Review and Test Code Replacements” on page 22-63
- “Deploy Code Replacement Library” on page 22-72
- “Math Function Code Replacement” on page 22-73
- “Memory Function Code Replacement” on page 22-75
- “Specify In-Place Code Replacement” on page 22-77
- “Replace MATLAB Functions with Custom Code Using `coder.replace`” on page 22-84
- “Reserved Identifiers and Code Replacement” on page 22-86
- “Customize Matching and Replacement Process for Functions” on page 22-88
- “Scalar Operator Code Replacement” on page 22-90
- “Addition and Subtraction Operator Code Replacement” on page 22-92
- “Small Matrix Operation to Processor Code Replacement” on page 22-97

- “Matrix Multiplication Operation to MathWorks BLAS Code Replacement” on page 22-101
- “Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement” on page 22-108
- “Remap Operator Output to Function Input” on page 22-115
- “Customize Matching and Replacement Process for Operators” on page 22-118
- “Fixed-Point Operator Code Replacement” on page 22-124
- “Binary-Point-Only Scaling Code Replacement” on page 22-134
- “Slope Bias Scaling Code Replacement” on page 22-137
- “Net Slope Scaling Code Replacement” on page 22-140
- “Equal Slope and Zero Net Bias Code Replacement” on page 22-147
- “Data Type Conversions (Casts) and Operator Code Replacement” on page 22-150
- “Shift Left Operations and Code Replacement” on page 22-154

What Is Code Replacement Customization?

Customize how and when the code generator replaces C/C++ code that it generates by default for functions and operators by developing a custom code replacement library. You can develop libraries interactively with the Code Replacement Tool or programmatically.

- Develop libraries tailored to specific application requirements
- Add identifiers to the list of reserved keywords the code generator considers during code replacement
- Customize the code generator’s matching and replacement process for functions

To get started, “Quick Start Library Development”.

Related Examples

- “Quick Start Library Development”
- “Develop a Code Replacement Library”

More About

- “What Is Code Replacement?”
- “Code You Can Replace from MATLAB Code”
- “Code Replacement Libraries”
- “Code Replacement Terminology”
- “Code Replacement Customization Limitations”

Code You Can Replace from MATLAB Code

In this section...

“About Code You Can Replace” on page 22-4

“Math Functions” on page 22-4

“Memory Functions” on page 22-9

“Operators” on page 22-9

About Code You Can Replace

Code that the code generator replaces depends on the code replacement library (CRL) that you use. By default, the code generator does not apply a code replacement library. Your choice of libraries is dependent on product licensing and whether you have access to custom libraries.

Math Functions

When generating C/C++ code from MATLAB code, depending on code replacement libraries available in your development environment, you can configure the code generator to replace instances of the following math functions with application-specific implementations.

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
abs ¹	Floating point	Scalar	Real
acos	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
acosd	Floating point	Scalar Vector Matrix	Real Complex
acot	Floating point	Scalar Vector Matrix	Real Complex
acotd	Floating point	Scalar	Real

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
		Vector Matrix	Complex
acoth	Floating point	Scalar Vector Matrix	Real Complex
acsc	Floating point	Scalar Vector Matrix	Real Complex
acscd	Floating point	Scalar Vector Matrix	Real Complex
acsch	Floating point	Scalar Vector Matrix	Real Complex
asec	Floating point	Scalar Vector Matrix	Real Complex
asecd	Floating point	Scalar Vector Matrix	Real Complex
asech	Floating point	Scalar Vector Matrix	Real Complex
asin	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
asind	Floating point	Scalar Vector Matrix	Real Complex

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
atan	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
atan2	Floating point	Scalar Vector Matrix	Real
atan2d	Floating point	Scalar Vector Matrix	Real
atand	Floating point	Scalar Vector Matrix	Real Complex
cos	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
ceil	<ul style="list-style-type: none"> • Floating-point • Scalar 	<ul style="list-style-type: none"> • Floating-point • Scalar 	<ul style="list-style-type: none"> • Floating-point • Scalar
cosd	Floating point	Scalar Vector Matrix	Real Complex
cosh	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
cot	Floating point	Scalar Vector Matrix	Real Complex
cotd	Floating point	Scalar Vector Matrix	Real Complex

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
coth	Floating point	Scalar Vector Matrix	Real Complex
csc	Floating point	Scalar Vector Matrix	Real Complex
cscd	Floating point	Scalar Vector Matrix	Real Complex
csch	Floating point	Scalar Vector Matrix	Real Complex
exp	Floating point	Scalar	Real
fix	Floating point	Scalar	Real
floor	<ul style="list-style-type: none"> • Floating-point • Scalar 	<ul style="list-style-type: none"> • Floating-point • Scalar 	<ul style="list-style-type: none"> • Floating-point • Scalar
hypot	Floating point	Scalar Vector Matrix	Real
ldexp	Floating point	Scalar	Real
log	Floating point	Scalar Vector Matrix	Real Complex
log10	Floating point	Scalar Vector Matrix	Real Complex
log2	Floating point	Scalar Vector Matrix	Real Complex
max	Integer Floating point	Scalar	Real

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
min	Integer Floating point	Scalar	Real
pow	Floating point	Scalar	Real
rem	Floating point	Scalar	Real
round	Floating point	Scalar	Real
sec	Floating point	Scalar Vector Matrix	Real Complex
secd	Floating point	Scalar Vector Matrix	Real Complex
sech	Floating point	Scalar Vector Matrix	Real Complex
sign	Floating point	Scalar	Real
sin	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
sind	Floating point	Scalar Vector Matrix	Real Complex
sinh	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
sqrt	Floating point	Scalar	Real
tan	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
<code>tand</code>	Floating point	Scalar Vector Matrix	Real Complex
<code>tanh</code>	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
¹ Wrap on integer overflow only			

Memory Functions

Depending on code replacement libraries available in your development environment, you can configure the code generator to replace instances of the following memory functions with application-specific implementations.

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
<code>memcmp</code>	Void pointer (<code>void*</code>)	Scalar Vector Matrix	Real Complex
<code>memcpy</code>	Void pointer (<code>void*</code>)	Scalar Vector Matrix	Real Complex
<code>memset</code>	Void pointer (<code>void*</code>)	Scalar Vector Matrix	Real Complex
<code>memset2zero</code>	Void pointer (<code>void*</code>)	Scalar Vector Matrix	Real Complex

Some target processors provide optimized functions to set memory to zero. Use the code replacement library programming interface to replace the `memset2zero` function with more efficient target-specific functions.

Operators

When generating C/C++ code from MATLAB code, depending on code replacement libraries available in your development environment, you can configure the code generator to replace instances of the following operators with application-specific implementations.

Mixed data type support indicates you can specify different data types of different inputs.

Operator	Key	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
Addition (+)	RTW_OP_ADD	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Subtraction (-)	RTW_OP_MINUS	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Multiplication (*) ¹	RTW_OP_MUL	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Division (/)	RTW_OP_DIV	Integer Floating point Fixed-point Mixed	Scalar	Real Complex
Data type conversion (cast)	RTW_OP_CAST	Integer Floating point ² Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Shift left (<<)	RTW_OP_SL	Integer Fixed-point Mixed	Scalar Vector Matrix	Real
Shift right arithmetic (>>) ³	RTW_OP_SRA	Integer Fixed-point Mixed	Scalar Vector Matrix	Real

Operator	Key	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
Shift right logical (>>)	RTW_OP_SRL	Integer Fixed-point Mixed	Scalar Vector Matrix	Real
Element-wise matrix multiplication (.*) ⁴	RTW_OP_ELEM_MUL	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Complex conjugation	RTW_OP_CONJUGATE	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Transposition (.')	RTW_OP_TRANS	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Hermitian (complex conjugate) transposition (')	RTW_OP_HERMITIAN	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Multiplication with transposition ¹	RTW_OP_TRMUL	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Multiplication with Hermitian transposition ¹	RTW_OP_HMMUL	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Greater than (>)	RTW_OP_GREATER_THAN	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex

Operator	Key	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
Greater than or equal (\geq)	RTW_OP_GREATER_THAN_OR_EQUAL	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Less than ($<$)	RTW_OP_LESS_THAN	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Less than or equal (\leq)	RTW_OP_LESS_THAN_OR_EQUAL	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Equal (\equiv)	RTW_OP_EQUAL	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Not equal (\neq)	RTW_OP_NOT_EQUAL	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
<p>¹ Can map to Basic Linear Algebra Subroutine (BLAS) multiplication functions.</p> <p>² Scaled floating point is not supported.</p> <p>³ Code replacement libraries that provide arithmetic shift right implementations should also provide logical shift right implementations, because some arithmetic shift rights are converted to logical shift rights during code generation.</p> <p>⁴ Use the multiplication (*) operator (RTW_OP_MUL) for scalar multiplication.</p>				

Related Examples

- “Quick Start Library Development”
- “Develop a Code Replacement Library”

- “Choose a Code Replacement Library”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code Replacement Libraries”
- “Code Replacement Terminology”
- “Code Replacement Customization Limitations”

Code Replacement Match and Replacement Process

When the code generator encounters a call site for a function or operator, it:

- 1 Creates and partially populates a code replacement entry object with the function or operator name or key and conceptual arguments.
- 2 Uses the entry object to query the configured code replacement library for a conceptual representation match. The code generator searches the tables in a code replacement library for a match in the order that the tables appear in the library. When searching for a match, the code generator takes into account:
 - Conceptual name or key
 - Arguments, including quantity, type, type qualifiers, and complexity
 - Algorithm (computation method)
 - Fixed-point saturation and rounding modes
 - Priority
- 3 When a match exists, the code generator returns a code replacement object, fully populated with the conceptual representation, implementation representation, and priority. If the code generator finds multiple matches within a table, the entry priority determines the match. The priority can range from 0 to 100. The highest priority is 0. The code generator uses a higher-priority entry over a similar entry with a lower priority.
- 4 Uses the C or C++ replacement function prototype in the code replacement object to generate code.

Related Examples

- “Customize Matching and Replacement Process for Functions”
- “Customize Matching and Replacement Process for Operators”

More About

- “Code Replacement Libraries”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Code Replacement Customization Limitations

- Code replacement verification — It is possible that code replacement behaves differently than you expect. For example, data types that you observe in code generator input might not match what the code generator uses as intermediate data types during an operation. Verify code replacements by examining generated code. See “Review and Test Code Replacements”.
- Tokens in file paths—You can include tokens in file paths when specifying build information for a code replacement entry by using the programming interface only. The ability to include tokens is not available from the Code Replacement Tool. See “Specify Build Information for Replacement Code”.
- Addition and subtraction operation replacements—See “Addition and Subtraction Operator Code Replacement” for relevant limitations.
- `coder.replace` function — See `coder.replace` for relevant limitations.

Related Examples

- “Review and Test Code Replacements”
- “Specify Build Information for Replacement Code”
- “Replace MATLAB Functions with Custom Code Using `coder.replace`”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “Code Replacement Libraries”
- “Code Replacement Terminology”

Develop a Code Replacement Library

- 1 “Identify Code Replacement Requirements”
- 2 “Prepare for Code Replacement Library Development”
- 3 “Define Code Replacement Mappings”
- 4 “Specify Build Information for Replacement Code”
- 5 “Register Code Replacement Mappings”
- 6 “Review and Test Code Replacements”
- 7 “Deploy Code Replacement Library”

Related Examples

- “Replace MATLAB Functions with Custom Code Using `coder.replace`”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code You Can Replace from MATLAB Code”
- “Code Replacement Libraries”
- “Code Replacement Terminology”
- “Code Replacement Customization Limitations”

External Web Sites

- “Replacing Math Functions and Operators”

Quick Start Library Development

This example shows how to develop a code replacement library that includes an entry for generating replacement code for the math function `sin`. You use the Code Replacement Tool.

Prerequisites

To complete this example, install the following software:

- MATLAB
- MATLAB Coder
- Embedded Coder
- C compiler

For instructions on installing MathWorks products, see the “Installation Guide”. If you have installed MATLAB and want to see what other MathWorks products are installed, in the Command Window, enter `ver`.

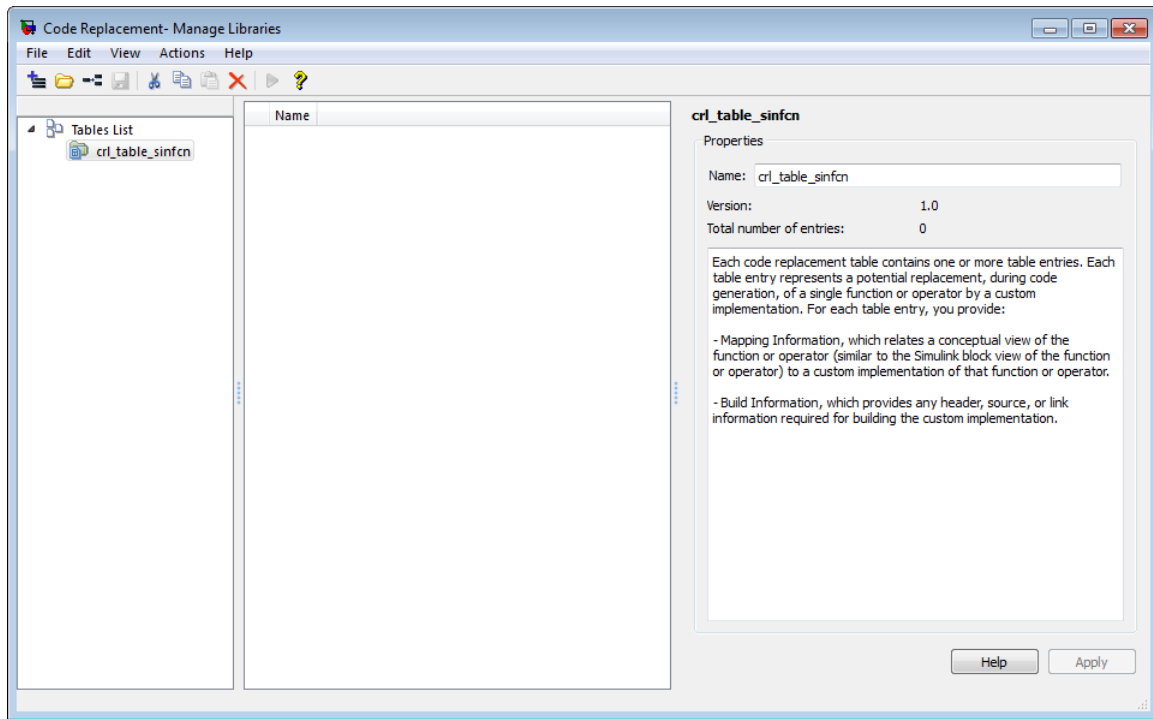
For a list of supported compilers, see http://www.mathworks.com/support/compilers/current_release/.

Open the Code Replacement Tool

- 1 Start a new MATLAB session.
- 2 Create or navigate (`cd`) to an empty folder.
- 3 At the command prompt, enter the `crtool` command. The Code Replacement Tool window opens.

Create Code Replacement Table

- 1 In the Code Replacement Tool window, select **File > New table**.
- 2 In the right pane, name the table `crl_table_sinfcn` and click **Apply**. Later, when you save the table, the tool saves it with the file name `crl_table_sinfcn.m`.



Create Table Entry

Create a table entry that maps a `sin` function with double input and double output to a custom implementation function.

- 1 In the left pane, select table `crl_table_sinfcn`. Then, select **File > New entry > Function**. The new entry appears in the middle pane, initially without a name.
- 2 In the middle pane, select the new entry.
- 3 In the right pane, on the **Mapping Information** tab, from the **Function** menu, select `sin`.
- 4 Leave **Algorithm** set to **Unspecified**, and leave parameters in the **Conceptual function** group set to default values.
- 5 In the **Replacement function** group, name the replacement function `sin_dbl`.
- 6 Leave the remaining parameters in the **Replacement function** group set to default values.

- 7 Click **Apply**. The tool updates the **Function signature preview** to reflect the specified replacement function name.
- 8 Scroll to the bottom of the **Mapping Information** tab and click **Validate entry**. The tool validates your entry.

The following figure shows the completed mapping information.

Mapping Information

Build Information

Function: sin

Entry information

Algorithm: Unspecified

Conceptual function

Used by code generation process for matching purposes

Conceptual arguments

y1
u1

Argument properties

Data type: double

Complex

Argument type: Scalar

Make conceptual and implementation argument types the same

Replacement function

Function prototype

Name: sin_dbl C++ namespace:

Function returns void

Function arguments

y1(return arg)
u1

Argument properties

Data type: double I/O type: OUTPUT

Const Pointer Complex

✕
+

Function signature preview

`double sin_dbl(double u1);`

Implementation attributes

Integer saturation mode: Unspecified Saturation

Rounding mode: Unspecified Rounding

Floor
Ceil
...

Allow expressions as inputs

Function modifies internal or global state

[Click here to add Build Information](#)

Validation

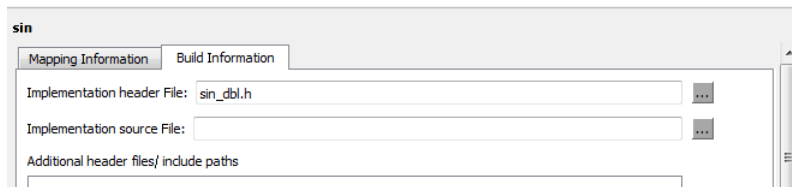
Validate entry Status: Validated

Help

Apply

Specify Build Information for Replacement Code

- 1 On the **Build Information** tab, for the **Implementation header file** parameter, enter `sin_dbl.h`.
- 2 Leave the remaining parameters set to default values.
- 3 Click **Apply**.



- 4 Optionally, you can revalidate the entry. Return to the **Mapping Information** tab and click **Validate entry**.

Create Another Table Entry

Create an entry that maps a `sin` function with `single` input and `double` output to a custom implementation function named `sin_sgl`. Create the entry by copying and pasting the `sin_dbl` entry.

- 1 In the middle pane, select the `sin_dbl` entry.
- 2 Select **Edit > Copy**
- 3 Select **Edit > Paste**
- 4 On the **Mapping Information** tab, in the **Conceptual function** section, set the data type of input argument `u1` to `single`.
- 5 In the **Replacement function** section, name the function `sin_sgl`. Set the data type of input argument `u1` to `single`.
- 6 Click **Apply**. Note the changes that appear for the **Function signature preview**.
- 7 On the **Build Information** tab, for the **Implementation header file** parameter, enter `sin_sgl.h`. Leave the remaining parameters set to default values and click **Apply**.

Validate the Code Replacement Table

- 1 Select **Actions > Validate table**.

- 2 If the tool reports errors, fix them, and rerun the validation. Repeat fixing and validating errors until the tool does not report errors. The following figure shows a validation report.

	Name	Implementation	NumIn	In1Type	In2Type	Out1Type	Out2Type	Priority
✓	sin	sin_dbl	1	double		double		100
✓	sin	sin_sgl	1	single		double		100

Save the Code Replacement Table

Save the code replacement table to a MATLAB file in your working folder. Select **File > Save table**. By default, the tool uses the table name to name the file. For this example, the tool saves the table in the file `cr1_table_sinfunc.m`.

Review the Code Replacement Table Definition

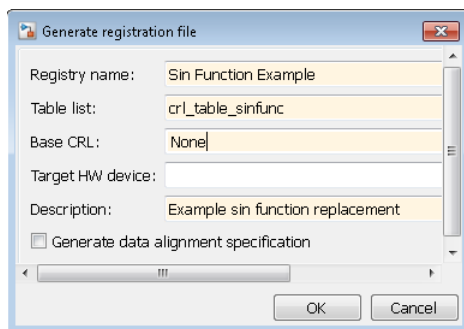
Consider reviewing the MATLAB code for your code replacement table definition. After using the tool to create an initial version of a table definition file, you can update, enhance, or copy the file in a text editor.

To review it, in MATLAB or another text editor, open the file `cr1_table_sinfunc.m`.

Generate a Registration File

Before you can use your code replacement table, you must register it as part of a code replacement library. Use the Code Replacement Tool to generate a registration file.

- 1 In the Code Replacement Tool, select **File > Generate registration file**.
- 2 In the **Generate registration file** dialog box, edit the dialog box fields to match the following figure, and then click **OK**.



- 3 In the **Select location** dialog box, specify a location for the registration file. The location must be on the MATLAB path or in the current working folder. Save the file. The tool saves the file as `rtwTargetInfo.m`.

Register the Code Replacement Table

At the command prompt, enter:

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

Review and Test Code Replacements

Apply your code replacement library. Verify that the code generator makes code replacements that you expect.

- 1 Check for errors. At the command line, invoke the table definition file. For example:

```
tbl = crl_table_sinfcn
```

```
tbl =
```

```
Tf1Table with properties:
```

```

        Version: '1.0'
    ReservedSymbols: []
StringResolutionMap: []
        AllEntries: [2x1 RTW.Tf1CFunctionEntry]
        EnableTrace: 1

```

If an error exists in the definition file, the invocation triggers a message to appear. Fix the error and try again.

- 2 Use the Code Replacement Viewer to check your code replacement entries. For example:

```
RTW.viewTf1('Sin Function Example')
```

In the viewer, select entries in your table and verify that the content is what you expect. The viewer can help you detect issues such as:

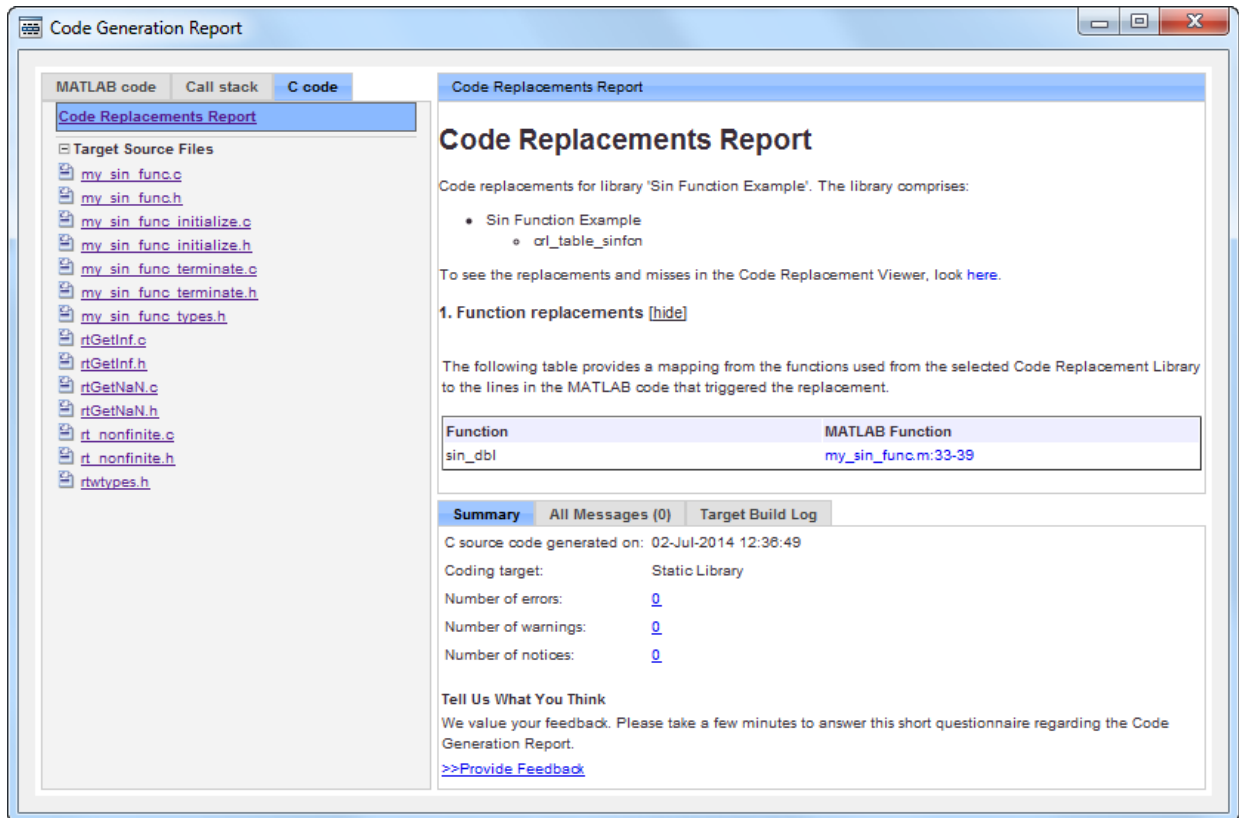
- Incorrect argument order.
- Conceptual argument names that do not match what is expected by the code generator.

- Incorrect priority settings.

- 3 Identify existing or create new MATLAB code that calls the `sin` function. For example:

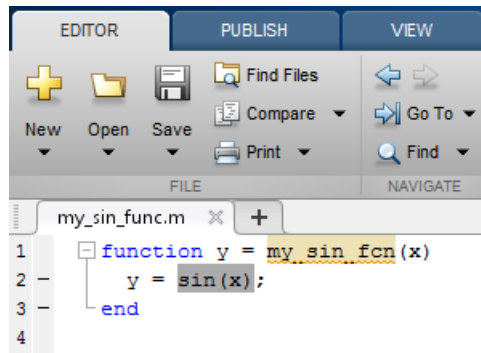
```
function y = my_sin_fnc(x)
    y = sin(x);
end
```

- 4 Create a project and name it `my_sin_func.prj`.
- 5 Add the function that includes a call to the `sin` function as an entry-point file and define the input types. For example, add `my_sin_func.m`.
- 6 Configure the code generator to use the code replacement library that you created. In the Project Settings dialog box, on the **Hardware** tab, set the **Code replacement library** parameter to **Sin Function Example**.
- 7 Configure the code generation report for code replacement analysis. On the **Debugging** tab, select:
 - **Always create a code generation report**
 - **Code replacements**
 - **Automatically launch a report if one is generated**
- 8 On the **Build** tab, select **Generate code only**. Before you build an executable, confirm that the code generator is replacing code as expected.
- 9 Click **Build** to generate C code.
- 10 Review code replacement results in the Code Replacements Report section of the code generation report.



The report indicates that the code generator found a match and applied the replacement code for the function `sin_dbl`.

- 11 Review the code replacements. In the report, under **Function replacements**, click the MATLAB function that triggered the replacement, `my_sin_func.m`. The MATLAB Editor opens and highlights the function call that triggers the code replacement.



Related Examples

- “Develop a Code Replacement Library”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code You Can Replace from MATLAB Code”
- “Code Replacement Libraries”
- “Code Replacement Terminology”
- “Code Replacement Customization Limitations”

External Web Sites

- “Replacing Math Functions and Operators”

Identify Code Replacement Requirements

In this section...

“Mapping Information Requirements” on page 22-37

“Build Information Requirements” on page 22-38

“Registration Information Requirements” on page 22-38

Mapping Information Requirements

- Are you defining a code replacement mapping for the first time?
- Are you updating code replacement entries in an existing library? Or, are you creating a new library?
- Are you rapid prototyping code replacements?
- Can you base your mappings on existing mappings?
- What type of code do you want to replace? Options include:
 - Math operation
 - Function
 - BLAS operation
 - CBLAS operation
 - Net slope fixed-point operation
 - Semaphore or mutex functions
- Do you want to change the inline or nonfinite behavior for functions?
- What specific functions and operations do you want to replace?
- What input and output arguments does the function or operator that you are replacing take? For each argument, what is the data type, complexity, and dimensionality?
- What does the prototype for your replacement code look like?
 - What is the replacement function name?
 - What are the input and output arguments?
 - Are there return values?

- What is the data type, complexity, and dimensionality of each argument and return value?

Build Information Requirements

- Does your replacement function implementation require a header file? If yes, specify the header file.
- If the replacement function implementation requires a header file, what is the path for that file?
- Is the source file for the replacement function in your working folder? If not, you can explicitly specify the source file name and extension. For example, if the file is required in the generated makefile or specified in a build information object, specify the source file.
- Does the replacement function use additional include files? If yes, what are they and what are the paths for those files?
- Does the replacement function use additional source files? If yes, what are they and what are the paths for those files?
- What compiler flags are required for compiling code that includes the replacement code?
- What linker flags are required for building an executable that includes the replacement code?
- Are the required header, source, and object files for building an executable that includes your replacement code in the working folder for your project? If not, before starting the build process, do you want the code generator to copy required files to the build folder?

Registration Information Requirements

- What do you want to name your code replacement library?
- What code replacement tables do you want to include in the library? What are the file names and paths for the tables?
- What is the purpose of the library? You can document the purpose as the library description.
- Does the library apply to specific hardware devices? If yes, what devices?
- Are you developing a hierarchy of code replacement libraries? Is the library that you are developing based (dependent) on another library? For example, you can specify a

general TI device library as the base library for a more specific TI C28x device library.

- Do you need to specify data alignment for the library? What data alignments are required? For each specification, what type of alignment is required and for what programming language?

Related Examples

- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code You Can Replace from MATLAB Code”
- “Code Replacement Libraries”
- “Code Replacement Terminology”
- “Code Replacement Customization Limitations”

Prepare for Code Replacement Library Development

After you identify your code replacement requirements, prepare for library development by reviewing this checklist:

- Get familiar with the library development process.
- Decide whether to define code replacement mappings and produce a registration file interactively with the Code Replacement Tool or programmatically.
- Identify or develop MATLAB code and Simulink models to test your code replacement library.
- Consider the hierarchy and organization of your library. A library can consist of multiple tables and each table can include multiple entries. How do you want to organize the library to optimize reuse of tables and entries? For example, a registration file can define code replacement tables organized in a hierarchy of code replacement libraries based on entries that increase in specificity:
 - Common entries
 - Entries for TI devices
 - Entries for TI C6xx devices
 - Entries specific to the TI C67x device
- If support files, such as header files, additional source files, and dynamically linked libraries are not in your current working folder, note their location. You need to specify the paths for such files.

Related Examples

- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code You Can Replace from MATLAB Code”
- “Code Replacement Libraries”
- “Code Replacement Terminology”

Define Code Replacement Mappings

In this section...

“Defining Code Replacement Mappings” on page 22-42

“Define Mappings Interactively with the Code Replacement Tool” on page 22-43

“Define Mappings Programmatically” on page 22-46

Defining Code Replacement Mappings

A code replacement mapping associates a conceptual representation of a function or operator that is familiar to the code generator with a custom implementation representation that specifies a C or C++ replacement function prototype. You capture a mapping as an entry in a code replacement table:

- Interactively, by using the Code Replacement Tool.
- Programmatically, by using a MATLAB programming interface.

The following table lists situations to help you decide when to use each approach.

Situation	Approach
Defining mappings for the first time.	Code Replacement Tool.
Rapid prototyping mappings.	Code Replacement Tool to quickly generate, register, and test mappings.
Developing a mapping as a template or starting point for defining similar mappings.	Code Replacement Tool to generate definition code that you can copy and modify.
Modifying a registration file, including copying and pasting content.	MATLAB Editor to update the programming interface directly.
Defining mappings that specify attributes not available from the Code Replacement Tool (for example, sets of algorithm parameters).	Programming interface.

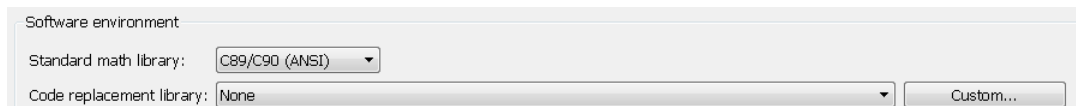
Define Mappings Interactively with the Code Replacement Tool

This example shows how to use the Code Replacement Tool to develop code replacement mappings. The tool is ideal for getting started with developing mappings, rapid prototyping, and developing a mapping to use as a starting point for defining similar mappings.

Open the Code Replacement Tool

Do one of the following:

- In the Command Window, enter the command `crtool`.
- In the Simulink Editor, open the Configuration Parameters dialog box and navigate to the **Code Generation > Interface >** pane. To the right of the **Code replacement library** parameter, click the **Custom**.



The **Custom** button is available only for ERT-based targets. Although an Embedded Coder license is not required to create a custom code replacement library, you must have an Embedded Coder license to use a such a library.

By default, the tool displays, left to right, a root pane, a list pane, and a dialog pane. You can manipulate the display:

- Drag boundaries to widen, narrow, shorten, or lengthen panes, and to resize table columns.
- Select **View > Show dialog pane** to hide or display the rightmost pane.
- Click a table column heading to sort the table based on contents of the selected column.
- Right-click a table column heading and select **Hide** to remove the column from the display. (You cannot hide the **Name** column.)

Create a Code Replacement Table

- 1 In the Code Replacement Tool window, select **File > New table**.

- 2 In the right pane, name the table and click **Apply**. Later, when you save the table, the tool uses the table name that you specify to name the file. For example, if you enter the name `my_sinfcn`, the tool names the file `my_sinfcn.m`.

Create Table Entries

Create one or more table entries. Each entry maps the conceptual representation of a function or operator to your implementation representation. The information that you enter depends on the type of entry you create. You must enter the following information:

- 1 In the left pane, select the table to which you want to add the entry.
- 2 Select **File > New entry > entry-type**, where **entry-type** is one of:
 - Math Operation
 - Function
 - BLAS Operation
 - CBLAS Operation
 - Net Slope Fixed Point Operation
 - Semaphore entry
 - Customization entry

The new entry appears in the middle pane, initially without a name.

- 3 In the middle pane, select the new entry.
- 4 In the right pane, on the **Mapping Information** tab, from the **Function** or **Operation** menu, select the function or operation that you want the code generator to replace. Regardless of the entry type, you must make a selection from this menu. Your selection determines what other information you specify.

With the exception of customization entries, you also specify information for your replacement function prototype. You can also specify implementation attributes, such as the rounding modes to apply.

- 5 If prompted, specify additional entry information that you want the code generator to use when searching for a match. For example, when you select an addition or subtraction operation, the tool prompts you to specify an algorithm (**Cast before operation** or **Cast after operation**).
- 6 Review the conceptual argument information that the tool populates for the function or operation. Conceptual input and output arguments represent arguments for

the function or operator being replaced. Conceptual arguments observe naming conventions ('y1', 'u1', 'u2', ...) and data types familiar to the code generator.

If you do not want the data types for your implementation to be the same as the conceptual argument types, clear the **Make the conceptual and implementation argument types the same** check box. For example, most ANSI-C functions operate on and return **double** data. Clear the check box if want to map a conceptual representation of the function, such as **sin**, which specifies an argument and return value of type **single** (**single sin(single)**), to an implementation representation that specifies an argument and return value of type **double** (**double sin(double)**). In this case, the code generator produces the following code:

```
y = (single) sin(u1);
```

If you select **Custom** for a function entry, specify only conceptual argument information.

- 7 Specify the name and argument information for your replacement function. As you enter the information and click **Apply**, the tool updates the **Function signature preview**.
- 8 Specify additional implementation attributes that apply. For example, depending on the type and name of the entry that you specify, the tool prompts you to specify:
 - Integer saturation mode
 - Rounding modes
 - Whether to allow inputs that include expressions
 - Whether a function modifies internal or global state
- 9 Click **Apply**.

Validate Tables and Entries

The Code Replacement Tool provides a way to validate the syntax of code replacement tables and table entries as you define them. If the tool finds validation errors, you can address them and retry the validation. Repeat the process until the tool does not report errors.

To...	Do...
Validate table entries	Select an entry, scroll to the bottom of the Mapping Information tab, and click Validate entry .

To...	Do...
	Alternatively, select one or more entries, right-click, and select Validate entries .
Validate a table	Select the table. Then, select Actions > Validate table .

Save a Table

When you save a table, the tool validates unvalidated content.

- 1 Select **File > Save table**.
- 2 In the Browse For Folder dialog box, specify a location and name for the file. Typically, you select a location on the MATLAB path. By default, the tool names the file using the name that you specify for the table with the extension `.m`.
- 3 Click **Save**.

Open and Modify Tables

After you save a code replacement table, to make changes:

- 1 Select **File > Open table**.
- 2 In the Import file dialog box, browse to the MATLAB file that contains the table.

Repeat the sequence to open and work on multiple tables.

If you open multiple tables, you can manage the tables together. For example, use the tool to:

- Create new table entries.
- Delete entries.
- Copy and paste or cut and paste information between tables.

Define Mappings Programmatically

This example shows how to define a code replacement mapping programmatically. The programming interface for defining code replacement table mappings is ideal for

- Modifying tables created with the Code Replacement Tool.
- Defining mappings for specialized entries that you cannot create with the Code Replacement Tool.

- Replicating and modifying similar entries and tables.

Create a Code Replacement Table

- 1 Create a table definition file that contains a function definition. For example:

```
function hTable = crl_table_sinfcn()
```

- 2 Within the function body, create the table by calling the function `RTW.Tf1Table`.

```
hTable = RTW.Tf1Table;
```

Create a Table Entry

For each function or operator that you want the code generator to replace, map a conceptual representation of the function or operator to an implementation representation as a table entry.

- 1 Within the body of a table definition file, create a code replacement entry object. Call one of the following functions:

Entry Type	Function
Math operation	<code>RTW.Tf1COperationEntry</code>
Function	<code>RTW.Tf1CFunctionEntry</code>
BLAS operation	<code>RTW.Tf1CBlasEntryGenerator</code>
CBLAS operation	<code>RTW.Tf1CBlasEntryGenerator</code>
Fixed-point addition and subtraction operations (support for <code>SlopesMustBeTheSame</code> and <code>ZeroNetBias</code> parameters)	<code>RTW.Tf1COperationEntryGenerator</code>
Net slope fixed-point operation	<code>RTW.Tf1COperationEntryGenerator_NetSlope</code>
Semaphore or mutex entry	<code>RTW.Tf1CSemaphoreEntry</code>
Custom function entry	<code>MyCustomFunctionEntry</code> (where <code>MyCustomFunctionEntry</code> is a class derived from <code>RTW.Tf1CFunctionEntryML</code>)
Custom operation entry	<code>MyCustomOperationEntry</code> (where <code>MyCustomOperationEntry</code> is a class derived from <code>RTW.Tf1COperationEntryML</code>)

For example:

```
hEnt = RTW.Tf1CFunctionEntry;
```

You can combine steps of creating the entry, setting entry parameters, creating conceptual and implementation arguments, and adding the entry to a table with a single function call to `registerCFunctionEntry`, `registerCPPFunctionEntry`, or `registerCPromotableMacroEntry` if you are creating an entry for a function and the function implementation meets the following criteria:

- Implementation argument names and order match the names and order of corresponding conceptual arguments.
- Input arguments are of the same type.
- The return and input argument names follow the code generator's default naming conventions:
 - Return argument is `y1`.
 - Input arguments are `u1`, `u2`, ..., `un`.

For example:

```
registerCFunctionEntry(hTable, 100, 1, 'sin', 'double', ...
    'sin_dbl', 'double', 'sin_dbl.h', '', '');
```

Set Entry Parameters

Set entry parameters, such as the priority, algorithm information, and implementation (replacement) function name. Call the function listed in the following table for the entry type that you created.

Entry Type	Function
Math operation	setTf1COperationEntryParameters
Function	setTf1CFunctionEntryParameters
BLAS operation	setTf1COperationEntryParameters
CBLAS operation	setTf1COperationEntryParameters
Fixed-point addition and subtraction operations where there is a many-to-one mapping, such as a mapping	setTf1COperationEntryParameters

Entry Type	Function
for a range of fixed-point types to the same replacement function (support for SlopesMustBeTheSame and ZeroNetBias parameters)	
Net slope fixed-point operation	setTf1COperationEntryParameters
Semaphore or mutex entry	setTf1CSemaphoreEntryParameters
Custom function entry	setTf1CFunctionEntryParameters
Custom operation entry	setTf1COperationEntryParameters

To see a list of the parameters that you can set, at the command line, create a new entry and omit the semicolon at the end of the command. For example:

```
hEnt = RTW.Tf1CFunctionEntry
```

```
hEnt =
```

```
Tf1CFunctionEntry with properties:
```

```

    Implementation: [1x1 RTW.CImplementation]
    SlopesMustBeTheSame: 0
    BiasMustBeTheSame: 0
    AlgorithmParams: []
    ImplType: 'FCN_IMPL_FUNCT'
    AdditionalHeaderFiles: {0x1 cell}
    AdditionalSourceFiles: {0x1 cell}
    OtherFiles: {0x1 cell}
    AdditionalIncludePaths: {0x1 cell}
    AdditionalSourcePaths: {0x1 cell}
    AdditionalLinkObjs: {0x1 cell}
    AdditionalLinkObjsPaths: {0x1 cell}
    AdditionalLinkFlags: {0x1 cell}
    AdditionalCompileFlags: {0x1 cell}
    SearchPaths: {0x1 cell}
    Key: ''
    Priority: 100
    ConceptualArgs: [0x1 handle]
    EntryInfo: []
    GenCallback: ''
    GenFileName: ''
    SaturationMode: 'RTW_SATURATE_UNSPECIFIED'
```

```

        RoundingModes: {'RTW_ROUND_UNSPECIFIED'}
    TypeConversionMode: 'RTW_EXPLICIT_CONVERSION'
        AcceptExprInput: 1
            SideEffects: 0
            UsageCount: 0
        RecordedUsageCount: 0
            Description: ''
    StoreFcnReturnInLocalVar: 0
        TraceManager: [1x1 RTW.Tf1TraceManager]

```

To see the implementation parameters, enter:

```
hEnt.Implementation
```

```
ans =
```

```
    CImplementation with properties:
```

```

        HeaderFile: ''
        SourceFile: ''
        HeaderPath: ''
        SourcePath: ''
        Return: []
    StructFieldMap: []
        Name: ''
        Arguments: [0x1 handle]
    ArgumentDescriptor: []

```

For example, to set entry parameters for the `sin` function and name your replacement function `sin_dbl`, use the following function call:

```
setTf1CFunctionEntryParameters(hEnt, ...
    'Key', 'sin', ...
    'ImplementationName', 'sin_dbl');
```

Create Conceptual Arguments

Create conceptual arguments and add them to the entry's array of conceptual arguments.

- Specify output arguments before input arguments.
- Specify argument names that comply with code generator argument naming conventions:
 - `y1` for a return argument
 - `u1, u2, ..., un` for input arguments

- Specify data types that are familiar to the code generator.
- The function signature, including argument naming, order, and attributes, must fulfill the signature match sought by function or operator callers.
- The code generator determines the size of the value for an argument with an unsized type, such as integer, based on hardware implementation configuration settings.

For each argument:

- 1 Identify whether the argument is for input or output, the name, and data type. If you do not know what arguments to specify for a supported function or operation, use the Code Replacement Tool to find them. For example, to find the conceptual arguments for the `sin` function, open the tool, create a table, create a function entry, and in the **Function** menu select `sin`.
- 2 Create and add the conceptual argument to an entry. You can choose a method from the methods listed in this table.

If	Then
<p>You want simpler code or want to explicitly specify whether the argument is scalar or nonscalar (vector or matrix).</p>	<p>Call the function <code>createAndAddConceptualArg</code>. For example:</p> <pre data-bbox="621 904 1184 1048">createAndAddConceptualArg(hEnt, ... 'RTW.Tf1ArgNumeric', ... 'Name', 'y1',... 'IOType', 'RTW_IO_OUTPUT',... 'DataTypeMode', 'double');</pre> <p>The second argument specifies whether the argument is scalar (<code>RTW.Tf1ArgNumeric</code> or <code>RTW.Tf1ArgMatrix</code>).</p>
<p>You want to create an argument based on a built-in argument definition (for example, scalar or nonscalar).</p>	<p>Call <code>getTf1ArgFromString</code> to create the argument. Then, call <code>addConceptualArg</code> to add the argument to the entry.</p> <pre data-bbox="621 1274 1299 1361">arg = getTf1ArgFromString(hEnt, 'y1', 'double'); arg.IOType = 'RTW_IO_OUTPUT'; addConceptualArg(hEnt, arg);</pre>

The following code shows the conceptual output and input argument definitions for the `sin` function:

```
% Conceptual Args
```

```

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

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

```

Create Implementation Arguments

Create implementation arguments for the C or C++ replacement function and add them to the entry.

- When replacing code, the code generator uses the argument names to determine how it passes data to the implementation function.
- For function replacements, the order of implementation argument names must match the order of the conceptual argument names.
- For operator replacements, the order of implementation argument names do not have to match the order of the conceptual argument names. For example, for an operator replacement for addition, $y1=u1+u2$, the conceptual arguments are $y1$, $u1$, and $u2$, in that order. If the signature of your implementation function is `t myAdd(t u2, t u1)`, where `t` is a valid C type, based on the argument name matches, the code generator passes the value of the first conceptual argument, $u1$, to the second implementation argument of `myAdd`. The code generator passes the value of the second conceptual argument, $u2$, to the first implementation argument of `myAdd`.
- For operator replacements, you can remap operator output arguments to implementation function input arguments.

For each argument:

- 1 Identify whether the argument is for input or output, the name, and the data type.
- 2 Create and add the implementation argument to an entry. You can choose a method from the methods listed in this table.

If	Then
You want to populate implementation arguments as copies of previously created matching conceptual arguments	Call the function <code>copyConceptualArgsToImplementation</code> . For example: <code>copyConceptualArgsToImplementation(hEnt);</code>

If	Then
You want to create and add implementation arguments individually, or vary argument attributes, while maintaining conceptual argument order	<p>Call functions <code>createAndSetCImplementationReturn</code> and <code>createAndAddImplementationArg</code>. For example:</p> <pre>createAndSetCImplementationReturn(hEnt, '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);</pre>

If	Then
<p>You want to minimize the amount of code, or specify constant arguments to pass to the implementation function</p>	<p>Create the argument with a call to the function <code>getTf1ArgFromString</code>. Then, use the convenience method <code>setReturn</code> or <code>addArgument</code> to specify whether an argument is a return value or argument and to add the argument to the entry's array of implementation arguments. For example:</p> <pre>arg = getTf1ArgFromString(hEnt, 'y1', 'double'); arg.IOType = 'RTW_IO_OUTPUT'; hEnt.Implementation.setReturn(arg);</pre> <p>arg = getTf1ArgFromString(hEnt, 'u1', 'double'); hEnt.Implementation.addArgument(arg);</p> <p>The following call to <code>getTf1ArgFromString</code> passes the constant 0 to argument u2:</p> <pre>arg = getTf1ArgFromString(hEnt, 'u2', 'int16', 0); hEnt.Implementation.addArgument(arg);</pre> <p>For semaphore and mutex entries, use the functions <code>getTf1DWorkFromString</code> and <code>addDWorkArg</code> to create and add a DWork argument to the entry. Then create implementation arguments as shown above with <code>getTf1ArgFromString</code> and the convenience methods <code>setReturn</code> and <code>addArgument</code>. For example:</p> <pre>arg = getTf1DWorkFromString('d1', 'void*'); hEnt.addDWorkArg(arg);</pre> <pre>arg = hEnt.getTf1ArgFromString('y1', 'void'); arg.IOType = 'RTW_IO_OUTPUT'; hEnt.Implementation.setReturn(arg);</pre> <pre>arg = hEnt.getTf1ArgFromString('u1', 'integer'); hEnt.Implementation.addArgument(arg);</pre> <pre>arg = hEnt.getTf1ArgFromString('d1', 'void**'); hEnt.Implementation.addArgument(arg);</pre>

The following code shows the implementation output and input argument definitions for the `sin` function:

```
% Implementation Args

arg = hEnt.getTf1ArgFromString('y1', 'double');
arg.IOType = 'RTW_IO_OUTPUT';
hEnt.Implementation.setReturn(arg);

arg = hEnt.getTf1ArgFromString('u1', 'double');
hEnt.Implementation.addArgument(arg);
```

Add an Entry to a Table

Add an entry to a code replacement table by calling the function `addEntry`.

```
addEntry(hTable, hEnt);
```

Validate an Entry

After you create or modify a code replacement table entry, validate it by invoking it at the MATLAB command line. For example:

```
hTbl = crl_table_sinfcn

hTbl =

RTW.Tf1Table
  Version: '1.0'
 AllEntries: [2x1 RTW.Tf1CFunctionEntry]
ReservedSymbols: []
StringResolutionMap: []
```

If the table includes errors, MATLAB reports them. The following examples shows how MATLAB reports a typo in a data type name:

```
hTbl = crl_table_sinfcn

??? RTW_CORE:tf1:Tf1Table: Unsupported data type, 'dooble'.

Error in ==> crl_table_sinfcn at 7
hTable.registerCFunctionEntry(100, 1, 'sin', 'dooble', 'sin_dbl', ...
```


Save a Table

Save the table definition file. Use the name of the table definition function to name the file, for example, `crl_table_sinfcn.m`.

Related Examples

- “Identify Code Replacement Requirements”
- “Prepare for Code Replacement Library Development”
- “Specify Build Information for Replacement Code”
- “Register Code Replacement Mappings”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”
- “Math Function Code Replacement”
- “Memory Function Code Replacement”
- “Specify In-Place Code Replacement”
- “Replace MATLAB Functions with Custom Code Using `coder.replace`”
- “Reserved Identifiers and Code Replacement”
- “Customize Matching and Replacement Process for Functions”
- “Scalar Operator Code Replacement”
- “Addition and Subtraction Operator Code Replacement”
- “Small Matrix Operation to Processor Code Replacement”
- “Matrix Multiplication Operation to MathWorks BLAS Code Replacement”
- “Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement”
- “Remap Operator Output to Function Input”
- “Customize Matching and Replacement Process for Operators”
- “Fixed-Point Operator Code Replacement”
- “Binary-Point-Only Scaling Code Replacement”
- “Slope Bias Scaling Code Replacement”
- “Net Slope Scaling Code Replacement”
- “Equal Slope and Zero Net Bias Code Replacement”
- “Data Type Conversions (Casts) and Operator Code Replacement”

- “Shift Left Operations and Code Replacement”

More About

- “Code You Can Replace from MATLAB Code”
- “Code Replacement Libraries”
- “Code Replacement Terminology”
- “Code Replacement Customization Limitations”

External Web Sites

- “Replacing Math Functions and Operators”

Specify Build Information for Replacement Code

In this section...

“Specifying Build Information” on page 22-58

“Specify Build Information Interactively with the Code Replacement Tool” on page 22-59

“Specify Build Information Programmatically” on page 22-61

Specifying Build Information

A code replacement table entry can specify build information for the code generator to use when replacing code for a match. The build information can include:

- Paths and file names for header files
- Paths and file names for source files
- Paths and file names for object files
- Compile flags
- Link flags

Add build information to an entry:

- Interactively, by using the Code Replacement Tool.
- Programmatically, by using a MATLAB programming interface.

The following table lists situations to help you decide when to use each approach.

Situation	Approach
Creating code replacement entries for the first time.	Code Replacement Tool.
You used the Code Replacement Tool to create the entries for which the build information applies.	Code Replacement Tool to quickly specify the build information.
Rapid prototyping entries.	Code Replacement Tool to quickly generate, register, and test entries.
Developing an entry to use as a template or starting point for defining similar entries.	Code Replacement Tool to generate entry code that you can copy and modify.

Situation	Approach
Modifying existing mappings.	MATLAB Editor to update the programming interface directly.

As you specify build information, consider the following information:

- If you use the programming interface, paths that you specify can include tokens. A token is a variable defined as a string or cell array of strings in the MATLAB workspace that you enclose with dollar signs (*\$variable\$*). The code generator evaluates and replaces a token with the defined value. For example, consider the path *\$myfolder\$*\folder1, where *myfolder* is a string variable defined in the MATLAB workspace as 'd:\work\source\module1'. The code generator generates the custom path as d:\work\source\module1\folder1.
- If an entry uses header, source, or object files that reside in a folder other than the build folder, you must copy the files to the build folder before the code generator can build the code. The Code Replacement Tool and programming information provide a way to specify this copy operation.

Specify Build Information Interactively with the Code Replacement Tool

The Code Replacement Tool provides a quick, easy way for you to specify build information for code replacement table entries. It is ideal for getting started with defining a table entry, rapid prototyping, and developing table entries to use as a starting point for defining similar mappings.

- 1 Determine the information that you must specify.
- 2 Open the Code Replacement Tool.
- 3 Select the code replacement table entry for which you want to specify the build information. In the left pane, select the table that contains the entry. In the middle pane, select the entry that you want to modify.
- 4 In the right pane, select the **Build Information** tab.
- 5 On the **Build Information** tab, specify your build information.

Field	Specify
Implementation header file	File name and extension for the header file the code generator needs to generate the replacement code. For example, <code>sin_db1.h</code> .

Field	Specify
Implementation source file	File name and extension for the C or C++ source file the code generator needs to generate the replacement code. For example, <code>sin_dbl.c</code> .
Additional header files/include paths	Paths and file names for additional header files the code generator needs to generate the replacement code. For example, <code>C:\libs\headerFiles</code> and <code>C:\libs\headerFiles\common.h</code> .
Additional source files/ paths	Paths and file names for additional source files the code generator needs to generate the replacement code. For example, <code>C:\libs\srcFiles</code> and <code>C:\libs\srcFiles\common.c</code> .
Additional object files/ paths	Paths and file names for additional object files the linker needs to build the replacement code. For example, <code>C:\libs\objFiles</code> and <code>C:\libs\objFiles\common.obj</code> .
Additional link flags	Flags the linker needs to generate an executable file for the replacement code.
Additional compile flags	Flags the compiler needs to generate object code for the replacement code.
Copy files to build directory	Whether the code generator needs to copy header, source, or object files from an external location to the build folder before building the replacement code.

- 6 Click **Apply**.
- 7 Select the **Mapping Information** tab. Scroll to the bottom of that table and click **Validate entry**. The tool validates the changes that you made to the entry.
- 8 Save the table that includes the entry that you just modified.

Specify Build Information Programmatically

The programming interface for specifying build information for a code replacement entry is ideal for:

- Modifying entries created with the Code Replacement Tool.
- Replicating and then modifying similar entries and tables.

The basic workflow for specifying build information programmatically is:

- 1 Identify or create the code replacement entry that you want to specify the build information.
- 2 Determine what information to specify.
- 3 Specify your build information.

Specify	Action
Implementation header file	<p>Use one of the following:</p> <ul style="list-style-type: none"> • Set properties <code>ImplementationHeaderFile</code> and <code>ImplementationHeaderPath</code> in a call to <code>setTf1CFunctionEntryParameters</code>, <code>setTf1COperationEntryParameters</code>, or <code>setTf1CSemaphoreEntryParameters</code>. For example: <pre>setTf1CFunctionEntryParameters(hEnt, ... 'ImplementationHeaderFile', 'sin_dbl.h', ... 'ImplementationHeaderPath', 'D:/lib/headerFiles' 'Key', 'sin', ... 'ImplementationName', 'sin_dbl');</pre> <ul style="list-style-type: none"> • Set argument <code>headerFile</code> in a call to <code>registerCFunctionEntry</code>, <code>registerCPPFunctionEntry</code>, or <code>registerCPromotableMacroEntry</code>
Implementation source file	<p>Set properties <code>ImplementationSourceFile</code> and <code>ImplementationSourcePath</code> in a call to <code>setTf1CFunctionEntryParameters</code>, <code>setTf1COperationEntryParameters</code>, or <code>setTf1CSemaphoreEntryParameters</code>. For example:</p> <pre>setTf1CFunctionEntryParameters(hEnt, ... 'ImplementationHeaderFile', 'sin_dbl.c', ... 'ImplementationHeaderPath', 'D:/lib/sourceFiles' 'Key', 'sin', ... 'ImplementationName', 'sin_dbl');</pre>

Specify	Action
Additional header files/include paths	<p>For each file, specify the file name and path in calls to the functions <code>addAdditionalHeaderFile</code> and <code>addAdditionalIncludePath</code>. For example:</p> <pre>libdir = fullfile('\$MATLAB_ROOT','..','..','lib'); hEnt = RTW.Tf1CFunctionEntry; addAdditionalHeaderFile(hEnt, 'common.h'); addAdditionalIncludePath(hEnt, fullfile(libdir, 'include'));</pre>
Additional source files/paths	<p>For each file, specify the file name and path in calls to the functions <code>addAdditionalSourceFile</code> and <code>addAdditionalSourcePath</code>. For example:</p> <pre>libdir = fullfile('\$MATLAB_ROOT','..','..','lib'); hEnt = RTW.Tf1CFunctionEntry; addAdditionalSourceFile(hEnt, 'common.c'); addAdditionalSourcePath(hEnt, fullfile(libdir, 'src'));</pre>
Additional object files/paths	<p>For each file, specify the file name and path in calls to the functions <code>addAdditionalLinkObj</code> and <code>addAdditionalLinkObjPath</code>. For example:</p> <pre>libdir = fullfile('\$MATLAB_ROOT','..','..','lib'); hEnt = RTW.Tf1CFunctionEntry; addAdditionalLinkObj(hEnt, 'sin.o'); addAdditionalLinkObjPath(hEnt, fullfile(libdir, 'bin'));</pre>
Compile flags	<p>Set the entry property <code>AdditionalCompileFlags</code> to a cell array of strings representing the required compile flags. For example:</p> <pre>hEnt = RTW.Tf1CFunctionEntry; hEnt.AdditionalCompileFlags = {'-Zi -Wall', '-O3'};</pre>

Specify	Action
Link flags	<p>Set the entry property <code>AdditionalLinkFlags</code> to a cell array of strings representing the required link flags. For example:</p> <pre>hEnt = RTW.Tf1CFunctionEntry; hEnt.AdditionalCompileFlags = {'-MD -Gy', '-T'};</pre>
Copy of header, source, and link object files that are specified in an entry to the build folder	<p>Use one of the following:</p> <ul style="list-style-type: none"> Set property <code>GenCallback</code> to <code>'RTW.copyFileToBuildDir'</code> in a call to <code>setTf1CFunctionEntryParameters</code>, <code>setTf1COperationEntryParameters</code>, or <code>setTf1CSemaphoreEntryParameters</code>. For example: <pre>setTf1CFunctionEntryParameters(hEnt, ... 'ImplementationHeaderFile', 'sin_dbl.h', ... 'ImplementationHeaderPath', 'D:/lib/headerFiles' 'Key', 'sin', ... 'ImplementationName', 'sin_dbl' 'GenCallback', 'RTW.copyFileToBuildDir');</pre> Set argument <code>genCallback</code> in a call to <code>registerCFunctionEntry</code>, <code>registerCPPFunctionEntry</code>, or <code>registerCPromotableMacroEntry</code> to <code>'RTW.copyFileToBuildDir'</code>. <p>After generating the C or C++ code, the code generator calls the function <code>RTW.copyFileToBuildDir</code>. The specified files are then available for the remaining steps of the build process.</p>

4 Save the table that includes the entry that you added or modified.

The following example defines a table entry for an optimized multiplication function that takes signed 32-bit integers and returns a signed 32-bit integer, taking saturation into account. Multiplications in the generated code are replaced with calls to the optimized function. The optimized function does not reside in the build folder. For the code generator to access the files, they must be copied into the build folder to be compiled and linked into the application.

The table entry specifies the source and header file names and paths. To request the copy operation, the table entry sets the `genCallback` to `'RTW.copyFileToBuildDir'` in the call to the `setTf1COperationEntryParameters` function. In this example,

the header file `s32_mul.h` contains an inlined function that invokes assembly functions contained in `s32_mul.s`. If a match occurs for the table entry, the function `RTW.copyFileToBuildDir` copies the specified source and header files to the build folder.

```
function hTable = make_my_crl_table

hTable = RTW.Tf1Table;

op_entry = RTW.Tf1COperationEntry;
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_MUL', ...
    'Priority', 100, ...
    'SaturationMode', 'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes', {'RTW_ROUND_UNSPECIFIED'}, ...
    'ImplementationName', 's32_mul_s32_s32_sat', ...
    'ImplementationHeaderFile', 's32_mul.h', ...
    'ImplementationSourceFile', 's32_mul.s', ...
    'ImplementationHeaderPath', {fullfile('${MATLAB_ROOT}','crl')}, ...
    'ImplementationSourcePath', {fullfile('${MATLAB_ROOT}','crl')}, ...
    'GenCallback', 'RTW.copyFileToBuildDir');
.
.
.
addEntry(hTable, op_entry);
```

The following example uses the functions `addAdditionalHeaderFile`, `addAdditionalIncludePath`, `addAdditionalSourceFile`, `addAdditionalSourcePath`, `addAdditionalLinkObj`, and `addAdditionalLinkObjPath` with the code generation callback function `RTW.copyFileToBuildDir`.

```
hTable = RTW.Tf1Table;

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

op_entry = RTW.Tf1COperationEntry;
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_ADD', ...
    'Priority', 90, ...
    'SaturationMode', 'RTW_SATURATE_UNSPECIFIED', ...
    'RoundingModes', {'RTW_ROUND_UNSPECIFIED'}, ...
    'ImplementationName', 's32_add_s32_s32', ...
    'ImplementationHeaderFile', 's32_add_s32_s32.h', ...
    'ImplementationSourceFile', 's32_add_s32_s32.c'...
    'GenCallback', 'RTW.copyFileToBuildDir');

addAdditionalHeaderFile(op_entry, 'all_additions.h');
addAdditionalIncludePath(op_entry, fullfile(libdir, 'include'));
addAdditionalSourceFile(op_entry, 'all_additions.c');
addAdditionalSourcePath(op_entry, fullfile(libdir, 'src'));
```

```
addAdditionalLinkObj(op_entry, 'addition.o');  
addAdditionalLinkObjPath(op_entry, fullfile(libdir, 'bin'));  
.  
.  
addEntry(hTable, op_entry);
```

Related Examples

- “Identify Code Replacement Requirements”
- “Prepare for Code Replacement Library Development”
- “Define Code Replacement Mappings”
- “Register Code Replacement Mappings”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “What Is Code Replacement Customization?”
- “Code Replacement Libraries”
- “Code Replacement Terminology”
- “Code Replacement Customization Limitations”

External Web Sites

- “Replacing Math Functions and Operators”

Register Code Replacement Mappings

In this section...
“Code Replacement Library Registration” on page 22-55
“Create Registration File Interactively with the Code Replacement Tool” on page 22-56
“Create Registration File Programmatically” on page 22-58
“Register a Code Replacement Library” on page 22-60
“Registration Files That Define Multiple Code Replacement Libraries” on page 22-60
“Registration Files That Define Code Replacement Library Hierarchies” on page 22-61

Code Replacement Library Registration

After you define code replacement entries in a code replacement table, you can include the table in a code replacement library that you register with the code generator. When registered, a library appears in the list of available code replacement libraries that you can choose from when configuring the code generator.

Register a code replacement table as a code replacement library:

- Interactively, by using the Code Replacement Tool
- Programmatically, by using a MATLAB programming interface

The following table lists situations when you might consider one approach over the other.

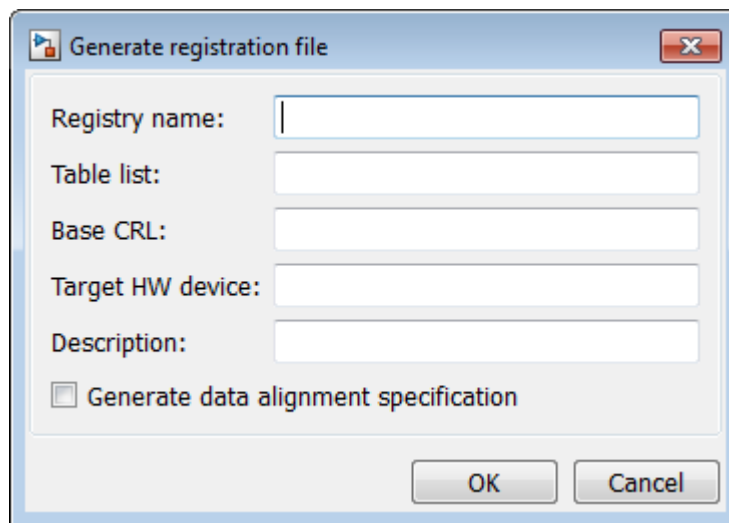
If...	Then...
Registering a code replacement table for the first time	Use the Code Replacement Tool.
You used the Code Replacement Tool to create the table	Use the Code Replacement Tool to quickly register the table.
Rapid prototyping code replacement	Use the Code Replacement Tool to quickly generate, register, and test entries.
Creating registration file to use as a template or starting point for defining similar registration files	Use the Code Replacement Tool to generate code that you can copy and modify.

If...	Then...
Modifying existing registration files	Use the MATLAB Editor to update the registration file.
Defining multiple code replacement libraries in one registration file	Use the MATLAB Editor to create a new or extend an existing registration file.
Defining code replacement library hierarchy in a registration file	Use the MATLAB Editor to create a new or extend an existing registration file.

Create Registration File Interactively with the Code Replacement Tool

The Code Replacement tool provides a quick, easy way for you to create a registration file for a code replacement table. It is ideal for getting started, rapid prototyping, and generating a registration file that you want to use as a starting point for similar registrations.

- 1 After you validate and save a code replacement table, select **File > Generate registration file** to open the **Generate registration file** dialog box.



- 2 Enter the registration information. Minimally, specify:

For...	Specify...
Registry name	String naming the code replacement library. For example, <code>Sin Function Example</code> .
Table list	<p>Strings naming one or more code replacement tables to include in the library. Specify each table as one of the following:</p> <ul style="list-style-type: none"> • Name of a table file on the MATLAB search path • Absolute path to a table file • Path to a table file relative to <code>\$(MATLAB_ROOT)</code> <p>You can specify multiple tables. If you do, separate the table specifications with a comma. For example:</p> <pre>crl_table_sinfcn, c:/work_crl/crl_table_muldiv</pre> <p>See “Registration Files That Define Multiple Code Replacement Libraries” on page 21-71 for examples of each type of table specification.</p>

Optionally, you can specify:

For...	Specify...
Description	Text string that describes the purpose and content of the library.
Target HW device	Strings naming one or more hardware devices the code replacement library supports. Separate names with a comma. To support all device types, enter an asterisk (*). For example, <code>TI C28x, TI C62x</code> .
Base CRL	String naming a code replacement library that you want to serve as a base library for the library you are registering. Use this field to specify library hierarchies. For example, you can specify a general <code>TI device library</code> as the base library for a more specific <code>TI C28x device library</code> .

For...	Specify...
Generate data alignment specification	Flag that enables data alignment specification.

Create Registration File Programmatically

The programming interface for creating a registration file for a code replacement table is ideal for:

- Modifying registration files created with the Code Replacement Tool
- Replicating and modifying similar registration files
- Defining multiple code replacement libraries in one registration file

The basic workflow for creating a registration file programmatically consists of the following steps:

- 1 Define an `rtwTargetInfo` function. The code generator recognizes this function as a customization file. The function definition must include at least the following content:

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@loc_register_tf1);

function this = loc_register_tf1

this(1) = RTW.Tf1Registry;
this(1).Name = 'crl-name';
this(1).TableList = {'table',...};
```

For...	Replace...
<code>this(1).Name = 'crl-name';</code>	<code>crl-name</code> with a string naming the code replacement library. For example, <code>Sin Function Example</code> .
<code>this(1).TableList = {'table',...};</code>	<code>table</code> with a string that identifies the code replacement table that contains your code replacement entries. Specify a table as one of the following:

For...	Replace...
	<ul style="list-style-type: none"> • Name of a table file on the MATLAB search path • Absolute path to a table file • Path to a table file relative to \$(MATLAB_ROOT) <p>You can specify multiple tables. If you do, separate the table specifications with commas.</p> <p>See “Registration Files That Define Multiple Code Replacement Libraries” on page 21-71 for examples of each type of table specification.</p>

Optionally, you can specify:

For...	Replace...
<code>this(1).Description = 'text'</code>	<i>text</i> with a string that describes the purpose and content of the library.
<code>this(1).TargetHWDeviceType = {'device-type', ...}</code>	<i>device-type</i> with a string that names a hardware device the code replacement library supports. You can specify multiple device types. Separate device types with a comma. For example, TI C28x, TI C62x. To support all device types, enter an asterisk (*).
<code>this(1).BaseTfl = 'base-lib'</code>	<i>base-lib</i> with a string that names a code replacement library that you want to serve as a base library for the library you are registering. Use this field to specify library hierarchies. For example, you can specify a general TI device library as the base library for a TI C28x device library.
	See “Registration Files That Define Code Replacement Library Hierarchies” on page 21-72 for an example.

For example:

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@loc_register_tfl);

function this = loc_register_tfl

this(1) = RTW.TflRegistry;
this(1).Name = 'Sin Function Example';
this(1).TableList = {'crl_table_sinfcn'};
this(1).TargetHWDeviceType = {'*'};
this(1).Description = 'Example - sin function replacement';
```

- 2 Save the file with the name `rtwTargetInfo.m`.
- 3 Place the file on the MATLAB path. When the file is on the MATLAB path, the code generator reads the file after starting and applies the customizations during the current MATLAB session.

Register a Code Replacement Library

Before you can use the code replacement tables defined in a registration file, you must refresh Simulink customizations within the current MATLAB session. To initiate a refresh, enter the following command:

```
sl_refresh_customizations
```

Registration Files That Define Multiple Code Replacement Libraries

Use the programming interface to create a registration file that defines multiple code replacement libraries. The following example defines multiple code replacement libraries. The `TableList` fields specify code replacement tables that reside at different locations. The tables reside on the MATLAB search path or at locations specified using path strings.

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@locCrlRegFcn);

function thisCrl = locCrlRegFcn

% Register a code replacement library for use with model: rtwdemo_crladdsub
thisCrl(1) = RTW.TflRegistry;
```



```

thisCr1(1).Name = 'Addition & Subtraction Examples';
thisCr1(1).Description = 'Example of addition/subtraction op replacement';
thisCr1(1).TableList = {'crl_table_addsub'};
thisCr1(1).TargetHWDeviceType = {'*'};

% Register a code replacement library for use with model: rtwdemo_crlmuldiv
thisCr1(2) = RTW.TflRegistry;
thisCr1(2).Name = 'Multiplication & Division Examples';
thisCr1(2).Description = 'Example of mult/div op repl for built-in integers';
thisCr1(2).TableList = {'c:/work_crl/crl_table_muldiv'};
thisCr1(2).TargetHWDeviceType = {'*'};

% Register a code replacement library for use with model: rtwdemo_crlfixpt
thisCr1(3) = RTW.TflRegistry;
thisCr1(3).Name = 'Fixed-Point Examples';
thisCr1(3).Description = 'Example of fixed-point operator replacement';
thisCr1(3).TableList = {fullfile('$MATLAB_ROOT', ...
    'toolbox', 'rtw', 'rtwdemos', 'crl_demo', 'crl_table_fixpt')};
thisCr1(3).TargetHWDeviceType = {'*'};

```

Registration Files That Define Code Replacement Library Hierarchies

Using the programming interface, you can organize multiple code replacement libraries in a hierarchy. The following example shows a registration file that defines four code replacement tables organized in a hierarchy of four code replacement libraries. The tables include entries that increase in specificity: common entries, entries for TI devices, entries for TI C6xx devices, and entries specific to the TI C67x device.

```

function rtwTargetInfo(cm)

cm.registerTargetInfo(@locCrlRegFcn);

function thisCrl = locCrlRegFcn

% Register a code replacement library that includes common entries
thisCrl(1) = RTW.TflRegistry;
thisCrl(1).Name = 'Common Replacements';
thisCrl(1).Description = 'Common code replacement entries shared by other libraries';
thisCrl(1).TableList = {'crl_table_general'};
thisCrl(1).TargetHWDeviceType = {'*'};

% Register a code replacement library for TI devices
thisCrl(2) = RTW.TflRegistry;
thisCrl(2).Name = 'TI Device Replacements';
thisCrl(2).Description = 'Code replacement entries shared across TI devices';
thisCrl(2).TableList = {'crl_table_TI_devices'};
thisCrl(2).TargetHWDeviceType = {'TI C28x', 'TI C55x', 'TI C62x', 'TI C64x', 'TI 67x'};
thisCrl(1).BaseTfl = 'Common Replacements'

% Register a code replacement library for TI c6xx devices
thisCrl(3) = RTW.TflRegistry;
thisCrl(3).Name = 'TI c6xx Device Replacements';
thisCrl(3).Description = 'Code replacement entries shared across TI C6xx devices';
thisCrl(3).TableList = {'crl_table_TIC6xx_devices'};
thisCrl(3).TargetHWDeviceType = {'TI C62x', 'TI C64x', 'TI 67x'};

```

```
% Register a code replacement library for the TI c67x device
thisCrl(3) = RTW.TflRegistry;
thisCrl(3).Name = 'TI c67x Device Replacements';
thisCrl(3).Description = 'Code replacement entries for the TI C67x device';
thisCrl(3).TableList = {'crl_table_TIC67x_device'};
thisCrl(3).TargetHWDeviceType = {'TI 67x'};
```

Related Examples

- “Define Code Replacement Mappings”
- “Specify Build Information for Replacement Code”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”
- “Deploy Code Replacement Library”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code Replacement Libraries”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Review and Test Code Replacements

In this section...

“Code Replacement Table Validation” on page 22-63

“Validate a Table Definition File” on page 22-63

“Review Table Content” on page 22-64

“Verify Library Registration and Content” on page 22-65

“Review and Trace Code Replacements Generated Using a Code Replacement Library” on page 22-66

“Determine Why Code Replacement Functions Are Not Used” on page 22-68

Code Replacement Table Validation

After you create or modify a code replacement table, use the following techniques to examine and validate the table and its entries.

- Invoke the table definition file
- Use the Code Replacement Viewer to examine libraries, tables, and entries
- Trace code replacements from the source for which you applied the code replacement library
- Examine code replacement cache hits and misses logged during code generation

Validate a Table Definition File

After you create or modify a code replacement table definition file, validate it. At the MATLAB command line, specify the name of the table in a call to the `isvalid` function. For example,

```
isvalid(cr1_table_sinfcn)
ans =
     1
```

MATLAB displays errors that it finds. In the following example, MATLAB detects a typo in a data type name.

```
isvalid(cr1_table_sinfcn)
??? RTW_CORE:tf1:Tf1Table: Unsupported data type, 'dooble'.
```

```
Error in ==> cr1_table_sinfcn at 7
hTable.registerCFFunctionEntry(100, 1, 'sin', 'dooble', 'sin_dbl', ...
```

Review Table Content

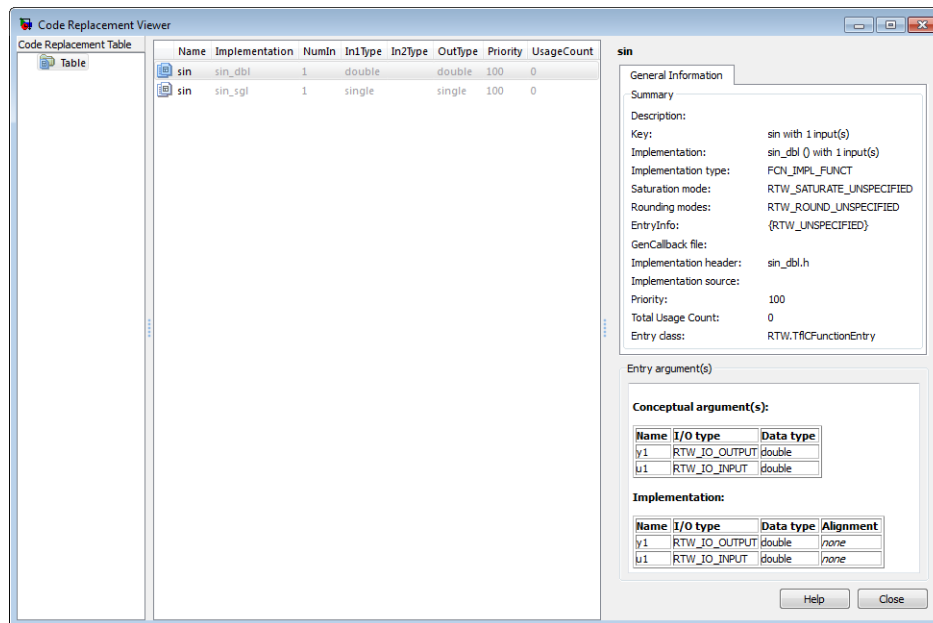
After you create or modify a code replacement table, use the Code Replacement Viewer to review the table and its entries.

- 1 Open the viewer. At the MATLAB command line, enter the following command, where *table* is the name of the code replacement table you want to review:

```
RTW.viewTf1(table)
```

For example,

```
RTW.viewTf1(cr1_table_sinfcn)
```



- 2 Select entries in the table and verify that the display of the contents meets your expectations. For example:

- 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.

Verify Library Registration and Content

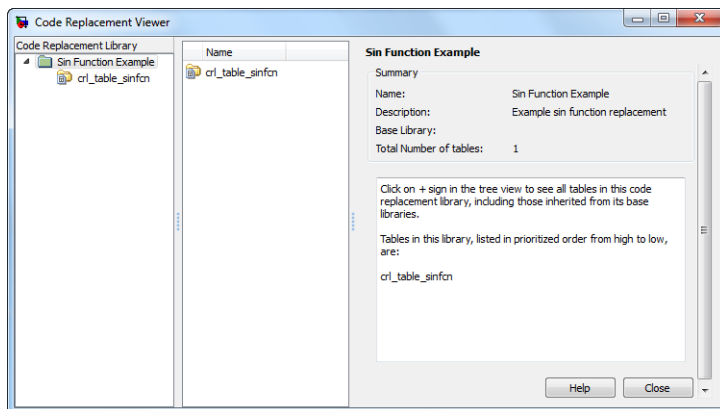
After you register a library that includes your code replacement table, use the Code Replacement Viewer to verify that the library is registered and examine the library and the tables it contains.

- 1 Open the viewer. At the MATLAB command line, enter the following command, where *library* is the name of the code replacement library you want to review:

```
RTW.viewTf1('library')
```

For example:

```
RTW.viewTf1('Sin Function Example')
```



If the tool does not display your library,

- The code replacement library registration file, `rtwTargetInfo.m`, might contain an error.

- You might need to refresh the library registration information. Initiate a refresh with the following command:

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

- 2 Select your library and examine and compare its tables. Common problems that you can detect include:
 - Relative order of tables in the library is not correct (by default, the tool displays tables in search order)
 - Table entry problems as listed in the previous section

Review and Trace Code Replacements Generated Using a Code Replacement Library

After you register a library that includes your code replacement tables, generate code and verify that the code generator replaces code as you expect. The following example illustrates two complementary approaches:

- Check the Code Replacement Report section of the code generation report for expected replacements
- Trace code replacements

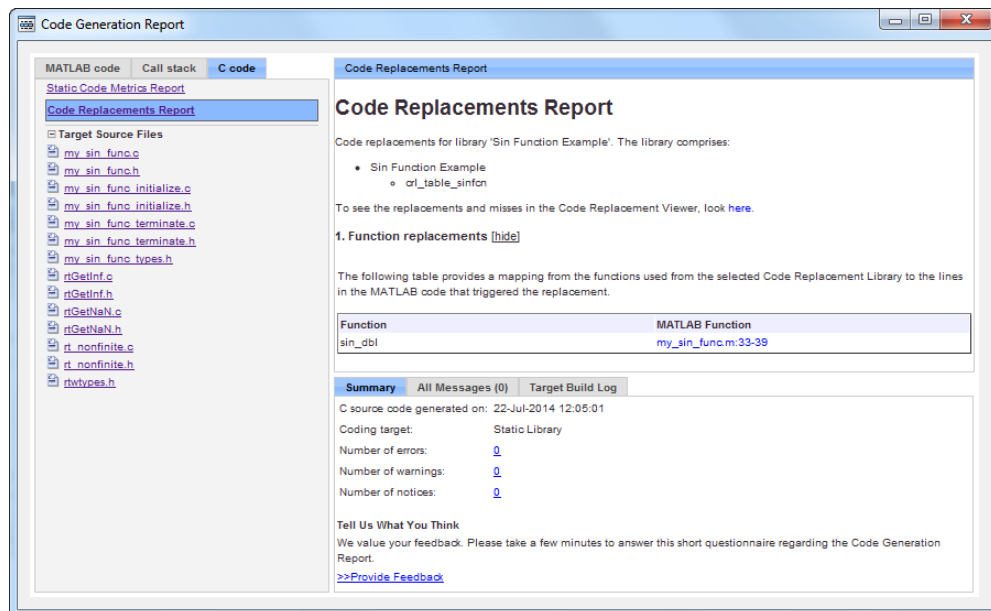
The Code Replacement Report details the code replacement library functions that the code generator uses for code replacements during code generation. It provides a mapping between each replacement instance and the line of MATLAB code that triggered the replacement. The Code Replacements report is not available for generated MEX functions.

- 1 Identify an existing or create a new MATLAB function for which you want the code generator to replace code. For example:

```
function y = my_sin_fnc(x)
    y = sin(x);
end
```

- 2 On the project **Build** tab, set the **Output type** to generate a library or executable.
- 3 Click **More Settings** to open the Project Settings dialog block.
- 4 Configure the code generator to use your code replacement library. On the **Hardware** tab, set the **Code replacement library** parameter to the name of your library. For example, **Sin Function Example**.

- 5 Configure the code generation report. On the **Debugging** tab, set the **Always create a code generation report**, **Code replacements**, and **Automatically launch a report if one is generated** parameters.
- 6 Configure the code generator to generate code only. On the **Build** tab, set the **Generate code only** parameter. You want to review your code replacements in the generated code before building an executable.
- 7 Click **Build** to generate replacement code and a report.
- 8 Open the **Code Replacements Report** section of the code generation report. That report lists the replacement functions that the code generator used, and provides a mapping between each replacement instance and the MATLAB code that triggered the replacement.



Review the report.

- Check whether expected function and operator code replacements occurred.
- In the replacements sections, trace each replacement instance back to the code that triggered the replacement.

- Click the link that opens the Code Replacement Viewer. In the viewer, review code replacement hits and misses.

Determine Why Code Replacement Functions Are Not Used

- “Code Replacement Hits and Misses” on page 22-68
- “View Cache Hits and Misses Using Viewer Trace Information” on page 22-68
- “View Cache Hits and Misses Using Command-Line” on page 22-69

Code Replacement Hits and Misses

Code replacement can behave differently than you expect. To verify code replacements, first inspect the generated code, as described in “Review and Trace Code Replacements Generated Using a Code Replacement Library” on page 22-66.

To analyze replacement behavior, in addition to examining the Code Replacements Report and examining your code replacement tables in the Code Replacement Viewer, you can view the library cache hits and misses logged the last time code was generated. This approach provides information on how to configure your table entry to get the desired replacement. The following techniques can help you determine why the code generator does not use code replacement functions:

- “View Cache Hits and Misses Using Viewer Trace Information” on page 22-68
- “View Cache Hits and Misses Using Command-Line” on page 22-69

For models that consist of model hierarchies, you must generate code for and review the trace information of each referenced model separately (stand alone). Logged cache hit and miss information captured in the Code Replacement Viewer is valid for the last model for which code was generated only. The code generator overwrites logged information as you generate code for each model in the hierarchy.

View Cache Hits and Misses Using Viewer Trace Information

When debugging a code replacement table entry, you can use hit and miss information in the Code Replacement Viewer to help determine why the code generator did not use a replacement function in the generated code.

- 1 On the project **Build** tab, set the **Output type** to generate a library or executable.
- 2 Configure the code generator to use your code replacement library. In a project, on the **Hardware** tab, set the **Code Replacement Library** parameter to the name of your library.

- 3 Configure the code generation report. In a project, on the **Build > More Settings > Debugging** tab, set the **Always create a code generation report**, **Code replacements**, and **Automatically launch a report if one is generated** parameters.
- 4 Generate replacement code and a report. A complete rebuild of code is required to display cache hits and misses.
- 5 Open the **Code Replacements Report** section of the code generation report. Click the link for opening the Code Replacement Viewer.
- 6 In the left pane of the viewer, select the code replacement table you want to examine. In the middle pane, the viewer lists information for entries in the table, including a usage count, which is the number of times the table entry was matched and its replacement function used during code generation.
- 7 In the middle pane, select the table entry to examine. In the right pane, the viewer displays **General Information** and **Trace Information** about the table entry.
- 8 In the right pane, select the **Trace Information** tab. The **Trace Information** tab lists **Hit Source Locations** and **Miss Source Locations**. The display provides links to each source location (the source for which code replacement was considered) and, for misses, lists a **Miss Reason**. If an expected replacement does not occur, you can use the hit and miss information to modify the match criteria in the code replacement table entry.

View Cache Hits and Misses Using Command-Line

To display the code replacement cache hits and misses logged during the most recent code generation session in the command window, use the following command:

```
>> cr1=get_param('model', 'TargetFcnLibHandle')
```

The resulting display includes the following fields:

Field	Description
HitCache	Table containing code replacement entries that were matched during a code generation session. These entries represent function implementations that should appear in the generated code.
MissCache	Table containing code replacement entries that failed to match during a code generation session. These entries are created by the code generation process for the purpose of querying the library to locate a registered implementation.

In the following example, the most recent code generation session logged one cache hit and zero cache misses. You can examine the logged `HitCache` entry using its table index.

```
a=get_param('my_sin_func','TargetFcnLibHandle')

a =

RTW.Tf1Control
    Version: '1.0'
    HitCache: [2x1 RTW.Tf1CFunctionEntry]
    MissCache: []
    TLCCallList: [0x1 handle]
    Tf1Tables: [10x1 RTW.Tf1Table]
    TargetHWWordLengths: '8,16,32,32'
    SupportNonFinite: 1
    TLCSupported: 1
    Recording: 0
    LoadedLibrary: 'Sin Function Example'
    TargetCharacteristics: [1x1 RTW.TargetCharacteristics]
    CurrentHWDevice: 'Specified'
```

a.HitCache(1)

```
ans =

Tf1CFunctionEntry with properties:

    Implementation: [1x1 RTW.CImplementation]
    SlopesMustBeTheSame: 0
    BiasMustBeTheSame: 0
    AlgorithmParams: []
        ImplType: 'FCN_IMPL_MACRO'
    AdditionalHeaderFiles: {0x1 cell}
    AdditionalSourceFiles: {0x1 cell}
        OtherFiles: {0x1 cell}
    AdditionalIncludePaths: {0x1 cell}
    AdditionalSourcePaths: {0x1 cell}
        AdditionalLinkObjs: {0x1 cell}
    AdditionalLinkObjsPaths: {0x1 cell}
        AdditionalLinkFlags: {0x1 cell}
    AdditionalCompileFlags: {0x1 cell}
        SearchPaths: {0x1 cell}
        Key: 'NULL'
        Priority: 100
    ConceptualArgs: [1x1 RTW.Tf1ArgNumeric]
```

```
EntryInfo: []
GenCallback: ''
GenFileName: ''
SaturationMode: 'RTW_SATURATE_UNSPECIFIED'
RoundingModes: {'RTW_ROUND_UNSPECIFIED'}
TypeConversionMode: 'RTW_EXPLICIT_CONVERSION'
AcceptExprInput: 0
SideEffects: 0
UsageCount: 3
RecordedUsageCount: 0
Description: ''
StoreFcnReturnInLocalVar: 0
TraceManager: [1x1 RTW.Tf1TraceManager]
```

Related Examples

- “Choose a Code Replacement Library”
- “Replace Code Generated from MATLAB Code”
- “Code Generation Reports”
- “Tracing Between Generated C Code and MATLAB Code”
- “Code Verification Through Software-in-the-Loop and Processor-in-the-Loop Execution”
- “Code Generation Workflow”
- “Program Builds”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “Code Replacement Libraries”
- “Code Replacement Terminology”
- “Code Replacement Customization Limitations”

External Web Sites

- “Replacing Math Functions and Operators”

Deploy Code Replacement Library

When you are ready to package and deploy a custom code replacement library for others to use,

- 1 Move your code replacement table files to an area that is on the MATLAB search path and that is accessible to and shared by other users.
- 2 Move the `rtwTargetInfo.m` registration file, to an area that is on the MATLAB search path and that is accessible to and shared by other users. If you are deploying a library to a folder in a development environment that already contains a `rtwTargetInfo.m` file, copy the registration code from your code replacement library version of `rtwTargetInfo.m` and paste it into the shared version of that file.
- 3 Register the library customizations or restart MATLAB.
- 4 Verify that the libraries are available for configuring the code generator and that code replacements occur as expected.
- 5 Inform users that the libraries are available and provide direction on when and how to apply them.

Related Examples

- “Review and Test Code Replacements”
- “Develop a Code Replacement Library”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code Replacement Libraries”
- “Code Replacement Terminology”

Math Function Code Replacement

This example shows how to define a code replacement mapping for a math function. The example defines a mapping for the `sin` function programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

- 1 Define a function for the definition.

```
function hTable = crl_table_sinfcn2()
%CRL_TABLE_SINFCN2 - Define function entry for code replacement table.
```

- 2 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 3 Create an entry for the function mapping with a call to the `RTW.Tf1CFunctionEntry` function.

```
% Create entry for sin function replacement
fcn_entry = RTW.Tf1CFunctionEntry;
```

- 4 Set function entry parameters with a call to the `setTf1CFunctionEntryParameters` function.

```
setTf1CFunctionEntryParameters(fcn_entry, ...
                               'Key',          'sin', ...
                               'Priority',      30, ...
                               'ImplementationName', 'mySin', ...
                               'ImplementationHeaderFile', 'basicMath.h',...
                               'ImplementationSourceFile', 'basicMath.c');
```

- 5 Create conceptual arguments `y1` and `u1`. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call.

```
createAndAddConceptualArg(fcn_entry, 'RTW.Tf1ArgNumeric', ...
                          'Name',    'y1', ...
                          'IOType',  'RTW_IO_OUTPUT', ...
                          'DataTypeMode', 'double');

createAndAddConceptualArg(fcn_entry, 'RTW.Tf1ArgNumeric', ...
                          'Name',    'u1', ...
                          'IOType',  'RTW_IO_INPUT', ...
                          'DataTypeMode', 'double');
```

- 6 Copy the conceptual arguments to the implementation arguments. This example uses a call to the `copyConceptualArgsToImplementation` function to create and add implementation arguments to the entry by copying matching conceptual arguments.

```
copyConceptualArgsToImplementation(fcn_entry);
```

- 7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, fcn_entry);
```

Related Examples

- “Define Code Replacement Mappings”
- “Specify In-Place Code Replacement”
- “Reserved Identifiers and Code Replacement”
- “Customize Matching and Replacement Process for Functions”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “Code You Can Replace from MATLAB Code”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Memory Function Code Replacement

This example shows how to define a code replacement mapping for a memory function. The example defines a mapping for the `memcpy` function programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

- 1 Define a function for the definition.

```
function hTable = crl_table_memcpy()
```

- 2 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 3 Create an entry for the function mapping with a call to the `RTW.Tf1CFunctionEntry` function.

```
% Create entry for void* memcpy(void*, void*, size_t)
fcn_entry = RTW.Tf1CFunctionEntry;
```

- 4 Set function entry parameters with a call to the `setTf1CFunctionEntryParameters` function.

```
% Set SideEffects to 'true' for function returning void to prevent it from
% being optimized away.
setTf1CFunctionEntryParameters(fcn_entry, ...
    'Key', 'memcpy', ...
    'Priority', 90, ...
    'ImplementationName', 'memcpy_int', ...
    'ImplementationHeaderFile', 'memcpy_int.h', ...
    'SideEffects', true);
```

- 5 Create conceptual arguments `y1`, `u1`, `u2`, and `u3`. This example uses calls to the `getTf1ArgFromString` and `addConceptualArg` functions to create and add the arguments.

```
arg = getTf1ArgFromString(hTable, 'y1', 'void*');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(fcn_entry, arg);

arg = getTf1ArgFromString(hTable, 'u1', 'void*');
addConceptualArg(fcn_entry, arg);

arg = getTf1ArgFromString(hTable, 'u2', 'void*');
addConceptualArg(fcn_entry, arg);

arg = getTf1ArgFromString(hTable, 'u3', 'size_t');
addConceptualArg(fcn_entry, arg);
```

- 6 Copy the conceptual arguments to the implementation arguments. This example uses a call to the `copyConceptualArgsToImplementation` function to create

and add implementation arguments to the entry by copying matching conceptual arguments.

```
copyConceptualArgsToImplementation(fcn_entry);
```

- 7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, fcn_entry);
```

Related Examples

- “Define Code Replacement Mappings”
- “Specify In-Place Code Replacement”
- “Reserved Identifiers and Code Replacement”
- “Customize Matching and Replacement Process for Functions”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “Code You Can Replace from MATLAB Code”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Specify In-Place Code Replacement

In this section...

“In-Place Code Replacement” on page 22-77

“Argument Specification Requirements” on page 22-77

“Interactive Argument Replacement Specification with Code Replacement Tool” on page 22-77

“Programmatic Argument Replacement Specification” on page 22-81

In-Place Code Replacement

In-place code replacement is an optimization technique that uses a single buffer, that is, the same memory, to store function input and output data, as in $x = f_{OO}(x)$.

When you generate C or C++ code from MATLAB code, the code generator supports in-place function argument code replacement. When you interactively create a code replacement table entry with the Code Replacement Tool, you can specify in-place function argument replacement. You can also specify in-place function argument replacement programmatically with the Code Replacement Library API.

Argument Specification Requirements

- The argument must be a pointer.
- An argument can be in-place with only one other argument.
- Specify an input argument as in-place (shares memory) with an output argument or an output argument as in-place with an input argument.

Interactive Argument Replacement Specification with Code Replacement Tool

This example shows how to specify in-place function argument replacement when replacing code for a MATLAB function with the Code Replacement Tool. The tool enforces in-place argument specification requirements as you add arguments and modify argument properties.

- 1 Create the following MATLAB function, `customFunction.m`.

```
function x = customFunction(x)
% Function that updates the input and returns it as an output

coder.replace('-errorifnoreplacement');
x = sin(x);
```

- 2 In the Code Replacement Tool, add a new table, select that table, and add a new function entry. For more information, see “Define Code Replacement Mappings”.
- 3 On the **Mapping Information** tab, select **Custom** for the **Function** parameter.
- 4 In the **function-name** text box, name the custom function. For this example, type the name `customFunction`.
- 5 Under the **Conceptual arguments** list box, click **+** to add two arguments. By default, the tool creates an output argument `y1` and an input argument `u1`, both of type `double`.
- 6 In the **Replacement function > Function prototype** section, type the name `custom_function_inplace_impl` in the **Name** text box.
- 7 Under the **Function arguments** list box, click **+** to add two function implementation arguments. By default, the tool creates an output argument `y1` and an input argument `u1`, both of type `double`.
- 8 For each input argument that you want to specify as in-place with a corresponding output argument, in the **Argument properties** box, select the **Pointer** check box. The **Argument properties** section of the dialog box expands to include an **In-place argument** drop-down list. For this example, in the **Function arguments** list, select input argument `u1`, and then select the **Pointer** check box.

Replacement function

Function prototype

Name: C++ namespace:

Function returns void

Function arguments

y1(return arg)
u1

Argument properties

Data type: I/O type:

Const Pointer Complex

Alignment value:

In-place argument mapping

In-place argument

Make constant value

Function signature preview

```
double custom_function_inplace_impl( double u1);
```

- 9 From the **In-place argument** list, select *y1*, the output argument for the code replacement mapping. The **Function arguments** list box is updated to show possible in-place argument mappings.

Replacement function

Function prototype

Name: C++ namespace:

Function returns void

Function arguments

```
void(return arg)
u1 <-> y1
y1 <-> u1
```

Argument properties

Data type: I/O type:

Const Pointer Complex

Alignment value:

In-place argument mapping

In-place argument

Make constant value

Function signature preview

```
void custom_function_inplace_impl( double* u1, double* y1);
```

- 10 Select and delete one of the two possible argument mappings. For this example, delete the mapping `y1<->u1`.
- 11 In the **Function signature preview** box, if the function signature appears as expected, click **Apply**. Otherwise, make adjustments, and then click **Apply**. The function signature for this example, appears as


```
void custom_function_inplace_impl(double* u1);
```
- 12 Click **Validate entry**.
- 13 Save the code replacement table in the same folder as `customFunction.m`. Name the file `htfl_inplace_table.m`.

To test the example:

- 1 Register the table that contains the entry in a code replacement library.
- 2 Configure the code generator to use a code replacement library and to include the Code Replacements Report in the code generation report.
- 3 Generate the replacement code and a code generation report.
- 4 Review the code replacements.

Programmatic Argument Replacement Specification

This example shows how to specify in-place function argument replacement when replacing code for a MATLAB function programmatically. For the input implementation argument that shares the memory buffer, the example:

- Sets the name of the implementation argument to the same name as the corresponding conceptual argument.
- Associates the corresponding implementation argument with the argument property `ArgumentForInPlaceUse`.

- 1 Create the following MATLAB function, `customFunction.m`.

```
function y = customFunction(x)
% Function that updates the input and returns it as an output

coder.replace('-errorifnoreplacement');
x = sin(x);
```

- 2 Define a function for the definition.

```
function hTable = crl_table_inplace()
```

- 3 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 4 Create an entry for the function mapping with a call to the `RTW.Tf1CFunctionEntry` function.

```
hEnt = RTW.Tf1CFunctionEntry;
```

- 5 Set function entry parameters with a call to the `setTf1CFunctionEntryParameters` function.

```
setTf1CFunctionEntryParameters(hEnt, ...
    'Key', 'customFunction', ...
    'Priority', 100, ...
    'ImplementationName', 'custom_function_inplace_impl', ...
    'SideEffects', true);
```

- 6 Create conceptual arguments `y1` and `u1`. This example uses calls to the `getTf1ArgFromString` and `addConceptualArg` functions to create and add the arguments.

```
arg = getTf1ArgFromString(hEnt, 'y1', 'double');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEnt, arg);
```

```
arg = getTf1ArgFromString(hEnt, 'u1', 'double');  
addConceptualArg(hEnt, arg);
```

- 7 Create the implementation arguments and add them to the entry. This example uses calls to the `getTf1ArgFromString` function to create implementation arguments that map to arguments in the replacement function prototype: output argument `y1` and input argument `u1`. For each argument, the example uses the convenience method `setReturn` or `addArgument` to specify whether an argument is a return value or argument. For each argument, this example adds the argument to the entry array of implementation arguments.

```
arg = getTf1ArgFromString(hEnt, 'y2', 'void');  
arg.IOType = 'RTW_IO_OUTPUT';  
hEnt.Implementation.setReturn(arg);
```

```
arg = getTf1ArgFromString(hEnt, 'u1', 'double*');  
arg.ArgumentForInPlaceUse = 'y1';  
hEnt.Implementation.addArgument(arg);
```

- 8 Add the entry to a code replacement table with a call to the `addEntry` function.

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

To test the example:

- 1 Register the table that contains the entry in a code replacement library.
- 2 Configure the code generator to use a code replacement library and to include the Code Replacements Report in the code generation report.
- 3 Generate the replacement code and a code generation report.
- 4 Review the code replacements.

Related Examples

- “Define Code Replacement Mappings”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “Code You Can Replace from MATLAB Code”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Replace MATLAB Functions with Custom Code Using `coder.replace`

The `coder.replace` function provides the ability to replace a specified MATLAB function with a code replacement function in generated code. Use `coder.replace` in MATLAB code from which you want to generate C code using:

- MATLAB Coder
- MATLAB code in a Simulink MATLAB Function block

You can replace MATLAB functions that have:

- Single or multiple inputs
- Single or multiple outputs
- Scalar and matrix inputs and outputs

Supported types include:

- `single`, `double` (complex and noncomplex)
- `int8`, `uint8` (complex and noncomplex)
- `int16`, `uint16` (complex and noncomplex)
- `int32`, `uint32` (complex and noncomplex)
- Fixed-point integers
- Mixed types (different type on each input)

Related Examples

- “Define Code Replacement Mappings”
- “Register Code Replacement Mappings”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “Code You Can Replace from MATLAB Code”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Reserved Identifiers and Code Replacement

The code generator and C programming language use, internally, reserved keywords for code generation. Do not use reserved keywords as identifiers or function names. Reserved keywords for code generation include many code replacement library identifiers, the majority of which are function names, such as `acos`.

To view a list of reserved identifiers for the code replacement library that you use to generate code, specify the name of the library in a call to the function `RTW.TargetRegistry.getInstance.getTf1ReservedIdentifiers`. For example:

```
cr1_ids = RTW.TargetRegistry.getInstance.getTf1ReservedIdentifiers('GNU99 (GNU)')
```

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.

To register additional code replacement reserved identifiers, use the `setReservedIdentifiers` function. This function registers specified reserved identifiers to be associated with a code replacement table.

You can register up to four reserved identifier structures in a code replacement table. You can associate one set of reserved identifiers with a code replacement library, while the other three (if present) must be associated with ANSI C. The following example shows a reserved identifier structure that specifies two identifiers and the associated header file.

```
d{1}.LibraryName = 'ANSI_C';  
d{1}.HeaderInfos{1}.HeaderName = 'math.h';  
d{1}.HeaderInfos{1}.ReservedIds = {'y0', 'y1'};
```

The code generator adds the identifiers to the list of reserved identifiers and honors them during the build procedure.

Related Examples

- “Define Code Replacement Mappings”
- “Register Code Replacement Mappings”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “What Is Code Replacement Customization?”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Customize Matching and Replacement Process for Functions

During the build process, the code generator uses:

- Preset match criteria to identify functions and operators for which application-specific implementations should replace default implementations
- Preset replacement function signatures

However, preset match criteria and preset replacement function signatures might not completely meet your function and operator replacement needs. For example,

- You might want to replace an operator with a particular fixed-point implementation function only when fraction lengths are within a particular range.
- When a match is made, you might want to modify your replacement function signature based on compile-time information, such as passing fraction-length values into the function.

When you need to add extra logic into the code replacement matching and replacement process, you can create custom code replacement table entries. Custom entries allow you to specify additional match criteria and/or modify the replacement function signature to meet application needs.

To create a custom code replacement entry:

- 1** Create a custom code replacement entry class, derived from `RTW.Tf1CFunctionEntryML` (for function replacement) or `RTW.Tf1COperationEntryML` (for operator replacement).
- 2** In your derived class, implement a `do_match` method with a fixed preset signature as a MATLAB function. In your `do_match` method, provide either or both of the following customizations for use by code replacement entries that instantiate the class:
 - a** Add additional match criteria not provided by the base class. The base class provides a match based on argument number, argument name, signedness, word size, slope (if not specified with wildcards), bias (if not specified with wildcards), math modes such as saturation and rounding, and operator or function key. For example, you can accept a match only when additional size or range conditions are met.
 - b** Modify the implementation signature by adding additional arguments or setting constant input argument values. For example, you can inject a constant value,

such as an input's scaling value, as an additional argument to the replacement function.

- 3 Create code replacement entries that instantiate your custom entry class.
- 4 Register a library containing the code replacement table that includes your entries.

During code generation, the code replacement matching process first tries to match function or operator call sites with the base class of your derived entry class. If a match is found, the software calls your `do_match` method to execute your additional match logic (if any) and your replacement function customizations (if any).

Related Examples

- [“Define Code Replacement Mappings”](#)
- [“Register Code Replacement Mappings”](#)
- [“Develop a Code Replacement Library”](#)
- [“Quick Start Library Development”](#)

More About

- [“What Is Code Replacement?”](#)
- [“What Is Code Replacement Customization?”](#)
- [“Code Replacement Match and Replacement Process”](#)
- [“Code Replacement Terminology”](#)

External Web Sites

- [“Replacing Math Functions and Operators”](#)

Scalar Operator Code Replacement

This example shows you how to define a code replacement mapping for a scalar operator. The example defines a mapping for the + (addition) operator programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

- 1 Define a function for the definition.

```
function hTable = crl_table_add_uint8
```

- 2 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 3 Create an entry for the operator mapping with a call to the `RTW.Tf1COperationEntry` function.

```
% Create operation entry
op_entry = RTW.Tf1COperationEntry;
```

- 4 Set function entry parameters with a call to the `setTf1COperationEntryParameters` function.

```
% Define addition operation of built-in uint8 data type
% Saturation on, Rounding unspecified
setTf1COperationEntryParameters(op_entry, ...
    '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');
```

- 5 Create conceptual arguments `y1`, `u1`, and `u2`. This example uses calls to the `getTf1ArgFromString` and `addConceptualArg` functions to create and add the arguments.

```
arg = getTf1ArgFromString(hTable, 'y1', 'uint8');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(op_entry, arg);
```

```
arg = getTf1ArgFromString(hTable, 'u1', 'uint8');
addConceptualArg(op_entry, arg);
```

```
arg = getTf1ArgFromString(hTable, 'u2', 'uint8');
addConceptualArg(op_entry, arg);
```

- 6 Copy the conceptual arguments to the implementation arguments. This example uses a call to the `copyConceptualArgsToImplementation` function to create

and add implementation arguments to the entry by copying matching conceptual arguments.

```
copyConceptualArgsToImplementation(op_entry);
```

- 7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

Related Examples

- “Define Code Replacement Mappings”
- “Addition and Subtraction Operator Code Replacement”
- “Remap Operator Output to Function Input”
- “Customize Matching and Replacement Process for Operators”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code You Can Replace from MATLAB Code”
- “Code Replacement Match and Replacement Process”
- “Code Replacement Libraries”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Addition and Subtraction Operator Code Replacement

In this section...

“Algorithm Options” on page 22-92

“Interactive Specification with Code Replacement Tool” on page 22-93

“Programmatic Specification” on page 22-93

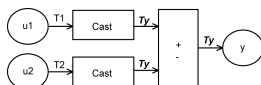
“Algorithm Classification” on page 22-93

“Limitations” on page 22-95

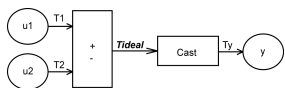
Algorithm Options

When creating a code replacement table entry for an addition or subtraction operator, first determine the type of algorithm that your library function implements.

- **Cast-before-operation (CBO), default** — Prior to performing the addition or subtraction operation, the algorithm type casts input values to the output type. If the output data 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.



- **Cast-after-operation (CAO)** — The algorithm computes the ideal result of the addition or subtraction operation of the two inputs. The algorithm then type casts 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 operand 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.



Interactive Specification with Code Replacement Tool

When you use the Code Replacement Tool to create a code replacement table entry for an addition or subtraction operation, the tool displays an **Algorithm** menu. Use that menu to specify the `Cast before operation` or `Cast after operation` algorithm for that entry.

Programmatic Specification

Create a code replacement table file, as a MATLAB function, that describes the addition or subtraction code replacement table entry. In the call to `setTf1COperationEntryParameters`, set at least these parameters:

- `Key` to `RTW_OP_ADD` or `RTW_OP_MINUS`
- `ImplementationName` to the name of your replacement function
- `EntryInfoAlgorithm` to `RTW_CAST_BFORE_OP` (cast-before-operation) or `RTW_CAST_AFTER_OP` (cast-after-operation)

This example sets parameters for a code replacement operator entry for a cast-after-operation implementation of a `uint8` addition.

```
op_entry = RTW.Tf1COperationEntry;
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_ADD', ...
    'EntryInfoAlgorithm', 'RTW_CAST_AFTER_OP', ...
    'ImplementationName', 'u8_add_u8_u8');
```

For more information, see `setTf1COperationEntryParameters`.

Algorithm Classification

During code generation, the code generator examines addition and subtraction operations, including adjacent type cast operations, to determine the type of algorithm to compute the expression result. Based on the data types in the expression and the type of the accumulator (type used to hold the result of the addition or subtraction operation), the code generator uses these rules.

- Floating-point types only

Input 1 Data Type	Input 2 Data Type	Accumulator Data Type	Output Data Type	Classification
double	double	double	double	CBO, CAO

Input 1 Data Type	Input 2 Data Type	Accumulator Data Type	Output Data Type	Classification
double	double	double	single	—
double	double	single	double	—
double	double	single	single	CBO
double	single	double	double	CBO, CAO
double	single	double	single	—
double	single	single	double	—
double	single	single	single	CBO
single	single	single	single	CBO, CAO
single	single	single	double	—
single	single	double	single	—
single	single	double	double	CBO, CAO

- Floating-point and fixed-point types on the immediate addition or subtraction operation

Algorithm	Conditions
CBO	One of the following is true: <ul style="list-style-type: none"> • Operation type is double. • Operation type is single and input types are single or fixed-point.
CAO	Operation type is a superset of input types—that is, output type can represent values of input types without loss of data.

- Fixed-point types only

Algorithm	Conditions
CBO	At least one of the following is true: <ul style="list-style-type: none"> • Accumlator type equals output type ($T_{acc} == T_{out}$). • Output type is a superset of input types ($T_{acc} \geq \{T_{in1}, T_{in2}\}$) and accumulator type is a superset of output type ($T_{acc} \geq T_{out}$). • Operation does not incur range or precision loss.

Algorithm	Conditions
CAO	Net bias is zero and the data types in the expression have equal slope adjustment factors. For more information on net bias, see “Addition” or “Subtraction” in “Fixed-Point Operator Code Replacement” (for MATLAB code) or “Fixed-Point Operator Code Replacement” (for Simulink models).

In many cases, the numerical result of a CBO operation is equal to that of a CAO operation. For example, if the input and output types are such that the operation produces the ideal result, as in the case of `int8 + int8 -> int16`. To maximize the probability of code replacement occurring in such cases, set the algorithm to cast-after-operation.

Limitations

- The code generator does not replace operations with nonzero net bias.
- When classifying an operation as a CAO operation, the code generator includes the adjacent casts in the expression when the expression involves only fixed-point types. Otherwise, the code generator classifies and replaces only the immediate addition or subtraction operation. Casts that the code generator excludes from the classification appear in the generated code.
- To enable the code generator to include multiple cast operations, which follow an addition or subtraction of fixed-point data, in the classification of an expression, the rounding mode must be `simplest` or `floor`. Consider the expression `y=(cast A) (cast B) (u1+u2)`. If the rounding mode of `(cast A)`, `(cast B)`, and the addition operator (+) are set to `simplest` or `floor`, the code generator takes into account `(cast A)` and `(cast B)` when classifying the expression and performing the replacement only.

Related Examples

- “Define Code Replacement Mappings”
- “Remap Operator Output to Function Input”
- “Customize Matching and Replacement Process for Operators”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code You Can Replace from MATLAB Code”
- “Code Replacement Match and Replacement Process”
- “Code Replacement Libraries”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”
- `rtwdemo_crl_cbo_cao`

Small Matrix Operation to Processor Code Replacement

This example shows how to define code replacement mappings that replace nonscalar small matrix operations with processor-specific intrinsic functions. The example defines a table containing two matrix operator replacement entries for the + (addition) operator and the `double` data type. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

- 1 Define a function for the definition.

```
function hTable = crl_table_matrix_add_double
```

- 2 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 3 Create the entry for the first operator mapping with a call to the `RTW.Tf1COperationEntry` function.

```
% Create table entry for matrix_sum_2x2_double
op_entry = RTW.Tf1COperationEntry;
```

- 4 Set operator entry parameters with a call to the `setTf1COperationEntryParameters` function. The code generator ignores saturation and rounding modes for floating-point nonscalar addition and subtraction. For code replacement entries for nonscalar addition and subtraction operations that do not involve fixed-point data, in the call to `setTf1COperationEntryParameters`, specify `'RTW_SATURATE_UNSPECIFIED'` for the `SaturationMode` property and `{'RTW_ROUND_UNSPECIFIED'}` for `RoundingModes`.

```
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_ADD', ...
    'Priority', 30, ...
    'SaturationMode', 'RTW_WRAP_ON_OVERFLOW', ...
    'ImplementationName', 'matrix_sum_2x2_double', ...
    'ImplementationHeaderFile', 'MatrixMath.h', ...
    'ImplementationSourceFile', 'MatrixMath.c', ...
    'ImplementationHeaderPath', LibPath, ...
    'ImplementationSourcePath', LibPath, ...
    'AdditionalIncludePaths', {LibPath}, ...
    'GenCallback', 'RTW.copyFileToBuildDir', ...
    'SideEffects', true);
```

- 5 Create conceptual arguments `y1`, `u1`, and `u2`. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. To specify a matrix argument in the function call, use the argument class `RTW.Tf1ArgMatrix`. Specify the base type and the dimensions for which the

argument is valid. The first table entry specifies [2 2] and the second table entry specifies [3 3].

```
% Specify operands and result
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name', 'y1', ...
    'IOType', 'RTW_IO_OUTPUT', ...
    'BaseType', 'double', ...
    'DimRange', [2 2]);
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix',...
    'Name', 'u1', ...
    'BaseType', 'double', ...
    'DimRange', [2 2]);
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix',...
    'Name', 'u2', ...
    'BaseType', 'double', ...
    'DimRange', [2 2]);
```

- 6 Create the implementation arguments. This example uses calls to the `getTf1ArgFromString` to create the arguments. The convenience methods `setReturn` and `addArgument` specify whether an argument is a return value or argument and adds the argument to the entry's array of implementation arguments.

```
arg = getTf1ArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.setReturn(arg);

arg = getTf1ArgFromString(hTable, 'u1', ['double' '*']);
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'u2', ['double' '*']);
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'y1', ['double' '*']);
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.addArgument(arg);
```

- 7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

- 8 Create the entry for the second operator mapping.

```
% Create table entry for matrix_sum_3x3_double
op_entry = RTW.Tf1COperationEntry;
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_ADD', ...
    'Priority', 30, ...
    'SaturationMode', 'RTW_WRAP_ON_OVERFLOW', ...
    'ImplementationName', 'matrix_sum_3x3_double', ...
    'ImplementationHeaderFile', 'MatrixMath.h', ...
    'ImplementationSourceFile', 'MatrixMath.c', ...
```

```

    'ImplementationHeaderPath', LibPath, ...
    'ImplementationSourcePath', LibPath, ...
    'AdditionalIncludePaths', {LibPath}, ...
    'GenCallback', 'RTW.copyFileToBuildDir', ...
    'SideEffects', true);

% Specify operands and result
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name', 'y1', ...
    'IOType', 'RTW_IO_OUTPUT', ...
    'BaseType', 'double', ...
    'DimRange', [3 3]);
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name', 'u1', ...
    'BaseType', 'double', ...
    'DimRange', [3 3]);
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name', 'u2', ...
    'BaseType', 'double', ...
    'DimRange', [3 3]);

% Specify replacement function signature
arg = getTf1ArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.setReturn(arg);
arg = getTf1ArgFromString(hTable, 'u1', ['double' '*']);
op_entry.Implementation.addArgument(arg);
arg = getTf1ArgFromString(hTable, 'u2', ['double' '*']);
op_entry.Implementation.addArgument(arg);
arg = getTf1ArgFromString(hTable, 'y1', ['double' '*']);
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.addArgument(arg);

addEntry(hTable, op_entry);

```

Related Examples

- “Define Code Replacement Mappings”
- “Matrix Multiplication Operation to MathWorks BLAS Code Replacement”
- “Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement”
- “Remap Operator Output to Function Input”
- “Customize Matching and Replacement Process for Operators”

- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code You Can Replace from MATLAB Code”
- “Code Replacement Match and Replacement Process”
- “Code Replacement Libraries”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Matrix Multiplication Operation to MathWorks BLAS Code Replacement

This example shows how to define code replacement mappings that replace nonscalar multiplication operations with Basic Linear Algebra Subroutine (BLAS) multiplication functions `xgemm` and `xgemv`. The example defines code replacement entries that map floating-point matrix/matrix and matrix/vector multiplication operations to MathWorks BLAS library multiplication functions `dgemm` and `dgemv`. The example defines the function mappings programmatically. Alternatively, you can use the Code Replacement Tool to define the same mappings.

BLAS libraries support matrix/matrix multiplication in the form of

$C = a(\text{op}(A) * \text{op}(B)) + bC$. `op(X)` means X, transposition of X, or Hermitian transposition of X. However, code replacement libraries support only the limited case of $C = \text{op}(A) * \text{op}(B)$ ($a = 1.0$, $b = 0.0$). Correspondingly, although BLAS libraries support matrix/vector multiplication in the form of $y = a(\text{op}(A) * x) + by$, code replacement libraries support only the limited case of $y = \text{op}(A) * x$ ($a = 1.0$, $b = 0.0$).

- 1 Define a function for the definition.

```
function hTable = crl_table_tmwblas_mmult_double
```

- 2 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 3 Define the path for the BLAS function library. If your replacement functions are on the MATLAB search path or are in your working folder, you can skip this step.

```
% Define library path for Windows or UNIX
arch = computer('arch');
if ~ispc
    LibPath = fullfile('$MATLAB_ROOT', 'bin', arch);
else
    % Use Stateflow to get the compiler info
    compilerInfo = sf('Private','compilerman','get_compiler_info');
    compilerName = compilerInfo.compilerName;
    if strcmp(compilerName, 'msvc90') || ...
        strcmp(compilerName, 'msvc80') || ...
        strcmp(compilerName, 'msvc71') || ...
        strcmp(compilerName, 'msvc60'), ...
        compilerName = 'microsoft';
end
LibPath = fullfile('$MATLAB_ROOT', 'extern', 'lib', arch, compilerName);
end
```

- 4 Create an entry for the first mapping with a call to the `RTW.Tf1BlasEntryGenerator` function.

```
% Create table entry for dgemm32
op_entry = RTW.Tf1BlasEntryGenerator;
```

- 5 Set operator entry parameters with a call to the `setTf1CFunctionEntryParameters` function. The function call sets matrix multiplication operator entry properties. The code generator ignores saturation and rounding modes for floating-point nonscalar addition and subtraction. For code replacement entries for nonscalar addition and subtraction operations that do not involve fixed-point data, in the call to `setTf1CFunctionEntryParameters`, specify `'RTW_SATURATE_UNSPECIFIED'` for the `SaturationMode` property and `{'RTW_ROUND_UNSPECIFIED'}` for `RoundingModes`.

```
if ispc
    libExt = 'lib';
elseif ismac
    libExt = 'dylib';
else
    libExt = 'so';
end
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_MUL', ...
    'Priority', 100, ...
    'ImplementationName', 'dgemm32', ...
    'ImplementationHeaderFile', 'blascompat32.h', ...
    'ImplementationHeaderPath', fullfile('${MATLAB_ROOT}','extern','include'), ...
    'AdditionalLinkObjs', {'libmwbblascompat32.' libExt}, ...
    'AdditionalLinkObjsPaths', {LibPath}, ...
    'SideEffects', true);
```

- 6 Create conceptual arguments `y1`, `u1`, and `u2`. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. To specify a matrix argument in the function call, use the argument class `RTW.Tf1ArgMatrix` and specify the base type and the dimensions for which the argument is valid. This type of table entry supports 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. The conceptual output argument for the `dgemm32` entry for matrix/matrix multiplication replacement specifies dimensions `[2 2; inf inf]`, while the conceptual output argument for the `dgemv32` entry for matrix/vector multiplication replacement specifies dimensions `[2 1; inf 1]`.

```
% Specify operands and result
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name', 'y1', ...
    'IOType', 'RTW_IO_OUTPUT', ...
```

```

        'BaseType',    'double', ...
        'DimRange',   [2 2; inf inf]);
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
        'Name',       'u1', ...
        'BaseType',   'double', ...
        'DimRange',   [2 2; inf inf]);
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
        'Name',       'u2', ...
        'BaseType',   'double', ...
        'DimRange',   [1 1; inf inf]);

```

- 7 Create the implementation arguments. This example uses calls to the `getTf1ArgFromString` and `RTW.Tf1ArgCharConstant` functions to create the arguments. The example code configures special implementation arguments that are required for `dgemm` and `dgemv` function replacements. The convenience methods `setReturn` and `addArgument` specify whether an argument is a return value or argument and adds the argument to the entry's array of implementation arguments.

```

% Using RTW.Tf1BlasEntryGenerator for xgemv requires the following
% implementation signature:
%
% void f(char* TRANSA, char* TRANSB, int* M, int* N, int* K,
%        type* ALPHA, type* u1, int* LDA, type* u2, int* LDB,
%        type* BETA, type* y, int* LDC)
%
% When a match occurs, the code generator computes the
% values for M, N, K, LDA, LDB, and LDC and inserts them into the
% generated code. TRANSA and TRANSB are set to 'N'.

% Specify replacement function signature

arg = getTf1ArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.setReturn(arg);

arg = RTW.Tf1ArgCharConstant('TRANSA');
% Possible values for PassByType property are
% RTW_PASSBY_AUTO, RTW_PASSBY_POINTER,
% RTW_PASSBY_VOID_POINTER, RTW_PASSBY_BASE_POINTER
arg.PassByType = 'RTW_PASSBY_POINTER';
op_entry.Implementation.addArgument(arg);

arg = RTW.Tf1ArgCharConstant('TRANSB');
arg.PassByType = 'RTW_PASSBY_POINTER';
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'M', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'N', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;

```

```

op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'K', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'ALPHA', 'double', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'u1', ['double' '*']);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'LDA', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'u2', ['double' '*']);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'LDB', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'BETA', 'double', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'y1', ['double' '*']);
arg.IOType = 'RTW_IO_OUTPUT';
arg.PassByType = 'RTW_PASSBY_POINTER';
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'LDC', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

```

- 8** Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

- 9** Create the entry for the second mapping.

```

% Create table entry for dgemv32
op_entry = RTW.Tf1BlasEntryGenerator;
if ispc

```

```

    libExt = 'lib';
elseif ismac
    libExt = 'dylib';
else
    libExt = 'so';
end
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_MUL', ...
    'Priority', 100, ...
    'ImplementationName', 'dgemv32', ...
    'ImplementationHeaderFile', 'blascompat32.h', ...
    'ImplementationHeaderPath', fullfile('${MATLAB_ROOT}','extern','include'), ...
    'AdditionalLinkObjs', {'libmwblascompat32.' libExt}], ...
    'AdditionalLinkObjsPaths', {LibPath},...
    'SideEffects', true);

% Specify operands and result
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name', 'y1', ...
    'IOType', 'RTW_IO_OUTPUT', ...
    'BaseType', 'double', ...
    'DimRange', [2 1; inf 1]);
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name', 'u1', ...
    'BaseType', 'double', ...
    'DimRange', [2 2; inf inf]);
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix',...
    'Name', 'u2', ...
    'BaseType', 'double', ...
    'DimRange', [1 1; inf 1]);

% Using RTW.Tf1BlasEntryGenerator for xgemv requires the following
% implementation signature:
%
% void f(char* TRANS, int* M, int* N,
%       type* ALPHA, type* u1, int* LDA, type* u2, int* INCX,
%       type* BETA, type* y, int* INCY)
%
% Upon a match, the CRL entry will compute the
% values for M, N, LDA, INCX, and INCY, and insert them into the
% generated code. TRANS will be set to 'N'.

% Specify replacement function signature

arg = getTf1ArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.setReturn(arg);

arg = RTW.Tf1ArgCharConstant('TRANS');
arg.PassByType = 'RTW_PASSBY_POINTER';
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'M', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;

```

```
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'N', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'ALPHA', 'double', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'u1', ['double' '*']);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'LDA', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'u2', ['double' '*']);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'INCX', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'BETA', 'double', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'y1', ['double' '*']);
arg.IOType = 'RTW_IO_OUTPUT';
arg.PassByType = 'RTW_PASSBY_POINTER';
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'INCY', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

addEntry(hTable, op_entry);
```

Related Examples

- “Define Code Replacement Mappings”
- “Small Matrix Operation to Processor Code Replacement”

- “Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement”
- “Remap Operator Output to Function Input”
- “Customize Matching and Replacement Process for Operators”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code You Can Replace from MATLAB Code”
- “Code Replacement Match and Replacement Process”
- “Code Replacement Libraries”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement

This example shows how to define code replacement mappings that replace nonscalar multiplication operations with ANSI/ISO C BLAS multiplication functions `xgemm` and `xgemv`. The example defines code replacement entries that map floating-point matrix/matrix and matrix/vector multiplication operations to ANSI/ISO C BLAS library multiplication functions `dgemm` and `dgemv`. The example defines the function mappings programmatically. Alternatively, you can use the Code Replacement Tool to define the same mappings.

BLAS libraries support matrix/matrix multiplication in the form of $C = a(\text{op}(A) * \text{op}(B)) + bC$. `op(X)` means `X`, transposition of `X`, or Hermitian transposition of `X`. However, code replacement libraries support only the limited case of $C = \text{op}(A) * \text{op}(B)$ ($a = 1.0, b = 0.0$). Correspondingly, although BLAS libraries support matrix/vector multiplication in the form of $y = a(\text{op}(A) * x) + by$, code replacement libraries support only the limited case of $y = \text{op}(A) * x$ ($a = 1.0, b = 0.0$).

- 1 Define a function for the definition.

```
function hTable = crl_table_cblas_mmult_double
```

- 2 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 3 Define the path for the CBLAS function library. For example:

```
LibPath = fullfile(matlabroot, 'toolbox', 'rtw', 'rtwdemos', 'crl_demo');
```

- 4 Create an entry for the first mapping with a call to the `RTW.Tf1BlasEntryGenerator` function.

```
% Create table entry for cblas_dgemm
op_entry = RTW.Tf1CBlasEntryGenerator;
```

- 5 Set operator entry parameters with a call to the `setTf1COperationEntryParameters` function. The function call sets matrix multiplication operator entry properties. The code generator ignores saturation and rounding modes for floating-point nonscalar addition and subtraction.

```
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_MUL', ...
```



```

'Priority',          100, ...
'ImplementationName', 'cblas_dgemm', ...
'ImplementationHeaderFile', 'cblas.h', ...
'ImplementationHeaderPath', LibPath, ...
'AdditionalIncludePaths', {LibPath}, ...
'GenCallback',      'RTW.copyFileToBuildDir', ...
'SideEffects',      true);

```

- 6 Create conceptual arguments `y1`, `u1`, and `u2`. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. To specify a matrix argument in the function call, use the argument class `RTW.Tf1ArgMatrix` and specify the base type and the dimensions for which the argument is valid. This type of table entry supports 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. The conceptual output argument for the `dgemm32` entry for matrix/matrix multiplication replacement specifies dimensions `[2 2; inf inf]`. The conceptual output argument for the `dgemv32` entry for matrix/vector multiplication replacement specifies dimensions `[2 1; inf 1]`.

```

% Specify operands and result
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name', 'y1', ...
    'IOType', 'RTW_IO_OUTPUT', ...
    'BaseType', 'double', ...
    'DimRange', [2 2; inf inf]);
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name', 'u1', ...
    'BaseType', 'double', ...
    'DimRange', [2 2; inf inf]);
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name', 'u2', ...
    'BaseType', 'double', ...
    'DimRange', [1 1; inf inf]);

```

- 7 Create the implementation arguments. This example uses calls to the `getTf1ArgFromString` function to create the arguments. The example code configures special implementation arguments that are required for `dgemm` and `dgemv` function replacements. The convenience methods `setReturn` and `addArgument` specify whether an argument is a return value or argument and adds the argument to the entry's array of implementation arguments.

```

% Using RTW.Tf1CBlasEntryGenerator for xgemm requires the following
% implementation signature:
%
% void f(enum ORDER, enum TRANSA, enum TRANSB, int M, int N, int K,
%         type ALPHA, type* u1, int LDA, type* u2, int LDB,
%         type BETA, type* y, int LDC)
%
% Since CRLs do not have the ability to specify enums, you must

```

```
% use integer. (This will cause problems with C++ code generation,  
% so for C++, use a wrapper function to cast each int to the  
% corresponding enumeration type.)  
%  
% When a match occurs, the code generator computes the  
% values for M, N, K, LDA, LDB, and LDC and insert them into the  
% generated code.  
  
% Specify replacement function signature  
  
arg = getTf1ArgFromString(hTable, 'y2', 'void');  
arg.IOType = 'RTW_IO_OUTPUT';  
op_entry.Implementation.setReturn(arg);  
  
arg = getTf1ArgFromString(hTable, 'ORDER', 'integer', 102);  
%arg.Type.ReadOnly=true;  
op_entry.Implementation.addArgument(arg);  
  
arg = getTf1ArgFromString(hTable, 'TRANSA', 'integer', 111);  
%arg.Type.ReadOnly=true;  
op_entry.Implementation.addArgument(arg);  
  
arg = getTf1ArgFromString(hTable, 'TRANSB', 'integer', 111);  
%arg.Type.ReadOnly=true;  
op_entry.Implementation.addArgument(arg);  
  
arg = getTf1ArgFromString(hTable, 'M', 'integer', 0);  
op_entry.Implementation.addArgument(arg);  
  
arg = getTf1ArgFromString(hTable, 'N', 'integer', 0);  
op_entry.Implementation.addArgument(arg);  
  
arg = getTf1ArgFromString(hTable, 'K', 'integer', 0);  
op_entry.Implementation.addArgument(arg);  
  
arg = getTf1ArgFromString(hTable, 'ALPHA', 'double', 1);  
op_entry.Implementation.addArgument(arg);  
  
arg = getTf1ArgFromString(hTable, 'u1', ['double' '*']);  
op_entry.Implementation.addArgument(arg);  
  
arg = getTf1ArgFromString(hTable, 'LDA', 'integer', 0);  
op_entry.Implementation.addArgument(arg);  
  
arg = getTf1ArgFromString(hTable, 'u2', ['double' '*']);  
op_entry.Implementation.addArgument(arg);  
  
arg = getTf1ArgFromString(hTable, 'LDB', 'integer', 0);  
op_entry.Implementation.addArgument(arg);  
  
arg = getTf1ArgFromString(hTable, 'BETA', 'double', 0);  
op_entry.Implementation.addArgument(arg);  
  
arg = getTf1ArgFromString(hTable, 'y1', ['double' '*']);  
arg.IOType = 'RTW_IO_OUTPUT';
```

```

op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'LDC', 'integer', 0);
op_entry.Implementation.addArgument(arg);

```

- 8 Add the entry to a code replacement table with a call to the addEntryfunction.

```
addEntry(hTable, op_entry);
```

- 9 Create the entry for the second mapping.

```

% Create table entry for cblas_dgemv
op_entry = RTW.Tf1CBlasEntryGenerator;
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW.OP_MUL', ...
    'Priority', 100, ...
    'ImplementationName', 'cblas_dgemv', ...
    'ImplementationHeaderFile', 'cblas.h', ...
    'ImplementationHeaderPath', LibPath, ...
    'AdditionalIncludePaths', {LibPath}, ...
    'GenCallback', 'RTW.copyFileToBuildDir', ...
    'SideEffects', true);

% Specify operands and result
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name', 'y1', ...
    'IOType', 'RTW_IO_OUTPUT', ...
    'BaseType', 'double', ...
    'DimRange', [2 1; inf 1]);
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name', 'u1', ...
    'BaseType', 'double', ...
    'DimRange', [2 2; inf inf]);
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name', 'u2', ...
    'BaseType', 'double', ...
    'DimRange', [1 1; inf 1]);

% Using RTW.Tf1CBlasEntryGenerator for xgemv requires the following
% implementation signature:
%
% void f(enum ORDER, enum TRANSA, int M, int N,
%         type ALPHA, type* u1, int LDA, type* u2, int INCX,
%         type BETA, type* y, int INCY)
%
% Since CRLs do not have the ability to specify enums, you must
% use integer. (This will cause problems with C++ code generation,
% so for C++, use a wrapper function to cast each int to the
% corresponding enumeration type.)
%
% Upon a match, the CRL entry will compute the
% values for M, N, LDA, INCX, and INCY and insert them into the
% generated code.

% Specify replacement function signature

```

```
arg = getTf1ArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.setReturn(arg);

arg = getTf1ArgFromString(hTable, 'ORDER', 'integer', 102);
%arg.Type.ReadOnly=true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'TRANSA', 'integer', 111);
%arg.Type.ReadOnly=true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'M', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'N', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'ALPHA', 'double', 1);
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'u1', ['double' '*']);
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'LDA', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'u2', ['double' '*']);
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'INCX', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'BETA', 'double', 0);
op_entry.Implementation.addArgument(arg);

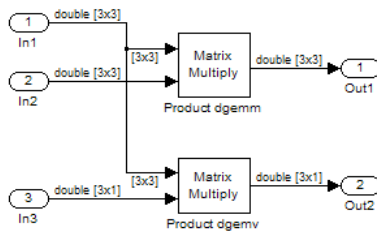
arg = getTf1ArgFromString(hTable, 'y1', ['double' '*']);
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'INCY', 'integer', 0);
op_entry.Implementation.addArgument(arg);

addEntry(hTable, op_entry);
```

To test this example, create a model that uses two Product blocks. For example:

- 1 Create a model that includes two Product blocks, such as the following:



- 2 Configure the model with the following settings:
 - On the **Solver** pane, select a fixed-step, discrete solver with a fixed-step size such as 0.1.
 - On the **Code Generation** pane, select an ERT-based system target file.
 - On the **Code Generation > Interface** pane, select the code replacement library that contains your addition operation entry.
- 3 For each Product block, set the block parameter **Multiplication** to the value `Matrix(*)`.
- 4 In the Model Explorer, configure the **Signal Attributes** for the In1, In2, and In3 source blocks. For In1 and In2, set **Port dimensions** to [3 3] and set the **Data type** to double. For In3, set **Port dimensions** to [3 1] and set the **Data type** to double.
- 5 Generate code and a code generation report.
- 6 Review the code replacements.

Related Examples

- “Define Code Replacement Mappings”
- “Small Matrix Operation to Processor Code Replacement”
- “Matrix Multiplication Operation to MathWorks BLAS Code Replacement”
- “Remap Operator Output to Function Input”
- “Customize Matching and Replacement Process for Operators”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “What Is Code Replacement?”

- “What Is Code Replacement Customization?”
- “Code You Can Replace from MATLAB Code”
- “Code Replacement Match and Replacement Process”
- “Code Replacement Libraries”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Remap Operator Output to Function Input

If your generated code must meet a specific coding pattern or you want more flexibility, for example, to further improve performance, you can remap operator outputs to input positions in an implementation function argument list.

Note: Remapping outputs to implementation function inputs is supported only for operator replacement.

For example, for a sum operation, the code generator produces code similar to:

```
add8_Y.Out1 = u8_add_u8_u8(add8_U.In1, add8_U.In2);
```

If you remap the output to the first input, the code generator produces code similar to:

```
u8_add_u8_u8(&add8_Y.Out1;, add8_U.In1, add8_U.In2);
```

The following table definition file for a sum operation remaps operator output `y1` as the first function input argument.

- 1 Define a function for the definition.

```
function hTable = crl_table_add_uint8
```

- 2 In the call to the `setTf1COperationEntryParameters` function for the operator replacement, set the `SideEffects` property to `true`.

- 3 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 4 Create an entry for the operator mapping with a call to the `RTW.Tf1COperationEntry` function.

```
% Create operation entry
op_entry = RTW.Tf1COperationEntry;
```

- 5 Set function entry parameters with a call to the `setTf1CFunctionEntryParameters` function. In the function call, set the property `SideEffects` to `true`.

```
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_ADD', ...
    'Priority', 90, ...
    'ImplementationName', 'u8_add_u8_u8', ...
```

```

        'ImplementationHeaderFile', 'u8_add_u8_u8.h', ...
        'ImplementationSourceFile', 'u8_add_u8_u8.c', ...
        'SideEffects', true );

```

- 6 Create conceptual arguments `y1`, `u1`, and `u2`. This example uses calls to the `getTf1ArgFromString` and `addConceptualArg` functions to create and add the arguments.

```

arg = getTf1ArgFromString(hTable, 'y1', 'uint8');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(op_entry, arg);

arg = getTf1ArgFromString(hTable, 'u1', 'uint8');
addConceptualArg(op_entry, arg );

arg = getTf1ArgFromString(hTable, 'u2', 'uint8');
addConceptualArg(op_entry, arg );

```

- 7 Create the implementation arguments. This example uses calls to the `getTf1ArgFromString` function to create the arguments. When defining the implementation function return argument, create a new `void` output argument, for example, `y2`. When defining the implementation function argument for the conceptual output argument (`y1`), set the operator output argument as an additional input argument. Mark its `IOType` as output. Make its type a pointer type. The convenience methods `setReturn` and `addArgument` specify whether an argument is a return value or argument and adds the argument to the entry's array of implementation arguments.

```

% Create new void output y2
arg = getTf1ArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.setReturn(arg);

% Set y1 as first input arg, mark IOType as output, and use pointer type
arg=getTf1ArgFromString(hTable, 'y1', 'uint8*');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.addArgument(arg);

arg=getTf1ArgFromString(hTable, 'u1', 'uint8');
op_entry.Implementation.addArgument(arg);

arg=getTf1ArgFromString(hTable, 'u2', 'uint8');
op_entry.Implementation.addArgument(arg);

```

- 8 Add the entry to a code replacement table with a call to the `addEntry` function.

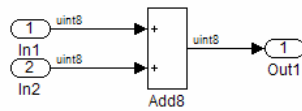
```

addEntry(hTable, op_entry);

```

To test this example, create a model that uses an Add block. For example:

- 1 Create a model that includes an Add block, such as the following:



- 2 Configure the model with the following settings:
 - On the **Solver** pane, select a fixed-step solver.
 - On the **Code Generation** pane, select an ERT-based system target file.
 - On the **Code Generation > Interface** pane, select the code replacement library that contains your addition operation entry.
 - On the **Optimization** pane, set **Signals and Parameters > Optimize global data access** to Use global to hold temporary results. This reduces data copies in the generated code.
- 3 Generate code and a code generation report.
- 4 Review the code replacements.

Related Examples

- “Define Code Replacement Mappings”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code You Can Replace from MATLAB Code”
- “Code Replacement Libraries”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Customize Matching and Replacement Process for Operators

This example shows how to create custom code replacement entries that add extra logic to the code replacement matching and replacement process. Custom entries allow you to specify additional match criteria or modify the replacement function signature to meet your application needs.

- You might want to replace an operator with a particular fixed-point implementation function only when fraction lengths are within a particular range.
- When a match occurs, you might want to modify your replacement function signature based on compile-time information, such as passing fraction-length values into the function.

The example modifies a fixed-point addition replacement such that the implementation function passes in the fraction lengths of the input and output data types as arguments.

Create Class Folder for Entry

Create a class folder using the name of your derived class, such as `@Tf1CustomOperationEntry`. Verify that the class folder is on the MATLAB search path or in your current working folder.

Create Derived Class that Defines `do_match` Method

In the class folder, create and save the following class definition file, `Tf1CustomOperationEntry.m`. This file defines the class `Tf1CustomOperationEntry`, which is derived from the base class `RTW.Tf1COperationEntryML`.

The derived class defines a `do_match` method. In the `do_match` signature:

- `ent` is the return handle, which is returned either as empty (indicating that the match failed) or as a `Tf1COperationEntry` handle.
- `hThis` is the handle to this object.
- `hCSO` is a handle to an object created by the code generator for querying the library for a replacement.
- The remaining arguments are the number of bits for various data types of the current target.

The `do_match` method adds required additional match criteria that the base class does not provide. The method makes required modifications to the implementation signature.

In this case, the `do_match` method can rely on the base class for checking word size and signedness. `do_match` must match only the number of conceptual arguments to the value 3 (two inputs and one output) and the bias for each argument to the value 0. If the code generator finds a match, `do_match` sets the return handle, removes slope and bias wildcards from the conceptual arguments (the match is for specific slope and bias values), and writes fraction-length values for the inputs and output into replacement function arguments 3, 4, and 5.

You can create and add the three additional implementation function arguments for passing fraction lengths in the class definition or in each code replacement entry definition that instantiates this class. This example creates the arguments, adds them to a code replacement table definition file, and sets them to specific values in the class definition code.

```
classdef TflCustomOperationEntry < RTW.TflCOperationEntryML
    methods
        function ent = do_match(hThis, ...
            hCSO, ... %#ok
            targetBitPerChar, ... %#ok
            targetBitPerShort, ... %#ok
            targetBitPerInt, ... %#ok
            targetBitPerLong) %#ok
            % DO_MATCH - Create a custom match function. The base class
            % checks the types of the arguments prior to calling this
            % method. This will check additional data and perhaps modify
            % the implementation function.

            % The base class checks word size and signedness. Slopes and biases
            % have been wildcarded, so the only additional checking to do is
            % to check that the biases are zero and that there are only three
            % conceptual arguments (one output, two inputs)

            ent = []; % default the return to empty, indicating the match failed

            if length(hCSO.ConceptualArgs) == 3 && ...
                hCSO.ConceptualArgs(1).Type.Bias == 0 && ...
                hCSO.ConceptualArgs(2).Type.Bias == 0 && ...
                hCSO.ConceptualArgs(3).Type.Bias == 0

                % Modify the default implementation. Since this is a
                % generator entry, a concrete entry is created using this entry
                % as a template. The type of entry being created is a standard
                % TflCOperationEntry. Using the standard operation entry
                % provides required information, and you do not need
                % a custom match function.
                ent = RTW.TflCOperationEntry(hThis);

                % Since this entry is modifying the implementation for specific
                % fraction-length values (arguments 3, 4, and 5), the conceptual argument
                % wildcards must be removed (the wildcards were inherited from the
                % generator when it was used as a template for the concrete entry).
```

```

% This concrete entry is now for a specific slope and bias.
% hCS0 holds the slope and bias values (created by the code generator).
for idx=1:3
    ent.ConceptualArgs(idx).CheckSlope = true;
    ent.ConceptualArgs(idx).CheckBias = true;

    % Set the specific Slope and Biases
    ent.ConceptualArgs(idx).Type.Slope = hCS0.ConceptualArgs(idx).Type.Slope;
    ent.ConceptualArgs(idx).Type.Bias = 0;
end

% Set the fraction-length values in the implementation function.
ent.Implementation.Arguments(3).Value = ...
    -1.0*hCS0.ConceptualArgs(2).Type.FixedExponent;
ent.Implementation.Arguments(4).Value = ...
    -1.0*hCS0.ConceptualArgs(3).Type.FixedExponent;
ent.Implementation.Arguments(5).Value = ...
    -1.0*hCS0.ConceptualArgs(1).Type.FixedExponent;
end
end
end
end

```

Create Code Replacement Entry

Create code replacement entries that instantiate your custom entry class. For this example, create and save a code replacement table that contains a single operator entry, an entry generator for unsigned 32-bit fixed-point addition operations, with arbitrary fraction-length values on the inputs and the output. This entry instantiates the derived class from the previous step.

If you want to replace all word sizes and signedness attributes (not just 32-bit and unsigned), you can use the same derived class, but not the same entry, because you cannot wildcard the `WordLength` and `IsSigned` arguments. For example, to support `uint8`, `int8`, `uint16`, `int16`, and `int32`, you must add five other distinct entries. Similarly, to use different implementation functions for saturation and rounding modes other than overflow and round to floor, you must add entries for those match permutations.

This table entry creates and adds three implementation arguments to hold the fraction-length values for the inputs and output. Alternatively, the entry can omit those argument definitions. Instead the `do_match` method of the derived class `TflCustomOperationEntry` can create and add the three implementation arguments. When the number of additional implementation arguments required can vary based on compile-time information, use the alternative approach.

- 1 In your working folder, create an entry definition file.

- 2 Define a function for the definition.

```
function hTable = crl_table_custom_add_uFix32
```

- 3 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 4 Create an entry for the custom operator mapping with a call to the `RTW.Tf1CustomOperationEntry` function.

```
%% Add Tf1CustomOperationEntry
```

```
op_entry = Tf1CustomOperationEntry;
```

- 5 Set function entry parameters with a call to the `setTf1COperationEntryParameters` function.

```
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_ADD', ...
    'Priority', 30, ...
    'SaturationMode', 'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes', {'RTW_ROUND_FLOOR'}, ...
    'ImplementationName', 'myFixptAdd', ...
    'ImplementationHeaderFile', 'myFixptAdd.h', ...
    'ImplementationSourceFile', 'myFixptAdd.c');
```

- 6 Create conceptual arguments `y1`, `u1`, and `u2`. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call.

```
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'y1', ...
    'IOType', 'RTW_IO_OUTPUT', ...
    'CheckSlope', false, ...
    'CheckBias', false, ...
    'DataType', 'Fixed', ...
    'Scaling', 'BinaryPoint', ...
    'IsSigned', false, ...
    'WordLength', 32);
```

```
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'u1', ...
    'IOType', 'RTW_IO_INPUT', ...
    'CheckSlope', false, ...
    'CheckBias', false, ...
    'DataType', 'Fixed', ...
    'Scaling', 'BinaryPoint', ...
    'IsSigned', false, ...
    'WordLength', 32);
```

```
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'u2', ...
    'IOType', 'RTW_IO_INPUT', ...
    'CheckSlope', false, ...
```

```

    'CheckBias', false, ...
    'DataType', 'Fixed', ...
    'Scaling', 'BinaryPoint', ...
    'IsSigned', false, ...
    'WordLength', 32);

```

- 7 Create the implementation arguments. This example uses calls to the `createAndSetCImplementationReturn` and `createAndAddImplementationArg` functions to create and add implementation arguments to the entry.

```

% Specify replacement function signature
createAndSetCImplementationReturn(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'y1', ...
    'IOType', 'RTW_IO_OUTPUT', ...
    'IsSigned', false, ...
    'WordLength', 32, ...
    'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'u1', ...
    'IOType', 'RTW_IO_INPUT', ...
    'IsSigned', false, ...
    'WordLength', 32, ...
    'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'u2', ...
    'IOType', 'RTW_IO_INPUT', ...
    'IsSigned', false, ...
    'WordLength', 32, ...
    'FractionLength', 0);

% Add 3 fraction-length args. Actual values are set during code generation.
createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumericConstant', ...
    'Name', 'f1_in1', ...
    'IOType', 'RTW_IO_INPUT', ...
    'IsSigned', false, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value', 0);

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumericConstant', ...
    'Name', 'f1_in2', ...
    'IOType', 'RTW_IO_INPUT', ...
    'IsSigned', false, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value', 0);

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumericConstant', ...
    'Name', 'f1_out', ...
    'IOType', 'RTW_IO_INPUT', ...
    'IsSigned', false, ...
    'WordLength', 32, ...

```

```
'FractionLength', 0, ...  
'Value', 0);
```

- 8 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

Related Examples

- “Define Code Replacement Mappings”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “Code Replacement Match and Replacement Process”
- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code You Can Replace from MATLAB Code”
- “Code Replacement Libraries”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Fixed-Point Operator Code Replacement

In this section...

“Fixed-Point Operator Entries” on page 22-124

“Fixed-Point Numbers and Arithmetic” on page 22-128

“Addition” on page 22-128

“Subtraction” on page 22-129

“Multiplication” on page 22-129

“Division” on page 22-130

“Data Type Conversion (Cast)” on page 22-131

“Shift” on page 22-131

Fixed-Point Operator Entries

If you have a Fixed-Point Designer license, you can define fixed-point operator code replacement entries to match:

- A binary-point-only scaling combination on the operator inputs and output.
- A slope bias scaling combination on the operator inputs and output.
- Relative scaling or net slope between multiplication or division operator inputs and output. Use one of these methods to map a range of slope and bias values to a replacement function for multiplication or division.
- Equal slope and zero net bias across addition or subtraction operator inputs and output. Use this method to disregard specific slope and bias values and map relative slope and bias values to a replacement function for addition or subtraction.

The following table maps common ways to match fixed-point operator code replacement entries with the associated fixed-point parameters that you specify in a code replacement table definition file.

Match	Create entry	Minimally specify parameters
A specific binary-point-only scaling combination on the operator inputs and output.	RTW.Tf1COperationEntry	createAndAddConceptualArg function: <ul style="list-style-type: none"> • CheckSlope: Specify the value true.

Match	Create entry	Minimally specify parameters
		<ul style="list-style-type: none"> • CheckBias: Specify the value <code>true</code>. • DataTypeMode (or DataType/Scaling equivalent): Specify fixed-point binary-point-only scaling. • FractionLength: Specify a fraction length (for example, 3).
A specific slope bias scaling combination on the operator inputs and output.	RTW.Tf1COperationEntry	<p>createAndAddConceptualArg function:</p> <ul style="list-style-type: none"> • CheckSlope: Specify the value <code>true</code>. • CheckBias: Specify the value <code>true</code>. • DataTypeMode (or DataType/Scaling equivalent): Specify fixed-point [slope bias] scaling. • Slope (or SlopeAdjustmentFactor/FixedExponent equivalent): Specify a slope value (for example, 15). • Bias: Specify a bias value (for example, 2).

Match	Create entry	Minimally specify parameters
<p>Net slope between operator inputs and output (multiplication and division).</p>	<p>RTW.Tf1COperationEntry-Generator_NetSlope</p>	<p>setTf1COperationEntryParameters function:</p> <ul style="list-style-type: none"> • NetSlopeAdjustmentFactor: Specify the slope adjustment factor (F) part of the net slope, $F2^E$ (for example, 1.0). • NetFixedExponent: Specify the fixed exponent (E) part of the net slope, $F2^E$ (for example, -3.0). <p>createAndAddConceptualArg function:</p> <ul style="list-style-type: none"> • CheckSlope: Specify the value false. • CheckBias: Specify the value false. • DataType: Specify the value 'Fixed'.

Match	Create entry	Minimally specify parameters
Relative scaling between operator inputs and output (multiplication and division).	RTW.Tf1COperationEntry-Generator	<p>setTf1COperationEntryParameters function:</p> <ul style="list-style-type: none"> • RelativeScalingFactorF: Specify the slope adjustment factor (F) part of the relative scaling factor, $F2^E$ (for example, 1.0). • RelativeScalingFactorE: Specify the fixed exponent (E) part of the relative scaling factor, $F2^E$ (for example, -3.0). <p>createAndAddConceptualArg function:</p> <ul style="list-style-type: none"> • CheckSlope: Specify the value false. • CheckBias: Specify the value false. • DataType: Specify the value 'Fixed'.
Equal slope and zero net bias across operator inputs and output (addition and subtraction).	RTW.Tf1COperationEntry-Generator	<p>setTf1COperationEntryParameters function:</p> <ul style="list-style-type: none"> • SlopesMustBeTheSame: Specify the value true. • MustHaveZeroNetBias: Specify the value true. <p>createAndAddConceptualArg function:</p> <ul style="list-style-type: none"> • CheckSlope: Specify the value false. • CheckBias: Specify the value false.

Fixed-Point Numbers and Arithmetic

Fixed-point numbers use integers and integer arithmetic to represent real numbers and arithmetic with the following encoding scheme:

$$V = \tilde{V} = SQ + B$$

- V is an arbitrarily precise real-world value.
- \tilde{V} is the approximate real-world value that results from fixed-point representation.
- Q is an integer that encodes \tilde{V} , referred to as the *quantized integer*.
- S is a coefficient of Q , referred to as the *slope*.
- B is an additive correction, referred to as the *bias*.

The general equation for an operation between fixed-point operands is:

$$(S_0 Q_0 + B_0) = (S_1 Q_1 + B_1) < op > (S_2 Q_2 + B_2)$$

The objective of fixed-point operator replacement is to replace an operator that accepts and returns fixed-point or integer inputs and output with a function that accepts and returns built-in C numeric data types. The following sections provide additional programming information for each supported operator.

Addition

The operation $V_0 = V_1 + V_2$ implies that

$$Q_0 = \left(\frac{S_1}{S_0} \right) Q_1 + \left(\frac{S_2}{S_0} \right) Q_2 + \left(\frac{B_1 + B_2 - B_0}{S_0} \right)$$

If an addition replacement function is defined such that the scaling on the operands and sum are equal and the net bias

$$\left(\frac{B_1 + B_2 - B_0}{S_0} \right)$$

is zero (for example, a function `s8_add_s8_s8` that adds two signed 8-bit values and produces a signed 8-bit result), then the operator entry must set the operator entry parameters `SlopesMustBeTheSame` and `MustHaveZeroNetBias` to `true`. (For parameter descriptions, see the reference page for the function `setTf1COperationEntryParameters`.)

Subtraction

The operation $V_0 = V_1 - V_2$ implies that

$$Q_0 = \left(\frac{S_1}{S_0} \right) Q_1 - \left(\frac{S_2}{S_0} \right) Q_2 + \left(\frac{B_1 - B_2 - B_0}{S_0} \right)$$

If a subtraction replacement function is defined such that the scaling on the operands and difference are equal and the net bias

$$\left(\frac{B_1 - B_2 - B_0}{S_0} \right)$$

is zero (for example, a function `s8_sub_s8_s8` that subtracts two signed 8-bit values and produces a signed 8-bit result), then the operator entry must set the operator entry parameters `SlopesMustBeTheSame` and `MustHaveZeroNetBias` to `true`. (For parameter descriptions, see the reference page for the function `setTf1COperationEntryParameters`.)

Multiplication

There are different ways to specify multiplication replacements. The most direct way is to specify an exact match of the input and output types. This is feasible if a model contains only a few (known) slope and bias combinations. Use the `Tf1COperationEntry` class

and specify the exact values of slope and bias on each argument. For scenarios where there are numerous slope/bias combinations, it is not feasible to specify each value with a different entry. Use a net slope entry or create a custom entry.

The operation $V_0 = V_1 * V_2$ implies, for binary-point-only scaling, that

$$\begin{aligned}S_0 Q_0 &= (S_1 Q_1)(S_2 Q_2) \\ Q_0 &= \left(\frac{S_1 S_2}{S_0} \right) Q_1 Q_2 \\ Q_0 &= S_n Q_1 Q_2\end{aligned}$$

where S_n is the net slope.

It is common to replace all multiplication operations that have a net slope of 1.0 with a function that performs C-style multiplication. For example, to replace all signed 8-bit multiplications that have a net scaling of 1.0 with the `s8_mul_s8_u8` replacement function, the operator entry must define a net slope factor, $F2^E$. You specify the values for F and E using operator entry parameters `NetSlopeAdjustmentFactor` and `NetFixedExponent`. (For parameter descriptions, see the reference page for the function `setTf1CoperationEntryParameters`.) For the `s8_mul_s8_u8` function, set `NetSlopeAdjustmentFactor` to 1 and `NetFixedExponent` to 0.0.

Note: When an operator entry specifies `NetSlopeAdjustmentFactor` and `NetFixedExponent`, matching entries must have arguments with zero bias.

Division

There are different ways to specify division replacements. The most direct way is to specify an exact match of the input and output types. This is feasible if a model contains only a few (known) slope and bias combinations. For this, use the `Tf1CoperationEntry` class and specify the exact values of slope and bias on each argument. For scenarios where there are numerous slope/bias combinations, it is not feasible to specify each value with a different entry. For this, use a net slope entry or create a custom entry (see “Customize Matching and Replacement Process for Functions”).

The operation $V_0 = (V_1 / V_2)$ implies, for binary-point-only scaling, that

$$S_0 Q_0 = \left(\frac{S_1 Q_1}{S_2 Q_2} \right)$$

$$Q_0 = S_n \left(\frac{Q_1}{Q_2} \right)$$

where S_n is the net slope.

It is common to replace all division operations that have a net slope of 1.0 with a function that performs C-style division. For example, to replace all signed 8-bit divisions that have a net scaling of 1.0 with the `s8_mul_s8_u8_` replacement function, the operator entry must define a net slope factor, $F2^E$. You specify the values for F and E using operator entry parameters `NetSlopeAdjustmentFactor` and `NetFixedExponent`. (For parameter descriptions, see the reference page for the function `setTf1COperationEntryParameters`.) For the `s16_netslope0p5_div_s16_s16` function, you would set `NetSlopeAdjustmentFactor` to 1 and `NetFixedExponent` to 0.0.

Note: When an operator entry specifies `NetSlopeAdjustmentFactor` and `NetFixedExponent`, matching entries must have arguments with zero bias.

Data Type Conversion (Cast)

The data type conversion operation $V_0 = V_1$ implies, for binary-point-only scaling, that

$$Q_0 = \left(\frac{S_1}{S_0} \right) Q_1$$

$$Q_0 = S_n Q_1$$

where S_n is the net slope.

Shift

The shift left or shift right operation $V_0 = (V_1 / 2^n)$ implies, for binary-point-only scaling, that

$$S_0 Q_0 = \left(\frac{S_1 Q_1}{2^n} \right)$$
$$Q_0 = \left(\frac{S_1}{S_0} \right) + \left(\frac{Q_1}{2^n} \right)$$
$$Q_0 = S_n \left(\frac{Q_1}{2^n} \right)$$

where S_n is the net slope.

Related Examples

- “Define Code Replacement Mappings”
- “Binary-Point-Only Scaling Code Replacement”
- “Slope Bias Scaling Code Replacement”
- “Net Slope Scaling Code Replacement”
- “Equal Slope and Zero Net Bias Code Replacement”
- “Data Type Conversions (Casts) and Operator Code Replacement”
- “Shift Left Operations and Code Replacement”
- “Remap Operator Output to Function Input”
- “Customize Matching and Replacement Process for Operators”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code You Can Replace from MATLAB Code”
- “Code Replacement Match and Replacement Process”
- “Code Replacement Libraries”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Binary-Point-Only Scaling Code Replacement

You can define code replacement entries for operations on fixed-point data types such that they match a binary-point-only scaling combination on operator inputs and output. These binary-point-only scaling entries can map the specified binary-point-scaling combination to a replacement function for addition, subtraction, multiplication, or division.

This example creates a code replacement entry for multiplication of fixed-point data types. You specify arguments using binary-point-only scaling. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

- 1 Define a function for the definition. For example:

```
function hTable = crl_table_fixed_binptscale
```

- 2 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 3 Create the entry for the operator mapping with a call to the `RTW.Tf1COperationEntry` function.

```
op_entry = RTW.Tf1COperationEntry;
```

- 4 Set operator entry parameters with a call to the `setTf1COperationEntryParameters` function. The parameters specify the type of operation as multiplication, the saturation mode as saturate on overflow, rounding modes as unspecified, and the name of the replacement function as `s32_mul_s16_s16_binarypoint`.

```
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_MUL', ...
    'Priority', 90, ...
    'SaturationMode', 'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes', {'RTW_ROUND_UNSPECIFIED'}, ...
    'ImplementationName', 's32_mul_s16_s16_binarypoint', ...
    'ImplementationHeaderFile', 's32_mul_s16_s16_binarypoint.h', ...
    'ImplementationSourceFile', 's32_mul_s16_s16_binarypoint.c');
```

- 5 Create conceptual arguments `y1`, `u1`, and `u2`. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. Each argument specifies that the data type is fixed-point, the mode is binary-point-only scaling, and its derived slope and bias values must exactly match the call-site slope and bias values. The output argument is 32 bits, signed, with a

fraction length of 28. The input arguments are 16 bits, signed, with fraction lengths of 15 and 13.

```
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric',...
    'Name',          'y1', ...
    'IOType',        'RTW_IO_OUTPUT', ...
    'CheckSlope',    true, ...
    'CheckBias',     true, ...
    'DataTypeMode',  'Fixed-point: binary point scaling', ...
    'IsSigned',      true, ...
    'WordLength',    32, ...
    'FractionLength', 28);

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'u1', ...
    'IOType',        'RTW_IO_INPUT', ...
    'CheckSlope',    true, ...
    'CheckBias',     true, ...
    'DataTypeMode',  'Fixed-point: binary point scaling', ...
    'IsSigned',      true, ...
    'WordLength',    16, ...
    'FractionLength', 15);

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'u2', ...
    'IOType',        'RTW_IO_INPUT', ...
    'CheckSlope',    true, ...
    'CheckBias',     true, ...
    'DataTypeMode',  'Fixed-point: binary point scaling', ...
    'IsSigned',      true, ...
    'WordLength',    16, ...
    'FractionLength', 13);
```

- 6 Create the implementation arguments. This example uses calls to the `createAndSetCImplementationReturn` and `createAndAddImplementationArg` functions to create and add implementation arguments to the entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types). In this case, the output argument is 32 bits and signed (`int32`). The input arguments are 16 bits and signed (`int16`).

```
createAndSetCImplementationReturn(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'y1', ...
    'IOType',        'RTW_IO_OUTPUT', ...
    'IsSigned',      true, ...
    'WordLength',    32, ...
    'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric',...
    'Name',          'u1', ...
    'IOType',        'RTW_IO_INPUT', ...
    'IsSigned',      true, ...
    'WordLength',    16, ...
    'FractionLength', 0);
```

```
createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric',...
    'Name',          'u2', ...
    'IOType',       'RTW_IO_INPUT', ...
    'IsSigned',     true, ...
    'WordLength',   16, ...
    'FractionLength', 0);
```

- 7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

Related Examples

- “Define Code Replacement Mappings”
- “Fixed-Point Operator Code Replacement”
- “Slope Bias Scaling Code Replacement”
- “Net Slope Scaling Code Replacement”
- “Equal Slope and Zero Net Bias Code Replacement”
- “Data Type Conversions (Casts) and Operator Code Replacement”
- “Shift Left Operations and Code Replacement”
- “Remap Operator Output to Function Input”
- “Customize Matching and Replacement Process for Operators”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code You Can Replace from MATLAB Code”
- “Code Replacement Match and Replacement Process”
- “Code Replacement Libraries”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Slope Bias Scaling Code Replacement

You can define code replacement for operations on fixed-point data types as matching a slope bias scaling combination on the operator inputs and output. The slope bias scaling entries can map the specified slope bias combination to a replacement function for addition, subtraction, multiplication, or division.

This example creates a code replacement entry for division of fixed-point data types. You specify arguments using slope bias scaling. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

- 1 Define a function for the definition. For example:

```
function hTable = cr1_table_fixed_s16divslopebias
```

- 2 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 3 Create the entry for the operator mapping with a call to the `RTW.Tf1COperationEntry` function.

```
op_entry = RTW.Tf1COperationEntry;
```

- 4 Set operator entry parameters with a call to the `setTf1COperationEntryParameters` function. The parameters specify the type of operation as division, the saturation mode as saturate on overflow, rounding modes as round to ceiling, and the name of the replacement function as `s16_div_s16_s16_slopebias`.

```
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_DIV', ...
    'Priority', 90, ...
    'SaturationMode', 'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes', {'RTW_ROUND_CEILING'}, ...
    'ImplementationName', 's16_div_s16_s16_slopebias', ...
    'ImplementationHeaderFile', 's16_div_s16_s16_slopebias.h', ...
    'ImplementationSourceFile', 's16_div_s16_s16_slopebias.c');
```

- 5 Create conceptual arguments `y1`, `u1`, and `u2`. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. Each argument specifies that the data type is fixed-point, the mode is slope bias scaling, and its specified slope and bias values must exactly match the call-site slope and bias values. The output argument and input arguments are 16 bits, signed, each with specific slope bias specifications.

```

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    '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);

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'u1', ...
    'IOType',       'RTW_IO_INPUT', ...
    'CheckSlope',   true, ...
    'CheckBias',    true, ...
    'DataTypeMode', 'Fixed-point: slope and bias scaling', ...
    'IsSigned',     true, ...
    'WordLength',   16, ...
    'Slope',        15, ...
    'Bias',         2);

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'u2', ...
    'IOType',       'RTW_IO_INPUT', ...
    'CheckSlope',   true, ...
    'CheckBias',    true, ...
    'DataTypeMode', 'Fixed-point: slope and bias scaling', ...
    'IsSigned',     true, ...
    'WordLength',   16, ...
    'Slope',        13, ...
    'Bias',         5);

```

- 6 Create the implementation arguments. This example uses calls to the `createAndSetCImplementationReturn` and `createAndAddImplementationArg` functions to create and add implementation arguments to the entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types). In this case, the output and input arguments are 16 bits and signed (`int16`).

```

createAndSetCImplementationReturn(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'y1', ...
    'IOType',       'RTW_IO_OUTPUT', ...
    'IsSigned',     true, ...
    'WordLength',   16, ...
    'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'u1', ...
    'IOType',       'RTW_IO_INPUT', ...
    'IsSigned',     true, ...
    'WordLength',   16, ...
    'FractionLength', 0);

```

```
createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric',...
    'Name',          'u2', ...
    'IOType',        'RTW_IO_INPUT', ...
    'IsSigned',      true, ...
    'WordLength',    16, ...
    'FractionLength', 0);
```

- 7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

Related Examples

- “Define Code Replacement Mappings”
- “Fixed-Point Operator Code Replacement”
- “Binary-Point-Only Scaling Code Replacement”
- “Net Slope Scaling Code Replacement”
- “Equal Slope and Zero Net Bias Code Replacement”
- “Data Type Conversions (Casts) and Operator Code Replacement”
- “Shift Left Operations and Code Replacement”
- “Remap Operator Output to Function Input”
- “Customize Matching and Replacement Process for Operators”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code You Can Replace from MATLAB Code”
- “Code Replacement Match and Replacement Process”
- “Code Replacement Libraries”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Net Slope Scaling Code Replacement

In this section...

“Multiplication and Division with Saturation” on page 22-140

“Multiplication and Division with Rounding Mode and Additional Implementation Arguments” on page 22-143

Multiplication and Division with Saturation

You can define code replacement entries for operations on fixed-point data types as matching net slope between operator inputs and output. The net slope entries can map a range of slope and bias values to a replacement function for multiplication or division.

This example creates a code replacement entry for division of fixed-point data types, using wrap on overflow saturation mode and a net slope. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

- 1 Define a function for the definition. For example:

```
function hTable = crl_table_fixed_netslopesaturate
```

- 2 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 3 Create the entry for the operator mapping with a call to the `RTW.Tf1COperationEntryGenerator_Netslope` function, which provides access to the fixed-point parameters `NetSlopeAdjustmentFactor` and `NetFixedExponent`.

```
wv = [16,32];
for iy = 1:2
    for inum = 1:2
        for iden = 1:2
            hTable = getDivOpEntry(hTable, ...
                fixdt(1,wv(iy)),fixdt(1,wv(inum)),fixdt(1,wv(iden)));
        end
    end
end
```

```
%-----
function hTable = getDivOpEntry(hTable, dty, dtnum, dtiden)
%-----
```



```
% Create an entry for division of fixed-point data types where
% arguments are specified using Slope and Bias scaling
% Saturation on, Rounding unspecified
```

```
funcStr = sprintf('user_div_%s_%s_%s',...
    typeStrFunc(dty),...
    typeStrFunc(dtnum),...
    typeStrFunc(dtdden));
```

```
op_entry = RTW.Tf1COperationEntryGenerator_NetSlope;
```

- 4 Set operator entry parameters with a call to the `setTf1COperationEntryParameters` function. The parameters specify the type of operation as division, the saturation mode as wrap on overflow, rounding modes as unspecified, and the name of the replacement function as `user_div_*`. `NetSlopeAdjustmentFactor` and `NetFixedExponent` specify the F and E parts of the net slope $F2^E$.

```
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_DIV', ...
    'Priority', 90, ...
    'SaturationMode', 'RTW_WRAP_ON_OVERFLOW',...
    'RoundingModes', {'RTW_ROUND_UNSPECIFIED'},...
    'NetSlopeAdjustmentFactor', 1.0, ...
    'NetFixedExponent', 0.0, ...
    'ImplementationName', funcStr, ...
    'ImplementationHeaderFile', [funcStr, '.h'], ...
    'ImplementationSourceFile', [funcStr, '.c']);
```

- 5 Create conceptual arguments `y1`, `u1`, and `u2`. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. Specify each argument as fixed-point and signed. Also, for each argument, specify that code replacement request processing does *not* check for an exact match to the call-site slope and bias values.

```
createAndAddConceptualArg(op_entry, ...
    'RTW.Tf1ArgNumeric', ...
    'Name', 'y1',...
    'IOType', 'RTW_IO_OUTPUT',...
    'CheckSlope', false,...
    'CheckBias', false,...
    'DataTypeMode', 'Fixed-point: slope and bias scaling',...
    'IsSigned', dty.Signed,...
    'WordLength', dty.WordLength,...
    'Bias', 0);
```

```
createAndAddConceptualArg(op_entry, ...
    'RTW.Tf1ArgNumeric', ...
    'Name', 'u1', ...
    'IOType', 'RTW_IO_INPUT',...
    'CheckSlope', false,...
    'CheckBias', false,...
```

```

        'DataTypeMode', 'Fixed-point: slope and bias scaling',...
        'IsSigned',    dtnum.Signed,...
        'WordLength',  dtnum.WordLength,...
        'Bias',        0);

createAndAddConceptualArg(op_entry, ...
    'RTW.Tf1ArgNumeric', ...
    'Name',             'u2', ...
    'IOType',           'RTW_IO_INPUT',...
    'CheckSlope',      false,...
    'CheckBias',       false,...
    'DataTypeMode',    'Fixed-point: slope and bias scaling',...
    'IsSigned',        dtden.Signed,...
    'WordLength',      dtden.WordLength,...
    'Bias',            0);

```

- 6 Create the implementation arguments. This example uses calls to the `getTf1ArgFromString` function to create the arguments. Implementation arguments must describe fundamental numeric data types (not fixed-point data types). The convenience methods `setReturn` and `addArgument` specify whether an argument is a return value or argument. These methods add the argument to the entry array of implementation arguments.

```

arg = getTf1ArgFromString(hTable, 'y1', typeStrBase(dty));
op_entry.Implementation.setReturn(arg);

arg = getTf1ArgFromString(hTable, 'u1', typeStrBase(dtnum));
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'u2', typeStrBase(dtden));
op_entry.Implementation.addArgument(arg);

```

- 7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

- 8 Define functions that determine the data type word length.

```

%-----
function str = typeStrFunc(dt)
%-----

if dt.Signed
    sstr = 's';
else
    sstr = 'u';
end
str = sprintf('%s%d', sstr, dt.WordLength);

%-----
function str = typeStrBase(dt)
%-----

if dt.Signed

```

```

    sstr = ;
else
    sstr = 'u';
end
str = sprintf('%sint%d',sstr,dt.WordLength);

```

Multiplication and Division with Rounding Mode and Additional Implementation Arguments

You can define code replacement entries for multiplication and division operations on fixed-point data types such that they match the net slope between operator inputs and output. The net slope entries can map a range of slope and bias values to a replacement function for multiplication or division.

This example creates a code replacement entry for division of fixed-point data types, using the ceiling rounding mode and a net slope scaling factor. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

- 1 Define a function for the definition. For example:

```
function hTable = crl_table_fixed_netsloperound
```

- 2 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 3 Create the entry for the operator mapping with a call to the `RTW.Tf1COperationEntryGenerator_Netslope` function, which provides access to the fixed-point parameters `NetSlopeAdjustmentFactor` and `NetFixedExponent`.

```
op_entry = RTW.Tf1COperationEntryGenerator_NetSlope;
```

- 4 Set operator entry parameters with a call to the `setTf1COperationEntryParameters` function. The parameters specify the type of operation as division, the saturation mode as saturation off, rounding modes as round to ceiling, and the name of the replacement function as `s16_div_s16_s16`. `NetSlopeAdjustmentFactor` and `NetFixedExponent` specify the F and E parts of the relative scaling factor $F2^E$.

```

setTf1COperationEntryParameters(op_entry, ...
    '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');

```

- 5 Create conceptual arguments `y1`, `u1`, and `u2`. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. Specify each argument as fixed-point, 16 bits, and signed. Also, for each argument, specify that code replacement request processing does *not* check for an exact match to the call-site slope and bias values.

```

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'y1', ...
    'IOType', 'RTW_IO_OUTPUT', ...
    'CheckSlope', false, ...
    'CheckBias', false, ...
    'DataType', 'Fixed', ...
    'IsSigned', true, ...
    'WordLength', 16);

```

```

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'u1', ...
    'IOType', 'RTW_IO_INPUT', ...
    'CheckSlope', false, ...
    'CheckBias', false, ...
    'DataType', 'Fixed', ...
    'IsSigned', true, ...
    'WordLength', 16);

```

```

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'u2', ...
    'IOType', 'RTW_IO_INPUT', ...
    'CheckSlope', false, ...
    'CheckBias', false, ...
    'DataType', 'Fixed', ...
    'IsSigned', true, ...
    'WordLength', 16);

```

- 6 Create the implementation arguments. This example uses calls to the `createAndSetCImplementationReturn` and `createAndAddImplementationArg` functions to create and add implementation arguments to the entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types). In this case, the output and input arguments are 16 bits and signed (`int16`).

```

createAndSetCImplementationReturn(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'y1', ...
    'IOType', 'RTW_IO_OUTPUT', ...
    'IsSigned', true, ...
    'WordLength', 16, ...
    'FractionLength', 0);

```

```

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric',...
    'Name', 'u1', ...
    'IOType', 'RTW_IO_INPUT', ...
    'IsSigned', true, ...
    'WordLength', 16, ...
    'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric',...
    'Name', 'u2', ...
    'IOType', 'RTW_IO_INPUT', ...
    'IsSigned', true, ...
    'WordLength', 16, ...
    'FractionLength', 0);

```

- 7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

Related Examples

- “Define Code Replacement Mappings”
- “Fixed-Point Operator Code Replacement”
- “Binary-Point-Only Scaling Code Replacement”
- “Slope Bias Scaling Code Replacement”
- “Equal Slope and Zero Net Bias Code Replacement”
- “Data Type Conversions (Casts) and Operator Code Replacement”
- “Shift Left Operations and Code Replacement”
- “Remap Operator Output to Function Input”
- “Customize Matching and Replacement Process for Operators”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code You Can Replace from MATLAB Code”
- “Code Replacement Match and Replacement Process”
- “Code Replacement Libraries”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Equal Slope and Zero Net Bias Code Replacement

You can define code replacement entries for addition or subtraction of fixed-point data types such that they match relative slope and bias values (equal slope and zero net bias) across operator inputs and output. These entries allow you to disregard slope and bias values. Map relative slope and bias values to a replacement function for addition or subtraction.

This example creates a code replacement entry for addition of fixed-point data types. Slopes must be equal and net bias must be zero across the operator inputs and output. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

- 1 Define a function for the definition.

```
function hTable = crl_table_fixed_slopeseq_netbiaszero
```

- 2 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 3 Create the entry for the operator mapping with a call to the `RTW.Tf1COperationEntryGenerator` function, which provides access to the fixed-point parameters `SlopesMustBeTheSame` and `MustHaveZeroNetBias`.

```
op_entry = RTW.Tf1COperationEntryGenerator;
```

- 4 Set operator entry parameters with a call to the `setTf1COperationEntryParameters` function. The parameters specify the type of operation as addition, the saturation mode as saturation off, rounding modes as unspecified, and the name of the replacement function as `u16_add_SameSlopeZeroBias`. `SlopesMustBeTheSame` and `MustHaveZeroNetBias` are set to `true`, indicating that slopes must be equal and net bias must be zero across the addition inputs and output.

```
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_ADD', ...
    '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');
```

- 5 Create conceptual arguments `y1`, `u1`, and `u2`. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. Each argument is specified as 16 bits and unsigned. Each argument specifies that code replacement request processing does *not* check for an exact match to the call-site slope and bias values.

```
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'y1', ...
    'IOType',        'RTW_IO_OUTPUT', ...
    'CheckSlope',    false, ...
    'CheckBias',     false, ...
    'IsSigned',      false, ...
    'WordLength',    16);

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'u1', ...
    'IOType',        'RTW_IO_INPUT', ...
    'CheckSlope',    false, ...
    'CheckBias',     false, ...
    'IsSigned',      false, ...
    'WordLength',    16);

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'u2', ...
    'IOType',        'RTW_IO_INPUT', ...
    'CheckSlope',    false, ...
    'CheckBias',     false, ...
    'IsSigned',      false, ...
    'WordLength',    16);
```

- 6 Create the implementation arguments. This example uses calls to the `createAndSetCImplementationReturn` and `createAndAddImplementationArg` functions to create and add implementation arguments to the entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types). In this case, the output and input arguments are 16 bits and unsigned (`uint16`).

```
createAndSetCImplementationReturn(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'y1', ...
    'IOType',        'RTW_IO_OUTPUT', ...
    'IsSigned',      false, ...
    'WordLength',    16, ...
    'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'u1', ...
    'IOType',        'RTW_IO_INPUT', ...
    'IsSigned',      false, ...
    'WordLength',    16, ...
    'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric', ...
```



```

    'Name',          'u2', ...
    'IOType',       'RTW_IO_INPUT', ...
    'IsSigned',     false, ...
    'WordLength',   16, ...
    'FractionLength', 0);

```

- 7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

Related Examples

- “Define Code Replacement Mappings”
- “Fixed-Point Operator Code Replacement”
- “Binary-Point-Only Scaling Code Replacement”
- “Slope Bias Scaling Code Replacement”
- “Net Slope Scaling Code Replacement”
- “Data Type Conversions (Casts) and Operator Code Replacement”
- “Shift Left Operations and Code Replacement”
- “Remap Operator Output to Function Input”
- “Customize Matching and Replacement Process for Operators”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code You Can Replace from MATLAB Code”
- “Code Replacement Match and Replacement Process”
- “Code Replacement Libraries”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Data Type Conversions (Casts) and Operator Code Replacement

You can use code replacement entries to replace code that the code generator produces for data type conversion (cast) operations.

This example creates a code replacement entry that replaces `int32` to `int16` data type conversion (cast) operations. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

- 1 Define a function for the definition.

```
function hTable = crl_table_cast_int32_to_int16
```

- 2 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 3 Create the entry for the operator mapping with a call to the `RTW.Tf1COperationEntry` function.

```
op_entry = RTW.Tf1COperationEntry;
```

- 4 Set operator entry parameters with a call to the `setTf1COperationEntryParameters` function. The parameters specify the type of operation as cast, the saturation mode as saturate on overflow, rounding modes as toward negative infinity, and the name of the replacement function as `my_sat_cast`.

```
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_CAST', ...
    'Priority', 50, ...
    'ImplementationName', 'my_sat_cast', ...
    'SaturationMode', 'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes', {'RTW_ROUND_FLOOR'}, ...
    'ImplementationHeaderFile', 'some_hdr.h', ...
    'ImplementationSourceFile', 'some_hdr.c');
```

- 5 Create the `int16` argument as conceptual argument `y1` and the implementation return value. This example uses calls to the `getTf1ArgFromString` and `addConceptualArg` functions to create the conceptual argument and add it to the entry. Convenience method `setReturn` specifies the argument as the implementation return value.

```
arg = getTf1ArgFromString(hTable, 'y1', 'int16');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(op_entry, arg);
op_entry.Implementation.setReturn(arg);
```

- 6 Create the `int32` argument as conceptual and implementation argument `u1`. This example uses calls to the `getTf1ArgFromString` and `addConceptualArg` functions to create the conceptual argument and add it to the entry. Convenience method `addArgument` specifies the argument as implementation input argument.

```
arg = getTf1ArgFromString(hTable, 'u1', 'int32');
addConceptualArg(op_entry, arg);
op_entry.Implementation.addArgument(arg);
```

- 7 Add the entry to a code replacement table with a call to the `addEntry` function.

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

You can use code replacement entries to replace code that the code generator produces for data type conversion (cast) operations.

This example creates a code replacement entry to replace data type conversions (casts) of fixed-point data types by using a net slope. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

- 1 Define a function for the definition.

```
function hTable = crl_table_cast_fixpt_net_slope
```

- 2 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 3 Create the entry for the operator mapping with a call to the `RTW.Tf1COperationEntryGenerator_Netslope` function, which provides access to the fixed-point parameters `NetSlopeAdjustmentFactor` and `NetFixedExponent`

```
op_entry = RTW.Tf1COperationEntryGenerator_NetSlope;
```

- 4 Set operator entry parameters with a call to the `setTf1COperationEntryParameters` function. The parameters specify the type of operation as cast, the saturation mode as saturate on overflow, rounding modes as toward negative infinity, and the name of the replacement function as `my_fxp_cast`. `NetSlopeAdjustmentFactor` and `NetFixedExponent` specify the F and E parts of the net slope $F2^E$.

```
InFL = 2;
InWL = 16;
InSgn = true;
OutFL = 4;
```

```

OutWL = 32;
OutSgn = true;
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_CAST', ...
    'Priority', 50, ...
    'SaturationMode', 'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes', {'RTW_ROUND_FLOOR'}, ...
    'NetSlopeAdjustmentFactor', 1.0, ...
    'NetFixedExponent', (OutFL - InFL), ...
    'ImplementationName', 'my_fxp_cast', ...
    'ImplementationHeaderFile', 'some_hdr.h', ...
    'ImplementationSourceFile', 'some_hdr.c');

```

- 5 Create conceptual arguments `y1` and `u1`. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. Each argument is specified as fixed-point and signed. Each argument specifies that code replacement request processing does *not* check for an exact match to the call-site slope and bias values.

```

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'y1', ...
    'IOType', 'RTW_IO_OUTPUT', ...
    'CheckSlope', false, ...
    'CheckBias', false, ...
    'DataTypeMode', 'Fixed-point: binary point scaling', ...
    'IsSigned', OutSgn, ...
    'WordLength', OutWL, ...
    'FractionLength', OutFL);

```

```

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'u1', ...
    'IOType', 'RTW_IO_INPUT', ...
    'CheckSlope', false, ...
    'CheckBias', false, ...
    'DataTypeMode', 'Fixed-point: binary point scaling', ...
    'IsSigned', InSgn, ...
    'WordLength', InWL, ...
    'FractionLength', InFL);

```

- 6 Create the implementation arguments. This example uses calls to the `createAndSetCImplementationReturn` and `createAndAddImplementationArg` functions to create and add implementation arguments to the entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types).

```

createAndSetCImplementationReturn(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'y1', ...
    'IOType', 'RTW_IO_OUTPUT', ...
    'IsSigned', OutSgn, ...
    'WordLength', OutWL, ...
    'FractionLength', 0);

```

```

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric', ...

```

```

    'Name',           'u1', ...
    'IOType',        'RTW_IO_INPUT', ...
    'IsSigned',      InSgn, ...
    'WordLength',    InWL, ...
    'FractionLength', 0);

```

- 7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

Related Examples

- “Define Code Replacement Mappings”
- “Fixed-Point Operator Code Replacement”
- “Binary-Point-Only Scaling Code Replacement”
- “Slope Bias Scaling Code Replacement”
- “Net Slope Scaling Code Replacement”
- “Equal Slope and Zero Net Bias Code Replacement”
- “Shift Left Operations and Code Replacement”
- “Remap Operator Output to Function Input”
- “Customize Matching and Replacement Process for Operators”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code You Can Replace from MATLAB Code”
- “Code Replacement Match and Replacement Process”
- “Code Replacement Libraries”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Shift Left Operations and Code Replacement

You can use code replacement entries to replace code that the code generator produces for shift (<<) operations.

This example creates a code replacement entry to replace shift left operations for `int16` data. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

- 1 Define a function for the definition.

```
function hTable = crl_table_shift_left_int16
```

- 2 Create a table to contain the entry with a call to the `RTW.Tf1Table` function.

```
hTable = RTW.Tf1Table;
```

- 3 Create the entry for the operator mapping with a call to the `RTW.Tf1COperationEntry` function.

```
op_entry = RTW.Tf1COperationEntry;
```

- 4 Set operator entry parameters with a call to the `setTf1COperationEntryParameters` function. The parameters specify the type of operation as shift left and the name of the replacement function as `my_shift_left`.

```
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_SL', ...
    'Priority', 50, ...
    'ImplementationName', 'my_shift_left', ...
    'ImplementationHeaderFile', 'some_hdr.h', ...
    'ImplementationSourceFile', 'some_hdr.c');
```

- 5 Create the `int16` argument as conceptual argument `y1` and the implementation return value. This example uses calls to the `getTf1ArgFromString` and `addConceptualArg` functions to create the conceptual argument and add it to the entry. Convenience method `setReturn` specifies the argument as the implementation return value.

```
arg = getTf1ArgFromString(hTable, 'y1', 'int16');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(op_entry, arg);
op_entry.Implementation.setReturn(arg);
```

- 6 Create the `int16` argument as conceptual and implementation argument `u1`. This example uses calls to the `getTf1ArgFromString` and `addConceptualArg` functions to create the conceptual argument and add it to the entry. Convenience method `addArgument` specifies the argument as an implementation input argument.

```
arg = getTflArgFromString(hTable, 'u1', 'int16');
addConceptualArg(op_entry, arg);
op_entry.Implementation.addArgument(arg);
```

- 7 Create the `int8` argument as conceptual and implementation argument `u2`. This example uses calls to the `getTflArgFromString` and `addConceptualArg` functions to create the conceptual argument and add it to the entry. This argument specifies the number of bits to shift the previous input argument. Because the argument type is not relevant, the example disables type checking by setting the `CheckType` property to `false`. Convenience method `addArgument` specifies the argument as implementation input argument.

```
arg = getTflArgFromString(hTable, 'u2', 'int8');
arg.CheckType = false;
addConceptualArg(op_entry, arg);
op_entry.Implementation.addArgument(arg);
```

- The function `getTflArgFromString` is called to create an `int8` input argument. This argument is added to the operator entry both as the third conceptual argument and the second implementation input argument.
- Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

You can use code replacement entries to replace code that the code generator produces for shift (`<<`) operations.

This example creates a code replacement entry to replace shift left operations for fixed-point data using a net slope. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

- 1 Define a function for the definition.

```
function hTable = crl_table_shift_left_fixpt_net_slope
```

- 2 Create a table to contain the entry with a call to the `RTW.TflTable` function.

```
hTable = RTW.TflTable;
```

- 3 Create the entry for the operator mapping with a call to the `RTW.TflCOperationEntryGenerator_Netslope` function. This function provides access to the fixed-point parameters `NetslopeAdjustmentFactor` and `NetFixedExponent`.

```
op_entry = RTW.TflCOperationEntryGenerator_NetSlope;
```

- 4 Set operator entry parameters with a call to the `setTf1COperationEntryParameters` function. The parameters specify the type of operation as shift left, the saturation mode as saturate on overflow, rounding modes as toward negative infinity, and the name of the replacement function as `my_fxp_shift_left`. `NetSlopeAdjustmentFactor` and `NetFixedExponent` specify the F and E parts of the net slope $F2^E$.

```
InFL = 2;
InWL = 16;
InSgn = true;
OutFL = 4;
OutWL = 32;
OutSgn = true;
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_SL', ...
    'Priority', 50, ...
    'SaturationMode', 'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes', {'RTW_ROUND_FLOOR'}, ...
    'NetSlopeAdjustmentFactor', 1.0, ...
    'NetFixedExponent', (OutFL - InFL), ...
    'ImplementationName', 'my_fxp_shift_left', ...
    'ImplementationHeaderFile', 'some_hdr.h', ...
    'ImplementationSourceFile', 'some_hdr.c');
```

- 5 Create conceptual arguments `y1` and `u1`. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. Each argument is specified as fixed-point and signed. Each argument specifies that code replacement request processing does *not* check for an exact match to the call-site slope and bias values.

```
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'y1', ...
    'IOType', 'RTW_IO_OUTPUT', ...
    'CheckSlope', false, ...
    'CheckBias', false, ...
    'DataTypeMode', 'Fixed-point: binary point scaling', ...
    'IsSigned', OutSgn, ...
    'WordLength', OutWL, ...
    'FractionLength', OutFL);

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'u1', ...
    'IOType', 'RTW_IO_INPUT', ...
    'CheckSlope', false, ...
    'CheckBias', false, ...
    'DataTypeMode', 'Fixed-point: binary point scaling', ...
    'IsSigned', InSgn, ...
    'WordLength', InWL, ...
    'FractionLength', InFL);
```


- 6 Create the implementation arguments. This example uses calls to the `createAndSetCImplementationReturn` and `createAndAddImplementationArg` functions to create and add implementation arguments to the entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types).

```
createAndSetCImplementationReturn(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'y1', ...
    'IOType',        'RTW_IO_OUTPUT', ...
    'IsSigned',      OutSgn, ...
    'WordLength',    OutWL, ...
    'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'u1', ...
    'IOType',        'RTW_IO_INPUT', ...
    'IsSigned',      InSgn, ...
    'WordLength',    InWL, ...
    'FractionLength', 0);
```

- 7 Create the `int8` argument as conceptual and implementation argument `u2`. This example uses calls to the `getTf1ArgFromString` and `addConceptualArg` functions to create the conceptual argument and add it to the entry. This argument specifies the number of bits to shift the previous input argument. Because the argument type is not relevant, type checking is disabled by setting the `CheckType` property to `false`. Convenience method `addArgument` specifies the argument as implementation input argument.

```
arg = getTf1ArgFromString(hTable, 'u2', 'uint8');
arg.CheckType = false;
addConceptualArg(op_entry, arg);
op_entry.Implementation.addArgument(arg);
```

- 8 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

Related Examples

- “Define Code Replacement Mappings”
- “Fixed-Point Operator Code Replacement”
- “Binary-Point-Only Scaling Code Replacement”
- “Slope Bias Scaling Code Replacement”
- “Net Slope Scaling Code Replacement”
- “Equal Slope and Zero Net Bias Code Replacement”
- “Data Type Conversions (Casts) and Operator Code Replacement”

- “Remap Operator Output to Function Input”
- “Customize Matching and Replacement Process for Operators”
- “Develop a Code Replacement Library”
- “Quick Start Library Development”

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code You Can Replace from MATLAB Code”
- “Code Replacement Match and Replacement Process”
- “Code Replacement Libraries”
- “Code Replacement Terminology”

External Web Sites

- “Replacing Math Functions and Operators”

Performance

Configuration

- “Configure Code Optimizations” on page 23-2
- “Specify Global Variable Localization” on page 23-5
- “Set Hardware Implementation Parameters” on page 23-6
- “Use External Mode with the ERT Target” on page 23-7

Configure Code Optimizations

Several parameters available on the **Optimization** panes configure your model to optimize code generation. The following table includes parameters on the **Optimization > General** pane:

To...	Select or Specify...
Remove initialization code for root-level inports and outports with a value of zero.	Select Remove root level I/O zero initialization .
Generate additional code to set float and double storage explicitly to value 0.0.	<p>Select Use memset to initialize floats and doubles to 0.0 When you set this parameter, the <code>memset</code> function clears internal storage (regardless of type) to the integer bit pattern 0 (that is, all bits are off). The additional code generated when the option is off is slightly less efficient.</p> <p>If the representation of floating-point zero that your compiler and target CPU use is identical to the integer bit pattern 0, you can gain efficiency by setting this parameter.</p>
Suppress the generation of code that initializes internal work structures (for example, block states and block outputs) to zero.	Select Remove internal state zero initialization .
Generate run-time initialization code for a block that has states only if the block is in a system that can reset its states, such as an enabled subsystem.	<p>Select Optimize initialization code for model reference This results in more efficient code.</p> <p>The following restrictions apply to using the Optimize initialization code for model reference parameter. However, these restrictions do not apply to a Model block that references a function-call model.</p> <ul style="list-style-type: none"> In a subsystem that resets states, do not include a Model block that references a model that has this parameter set to <code>on</code>. For example, in an enabled subsystem with the States when enabling block parameter set to <code>reset</code>, do not include a Model block that references a model that has

To...	Select or Specify...
	<p>the Optimize initialization code for model reference parameter set to <code>on</code>.</p> <ul style="list-style-type: none"> If you set the Optimize initialization code for model reference parameter to <code>off</code> in a model that includes a Model block that directly references a referenced model, do not set the Optimize initialization code for model reference parameter for the referenced model to <code>on</code>.
Remove wrapping code that handles out-of-range floating-point to integer conversion results.	<p>Select Remove code from floating-point to integer conversions that wraps out-of-range values. This reduces the size and increases the speed of the generated code at the cost of potentially producing results that do not match simulation in the case of out-of-range values.</p>
Suppress generation of code that guards against fixed-point division by zero.	<p>Select Remove code that protects against division arithmetic exceptions. When you select this parameter, simulation results and results from generated code may not be in bit-for-bit agreement.</p>
To minimize the amount of memory allocated for absolute and elapsed time counters.	<p>Specify an integer value for Application lifespan (days). For more information on the allocation and operation of absolute and elapsed timers, see “Timers”, “Use Timers in Asynchronous Tasks”, and “Control Memory Allocation for Time Counters” in the Simulink Coder documentation.</p>

The following table includes optimization parameters on the **Optimization > Signals and Parameters** pane:

To...	Select or Specify...
Control whether parameter data for reusable subsystems is generated in a separate header file for each subsystem or in a single parameter data structure	Select Hierarchical or NonHierarchical for Parameter structure .
Replace multiply operations in array indices when accessing arrays in a loop	Select Simplify array indexing .

To...	Select or Specify...
Store Boolean signals as one-bit bitfields instead of as a Boolean data type	Select Pack Boolean data into bitfields . Selecting this parameter enables the Bitfield declarator type specifier . To optimize your code further, select <code>uchar_T</code> , however this optimization benefit is dependent on your choice of target.
Pass each reusable subsystem output argument as an address of a local to reduce global memory usage and eliminate copying local variables back to global block I/O structures	Select Individual arguments for Pass reusable subsystem outputs as .

Specify Global Variable Localization

When you generate code for a model, the code generator can optimize variable references by replacing global variables with local variables. Replacing global variables with local variables improves execution speed and reduces RAM/ROM. Creating more local variables can increase stack usage.

Some of the global variables that the code generator can localize include:

- Global signals that cross subsystem boundaries.
- Global signals across Simulink and Stateflow domains.
- Unused global state variables.
- Redundant local Data Store Memory block signals.

To enable the global variable localization analysis:

- 1** In the Configuration Parameters dialog box, on the **Code Generation** pane, in the **System target file** box, specify an ERT target.
- 2** Verify that the `OptimizeBlockIOStorage` parameter is set to 'on':

```
>> get_param(gcs, 'OptimizeBlockIOStorage')
ans =
    on
```

- 3** Verify that `AdvancedOptControl` is not set to '-SLCI':

```
>> get_param(gcs, 'AdvancedOptControl')
ans =
    ''
```

- 4** Set the storage class for signals to `Auto`.

The code generator does not localize global variables for MATLAB system objects or AUTOSAR.

Set Hardware Implementation Parameters

Specification of target-specific characteristics of generated code (such as word sizes for `char`, `short`, `int`, and `long` data types, or desired rounding behaviors in integer operations) can be critical in embedded systems development. The **Hardware Implementation** category of options in a configuration set provides a simple and flexible way to control such characteristics in both simulation and code generation.

Before generating and deploying code, you should become familiar with the options on the **Hardware Implementation** pane of the Configuration Parameters dialog box. See “Hardware Implementation Pane” in the Simulink documentation and “Platform Options for Development and Deployment” in the Simulink Coder documentation for full details on the **Hardware Implementation** pane.

By configuring the **Hardware Implementation** properties of your model's active configuration set to match the behaviors of your compiler and hardware, you can generate more efficient code. For example, if you specify the **Byte ordering** property, you can avoid generation of extra code that tests the byte ordering of the target CPU.

You can use the `rtwdemo_targetsettings` demo model to determine some implementation-dependent characteristics of your C or C++ compiler, as well as characteristics of your target hardware. By using this model in conjunction with your target development system and debugger, you can observe the behavior of the code as it executes on the target. You can then use this information to configure the **Hardware Implementation** parameters of your model.

To use this model, type the command

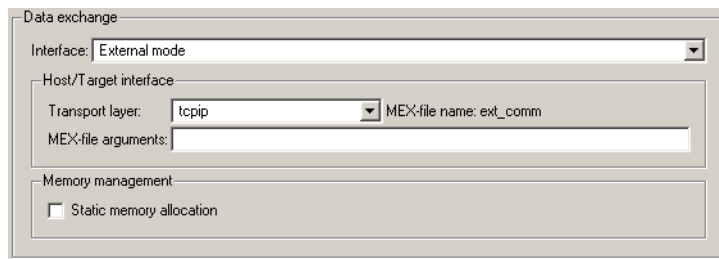
```
rtwdemo_targetsettings
```

Follow the instructions in the model window.

Use External Mode with the ERT Target

Selecting the **External mode** option turns on generation of code to support external mode communication between host (Simulink) and target systems. The Embedded Coder software supports Simulink external mode features, as described in the “Host/Target Communication” section of the Simulink Coder documentation.

This section discusses external mode options that may be of special interest to embedded systems designers. The next figure shows the **Data Exchange** subpane of the Configuration Parameters dialog box, **Interface** pane, with **External mode** selected.



Memory Management

Consider the **Memory management** option **Static memory allocation** before generating external mode code for an embedded target. Static memory allocation is generally desirable, as it reduces overhead and promotes deterministic performance.

When you select the **Static memory allocation** option, static external mode communication buffers are allocated in the target application. When **Static memory allocation** is deselected, communication buffers are allocated dynamically (with `malloc`) at run time.

Generation of Pure Integer Code with External Mode

The Embedded Coder software supports generation of pure integer code when external mode code is generated. To do this, select the **External mode** option, and deselect the **Support floating-point numbers** option in the **Software environment** section of the **Interface** pane.

This enhancement lets you generate external mode code that is free of storage definitions of double or float data type, and allows your code to run on integer-only processors

If you intend to generate pure integer code with **External mode** on, note the following requirements:

- All trigger signals must be of data type `int32`. Use a Data Type Conversion block if needed.
- When pure integer code is generated, the simulation stop time specified in the **Solver** options is ignored. To specify a stop time, run your target application from the MATLAB command line and use the `-tf` option. (See “Run the External Program” in the Simulink Coder documentation.) If you do not specify this option, the application executes indefinitely (as if the stop time were `inf`).

When executing pure integer target applications, the stop time specified by the `-tf` command line option is interpreted as the number of base rate ticks to execute, rather than as an elapsed time in seconds. The number of ticks is computed as

$$\text{stop time in seconds} / \text{base rate step size in seconds}$$

Code Execution Profiling

- “Execution Profiling for Generated Code” on page 24-2
- “Code Execution Profiling for SIL and PIL” on page 24-5
- “Configure Code Execution Profiling for SIL and PIL” on page 24-6
- “Execution Profiling for Atomic Subsystems and Model Reference Hierarchies” on page 24-8
- “View and Compare Code Execution Times” on page 24-10
- “Analyze Code Execution Data” on page 24-16
- “Tips and Limitations” on page 24-18

Execution Profiling for Generated Code

Use code execution profiling to determine:

- Whether the generated code meets real-time requirements of your target hardware.
- Code sections that require performance improvements.

The following tasks represent a general workflow that uses code execution profiling:

- 1 With the Simulink model, design and optimize your algorithm.
- 2 Configure the model for code execution profiling, and generate code.
- 3 Execute generated code on target.
- 4 Analyze performance through code execution profiling plots and reports. For example, check that the algorithm code satisfies real-time requirements:
 - If the algorithm code easily meets the requirements, consider enhancing your algorithm to exploit available processing power.
 - If the code cannot be executed in real time, look for ways to reduce execution time.

Identify the tasks that require the most time. For these tasks, investigate whether trade-offs between functionality and speed are possible.

If your target is a multicore processor, distribute the execution of algorithm code across available cores.

- 5 If required, refine the model and return to step 2.

To find information about code execution profiling with Simulink products, use the following table.

Target	Execution Feature	Type of Profiling	Relevant Products	See
Host computer	Model configured for concurrent execution	Execution time	Simulink Coder	<ul style="list-style-type: none"> • “Build and Download to a Multicore Target” • “Concurrent Execution Example Models”
Host computer	Software-in-the-loop (SIL)	Execution time	Embedded Coder	<ul style="list-style-type: none"> • “Code Execution Profiling for SIL and PIL”

Target	Execution Feature	Type of Profiling	Relevant Products	See
				<ul style="list-style-type: none"> • “Configure Code Execution Profiling for SIL and PIL” • “Execution Profiling for Atomic Subsystems and Model Reference Hierarchies” • “View and Compare Code Execution Times” • “Analyze Code Execution Data”
Embedded hardware or instruction set simulator	Processor-in-the-loop (PIL)	Execution time	Embedded Coder	<ul style="list-style-type: none"> • “Code Execution Profiling for SIL and PIL” • “Configure Code Execution Profiling for SIL and PIL” • “Execution Profiling for Atomic Subsystems and Model Reference Hierarchies” • “View and Compare Code Execution Times” • “Analyze Code Execution Data”
Target support packages	Standalone execution, PIL	Execution time	Embedded Coder	<ul style="list-style-type: none"> • “Code Execution Profiling for IDE and Toolchain Targets” • “Perform Execution Time Profiling for IDE and Toolchain Targets”
Target support packages	Standalone execution	Stack	Embedded Coder	<ul style="list-style-type: none"> • “Code Execution Profiling for IDE and Toolchain Targets” • “Perform Stack Profiling with IDE and Toolchain Targets”

Target	Execution Feature	Type of Profiling	Relevant Products	See
Simulink Real-Time™	Hardware-in-the-loop (HIL)	Execution time	Simulink Coder, Simulink Real-Time	<ul style="list-style-type: none"> • “Execution Profiling for Real-Time Applications” • “Configure Real-Time Application for Profiling” • “Generate Real-Time Application Execution Profile”
Simulink Real-Time	Model configured for concurrent execution, HIL	Execution time	Simulink Coder, Simulink Real-Time	<ul style="list-style-type: none"> • “Execution Profiling for Real-Time Applications” • “Concurrent Execution with Simulink® Real-Time™”

Code Execution Profiling for SIL and PIL

During a software-in-the-loop (“SIL”) or processor-in-the-loop (“PIL”) simulation, you can produce a profile of execution times for tasks and functions in your generated code. The software calculates execution times from data that is obtained through instrumentation probes added to the SIL or PIL test harness or placed inside generated code.

Use the execution time profile to check whether your code runs in real time on your target hardware:

- If code execution overruns, look for ways to reduce execution time. For example:
 - 1 Identify tasks that require the most time.
 - 2 In these tasks, investigate whether trade-offs between functionality and speed are possible.
 - 3 Make the changes that reduce execution time.
- If your code easily meets real-time requirements, consider enhancing functionality to exploit the unused processing power.

Note: Tasks are main entry points into the generated code. For example, the step function for a sample rate or the *model_initialize* function.

The software collects execution time measurements in a variable that you specify. See “Configure Code Execution Profiling for SIL and PIL” on page 24-6.

At the end of the SIL or PIL simulation, you can:

- View a report of code execution times.
- Use the Simulation Data Inspector to view and compare plots of function execution times.
- Analyze the measurements within the MATLAB environment.

See “View and Compare Code Execution Times” on page 24-10 and “Analyze Code Execution Data” on page 24-16.

Configure Code Execution Profiling for SIL and PIL


To configure code execution profiling for a “SIL” or “PIL” simulation:

- 1 In your top model, open the Configuration Parameters dialog box, and select the **Code Generation > Verification** pane.
- 2 Select the **Measure task execution time** check box.
- 3 If you also want function execution times, select the **Measure function execution times** check box. For information about how you apply this setting to atomic subsystems and model reference hierarchies, see “Execution Profiling for Atomic Subsystems and Model Reference Hierarchies” on page 24-8.
- 4 In the **Workspace variable** field, specify a name. When you run the simulation, the software generates a variable with this name in the MATLAB base workspace. The variable contains the execution time measurements, and is an object of type `coder.profile.ExecutionTime`.

If you select the **Data Import/Export > Save simulation output as single object** check box, the software creates the variable in the `Simulink.SimulationOutput` object that you specify.

- 5 From the **Save options** drop-down list, select one of the following:
 - **Summary data only** — If you want to generate only a report and reduce memory usage, for example, during a long simulation. See “View and Compare Code Execution Times” on page 24-10.
 - **All measurement and analysis data** — Allows you to generate a report and store execution time profiles in the base workspace. After the simulation, you can use methods from the `coder.profile.ExecutionTime` and `coder.profile.ExecutionTimeSection` classes to retrieve execution time measurements for every call to each profiled section of code that occurs during the simulation. See “Analyze Code Execution Data” on page 24-16.
- 6 Click **OK**.

For a PIL simulation, you must configure a hardware-specific timer. When you set up the connectivity configuration for your target, create a timer object. See “Create PIL Target Connectivity Configuration”. This action is not required for a SIL simulation.

If you select **All measurement and analysis data** from the **Save options** drop-down list, the generated report displays Simulation Data Inspector icons . When you

click one of these icons, the software imports simulation results into the Simulation Data Inspector. You can then plot execution times and manage and compare plots from various simulations. See “View and Compare Code Execution Times” on page 24-10.

For details about automatically importing simulation data into the Simulation Data Inspector, see “Validate System Behavior”.

Execution Profiling for Atomic Subsystems and Model Reference Hierarchies

To generate execution data for tasks only, on the **Code Generation > Verification** pane of the Configuration Parameters dialog box, select the **Measure task execution time** check box and clear the **Measure function execution times** check box.

To generate function execution data for atomic subsystems in the top model, on the **Code Generation > Verification** pane, you must select the **Measure task execution time** and **Measure function execution times** check boxes.

The generation of function execution data requires the insertion of measurement probes into the generated code. The software inserts measurement probes for an atomic subsystem only if you set the **Function packaging** field (on the **Code Generation** tab of the Function Block Parameters dialog box) to either **Nonreusable function** or **Reusable function**. If the field is set to **Auto**, then the insertion of probes depends on the packaging choice that results from the **Auto** setting. If the field is set to **Inline**, the software does not insert probes.

Note: In the generated code, the software wraps each function call with measurement probes except when:

- The call site cannot be wrapped because of expression folding (see “Minimize Computations and Storage for Intermediate Results”).
 - The call site is located in the shared utility code (see “Sharing Utility Code”).
-

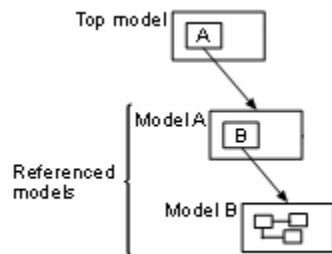
You might not want to generate profiles for specific subsystems. To disable code execution profiling for a subsystem in the top model:

- 1 Right-click the subsystem.
- 2 From the context menu, select **Properties**.
- 3 In the Block Properties dialog box, select the **General** tab.
- 4 In the **Tag** field, enter `DoNotProfile`.
- 5 Click **OK**.

To generate function execution data for model reference hierarchies:

- 1 In the top model, open the Configuration Parameters dialog box, and select the **Code Generation > Verification** pane.
- 2 Select the **Measure task execution time** check box.
- 3 For each Model block that you want to profile, select **Measure function execution times** only at the reference level for which you require function execution data.

For example, consider a top model that has Model block A, which in turn contains Model block B.



If you want to generate execution times for functions from model B, select **Measure task execution time** for the top model and **Measure function execution times** for model B.

Note: By default, the Model block parameter, **Code under test** is set to **Model** reference. If this block parameter is set to **Top model**, the configuration parameter **Measure task execution time** for the top model and the referenced model must be the same. Otherwise, the software produces an error.

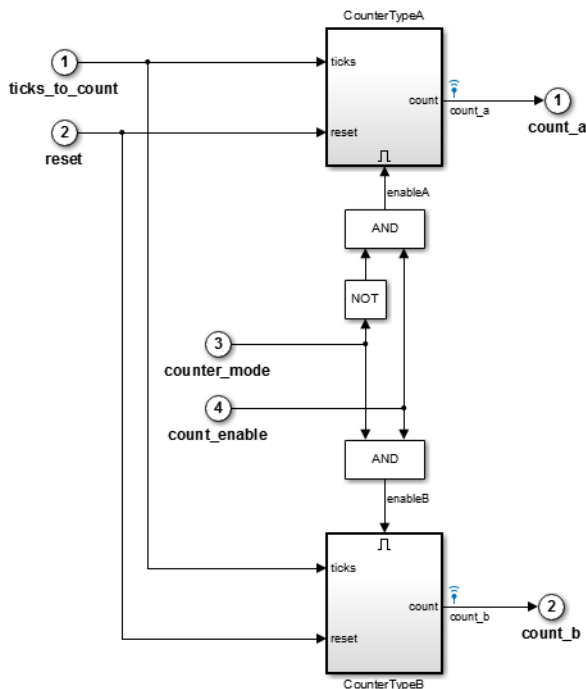
If your top model has a PIL block, the execution profiling settings that apply to the PIL block are the settings from the original model that you used to create the PIL block. See “Use a SIL or PIL Block”. The execution profiling settings of your top model do not apply to the PIL block.

View and Compare Code Execution Times

After a SIL or PIL simulation, you can:

- Use `report` to open a report of code execution times.
- Use the Simulation Data Inspector to:
 - Plot execution times.
 - Manage and compare plots from various simulations.

Consider the model `rtwdemo_sil_topmodel`, which has two subsystems `CounterTypeA` and `CounterTypeB`.



To generate code execution times for the subsystems, on the **Configuration Parameters > Code Generation > Verification** pane:

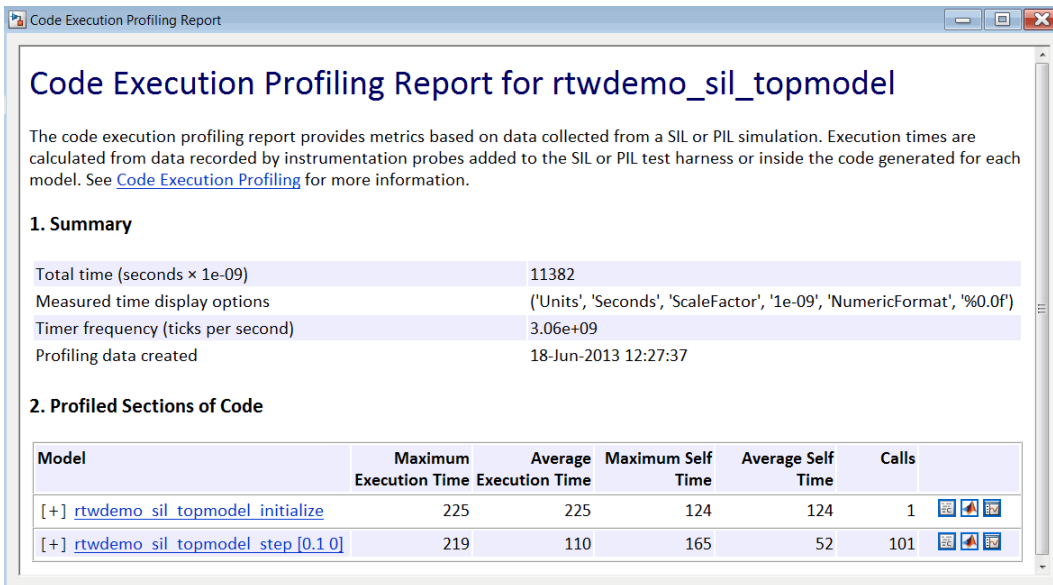
- 1 Select the following check boxes:
 - Measure task execution time
 - Measure function execution times
- 2 Specify a **Workspace variable**, for example, `executionProfile`.
- 3 From the **Save options** drop-down list, select **All measurement and analysis data**.

When you run a SIL simulation, the software generates the variable `executionProfile` in the MATLAB base workspace.

Note: If you select the **Data Import/Export > Save simulation output as single object** check box, the software creates the variable in your specified `Simulink.SimulationOutput` object.

To display a code execution report, in the Command Window, enter:

























```
>> report(executionProfile)
```



Part 1 provides a summary. Part 2 contains information about profiled code sections.


You can expand and collapse profiled sections in Part 2 by clicking [+] and [-] respectively. The following graphic shows fully expanded sections.

2. Profiled Sections of Code

Model	Maximum Execution Time	Average Execution Time	Maximum Self Time	Average Self Time	Calls	
[-] rtwdemo_sil_topmodel_initialize	225	225	124	124	1	  
[-] CounterTypeA	44	44	26	26	1	  
CounterTypeA	18	18	18	18	1	  
[-] CounterTypeB	56	56	41	41	1	  
CounterTypeB	16	16	16	16	1	  
[-] rtwdemo_sil_topmodel_step [0.1 0]	219	110	165	52	101	  
CounterTypeA	46	36	46	36	101	  
CounterTypeB	37	22	37	22	101	  

The report contains time measurements for:

- The model initialization function `rtwdemo_sil_topmodel_initialize`.
- A task represented by the step function `rtwdemo_sil_topmodel_step [0.1 0]`.
- Functions generated from the subsystems `CounterTypeA` and `CounterTypeB`.

You can go to a profiled code section in the Generated Code view of the Code Generation Report. In the Code Execution Profiling Report, on a code section row, click the icon . For example, if you click the icon in the `rtwdemo_sil_topmodel_initialize` row, you see the measurement probes around the call site in the SIL test harness.

```

59  XIL\_INTERFACE\_ERROR\_CODE xilInitialize(uint32_T xilFcnId) {
60      XIL\_INTERFACE\_ERROR\_CODE errorCode = XIL\_INTERFACE\_SUCCESS;
61      /* initialize output storage owned by In-the-Loop */
62      /* Single In-the-Loop Component */
63      if (xilFcnId == 0) {
64          PROFILE\_START\_TASK\_SECTION\(1U\);
65          rtwdemo\_sil\_topmodel\_initialize();
66          PROFILE\_END\_TASK\_SECTION\(1U\);
67      }
68      else {
69          errorCode = XIL\_INTERFACE\_UNKNOWN\_FCNDID;
70      }
71      return errorCode;
72  }

```



By default, the report displays time in nanoseconds (10^{-9} seconds). You can specify the time unit and numeric display format. For example, to display time in microseconds (10^{-6} seconds), use the following command:

```
>>report(executionProfile,'Units','Seconds','ScaleFactor','1e-06','NumericFormat','%0.3f')
```

The report displays time in seconds only if the timer is calibrated, that is, the number of timer ticks per second is known. On a Windows machine, the software automatically determines this value for a SIL simulation. On a Linux machine, you must manually calibrate the timer. For example, if your processor speed is 1 GHz, specify the number of timer ticks per second:

```
>>executionProfile.TimerTicksPerSecond = 1e9;
```

For more information, see “Analyze Code Execution Data” on page 24-16.



To display measured execution times for a task or function, click the Simulation Data Inspector icon  on the corresponding row. You can use the Simulation Data Inspector to manage and compare plots from various simulations.




For information, see “Inspect Signal Data with Simulation Data Inspector”.

Note: To observe how code sections are invoked over the execution timeline, use the `timeline` function.

The following table describes the information provided in the code section profiles.

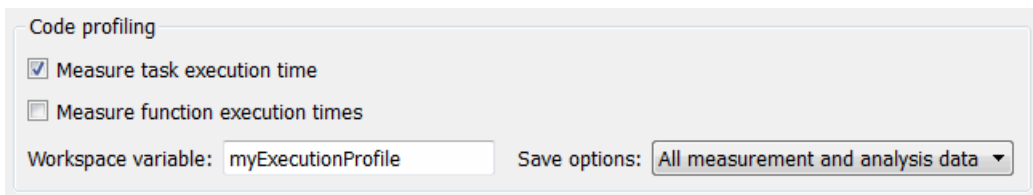
Column	Description
Model	Name of task, top model, subsystem, or Model block. Click the link to go to the model. With a task, the sample period and sample offset are listed next to the task name. For example, <code>rtwdemo_sil_topmodel_step [0.1 0]</code> indicates that the sample period is 0.1 seconds and the sample offset is 0.
Maximum Execution Time	Maximum time between start and end of function execution. Includes time spent in child functions.
Average Execution Time	Average time between start and end of execution. Includes time spent in child functions.
Maximum Self Time	Longest time spent in function. Excludes time spent in child functions.
Average Self Time	Average time spent in function. Excludes time spent in child functions.
Calls	Number of calls made to task or function.
	Icon that you click in the Code Execution Profiling Report to see the profiled code section in the Generated Code view of the Code Generation Report. Code section can be a task or function. The specified workspace variable, for example, <code>executionProfile</code> , must be present in the base workspace.
	Icon that you click to display the profiled code section in the Command Window. Equivalent to executing the command <code>executionProfile.Sections(i)</code> . The specified workspace variable, for example, <code>executionProfile</code> , must be present in the base workspace.

Column	Description
	<p>Icon that you click to display measured execution times with Simulation Data Inspector.</p> <p>The specified workspace variable, for example, <code>executionProfile</code>, must be present in the base workspace.</p>

Analyze Code Execution Data

After a SIL or PIL simulation, you can analyze execution time data using methods from the `coder.profile.ExecutionTime` and `coder.profile.ExecutionTimeSection` classes.

Consider the model `rtwdemo_sil_topmodel`. Specify the following profiling options, and then run a SIL simulation.



Code profiling

Measure task execution time

Measure function execution times

Workspace variable: Save options:

The software generates the workspace variable `myExecutionProfile`, an `coder.profile.ExecutionTime` object.

To get the total number of code sections that have profiling data, use the `Sections` method.

```
>> no_of_Sections = myExecutionProfile.Sections
no_of_Sections =
    2
>>
```

To get the `coder.profile.ExecutionTimeSection` object for a profiled code section, use the method `Sections`.

```
>> FirstSectionProfile = myExecutionProfile.Sections(1)
coder.profile.ExecutionTimeTaskSection
    Section name = rtwdemo_sil_topmodel_initialize
    Sample period = 0
    Sample offset = 0
>>
>> SecondSectionProfile = myExecutionProfile.Sections(2)
coder.profile.ExecutionTimeTaskSection
    Section name = rtwdemo_sil_topmodel_step [0.1 0]
    Sample period = 0.1
    Sample offset = 0
```

```
>>
```

Use `coder.profile.ExecutionTimeSection` methods to extract profiling information for a particular code section. For example, use `Name` to obtain the name of a profiled task.

```
>> name_of_section = SecondSectionProfile.Name
```

```
name_of_section =
```

```
rtwdemo_sil_topmodel_step [0.1 0]
```

```
>>
```

If the timer is uncalibrated and you know the timer rate, for example 2.2 GHz, you can use the `coder.profile.ExecutionTime` method `TimerTicksPerSecond` to calibrate the timer:

```
>> myExecutionProfile.TimerTicksPerSecond(2.2e9)
```

```
>> SecondSectionProfile = myExecutionProfile.Sections(2);
```

```
>>
```

Tips and Limitations

Triggered Model Block

Consider the case where a triggered Model block is configured to run in the SIL or PIL simulation mode. The software generates one execution time measurement each time the referenced model is triggered to run. If there are multiple triggers in a single time step, the software generates multiple measurements for the triggered Model block. Conversely, if there is no trigger in a given time step, the software generates no time measurements.

Outliers in Execution Time Profiles

After the start of a measurement, the operating system might preempt a SIL application, making the code execution profiling result for the time step unreliable. As a consequence of preemption, you might see outliers in your execution time profiles, with execution times that are longer than expected.

For execution time measurements greater than 2^{32} ticks, the counter wraps. (If you are profiling a real-time, multi-core application, the software reduces the effective measurement range of the timer to accommodate synchronization discrepancies in the recording of timer values for different cores.) Counter wrapping occurs when the actual execution time is very long. Counter wrapping results in a *measured* execution time that is shorter than expected.

Hardware-Specific Timer

If your target configuration does not already specify a timer, you must specify one. To specify a timer, you must create a timer object that provides details of the hardware-specific timer and associated source files:

- For SIL simulation, the timer word length is determined by your MEX compiler. The word length is 64 bits, unless your selected MEX compiler is Lcc. In this case, the word length is 32 bits.
- For PIL simulation, you can specify an 8-, 16-, 32-, or 64-bit timer.

Task Context Switching Due to Preemption

Profiling instrumentation is intrusive and affects the quantity that it is meant to measure. Therefore, the design goal is to maximize code understanding with a minimum

of instrumentation. For example, with a real-time system, there can be task context switches due to preemption. These context switches are not explicitly instrumented. To record the start and end of each task, the software must infer context switches from instrumentation. As a result, the software reports behavior that is an estimate. The estimate is subject to error because of incomplete instrumentation within the kernel.

In some cases, when the software cannot accurately determine behavior, the software generates a warning:

Warning: Analysis unsuccessful for one or more profiling data points. ... For example, the software might generate this warning when not all mutex take system calls (associated with rate transitions) are instrumented. In the case of Simulink Real-Time, this situation might arise if you generate code for a model reference hierarchy without enabling function profiling for all referenced models (`set_param(model, 'CodeProfilingInstrumentation', 'on')`). If a mutex take system call is not instrumented, a task context switch might occur that is not visible to the execution profiling analysis.

In other cases, although the software cannot accurately determine behavior, the software does *not* generate a warning.

Data Type Replacement Support

Data type replacement does not support the measurement of function execution times. For your model, clear one of the following check boxes:

- **Configuration Parameters > Code Generation > Verification > Measure function execution times**
- **Configuration Parameters > Code Generation > Data Type Replacement > Replace data type names in the generated code**

Code Execution Profiling for MATLAB Coder

- “Execution Time Profiling for SIL and PIL” on page 25-2
- “Generate Execution Time Profile” on page 25-3
- “View Execution Time Profile” on page 25-4
- “Analyze Execution Time Data” on page 25-7

Execution Time Profiling for SIL and PIL

During a software-in-the-loop (SIL) or processor-in-the-loop (PIL) execution, you can produce a profile of execution times for code generated from entry-point functions. The software calculates execution times from data that is obtained through instrumentation probes added to the SIL or PIL test harness.

Use the execution time profile to check whether your code runs within the required time on your target hardware:

- If code execution overruns, look for ways to reduce execution time.
- If your code easily meets time requirements, consider enhancing functionality to exploit the unused processing power.

At the end of the SIL or PIL execution, you can:

- View a report of code execution times.
- Use the Simulation Data Inspector to view and compare plots of function execution times.
- Access and analyze execution time profiling data.

Note: SIL and PIL execution supports multiple entry-point functions. An entry-point function can call another entry-point function as a subfunction. However, the software generates execution time profiles only for functions that are called at the entry-point level. The software does not generate execution time profiles for entry-point functions that are called as subfunctions by other entry-point functions.

For more information, see:

- “Generate Execution Time Profile” on page 25-3
- “View Execution Time Profile” on page 25-4
- “Analyze Execution Time Data” on page 25-7

Generate Execution Time Profile

Before running a software-in-the-loop (SIL) or processor-in-the-loop (PIL) execution, enable execution time profiling using the Project Interface:

- 1 Select the **Build** tab.
- 2 Under **Settings**, click **More settings**, which opens the Project Settings dialog box.
- 3 From the **All Settings > Advanced > Enable entry point execution profiling for SIL/PIL** drop-down list, select **Yes**.

Or, from the Command Window, specify the `CodeExecutionProfiling` property of your `coder.EmbeddedCodeConfig` object. For example:

```
config.CodeExecutionProfiling = true;
```

For information about setting up a SIL or PIL execution, see “Code Verification”.

View Execution Time Profile

If you enable SIL or PIL execution time profiling, at the end of the execution, the software generates a link in the Command Window. For example:

```
Current plot held
### Starting SIL execution for 'kalman01'
    To terminate execution: clear kalman01_sil
    Execution profiling report available after termination.
Current plot released
```

Click the `clear kalman01_sil` link. The software terminates the execution process and displays a new link.

```
### Stopping SIL execution for 'kalman01'
    Execution profiling report: report(getCoderExecutionProfile('kalman01'))
```

Click the report link. The software opens the Code Execution Profiling Report.







Code Execution Profiling Report for kalman01

The code execution profiling report provides metrics based on data collected from a SIL or PIL execution. Execution times are calculated from data recorded by instrumentation probes added to the SIL or PIL test harness or inside the code generated for each component. See [Code Execution Profiling](#) for more information.

1. Summary

Total time (seconds \times 1e-09)	2206501
Measured time display options	('Units', 'Seconds', 'ScaleFactor', '1e-09', 'NumericFormat', '%0.0f')
Timer frequency (ticks per second)	3.06e+09
Profiling data created	01-Apr-2014 10:21:25

2. Profiled Sections of Code

Section	Maximum Execution Time	Average Execution Time	Maximum Self Time	Average Self Time	Calls	
kalman01_initialize	1076	1076	1076	1076	1	 
kalman01	16009	7351	16009	7351	300	 
kalman01_terminate	138	138	138	138	1	 

The first section provides a summary. The second section contains information about profiled code sections.

The report contains time measurements for:

- The *entry_point_fn_initialize* function, for example, `kalman01_initialize`.
- The entry-point function, for example, `kalman01`.

- The `entry_point_fn_terminate` function, for example, `kalman01_terminate`.


By default, the report displays time in nanoseconds (10^{-9} seconds). You can specify the time unit and numeric display format. For example, to display time in microseconds (10^{-6} seconds), use the “report” command:

```
executionProfile=getCoderExecutionProfile('kalman01'); % Create workspace var
report(executionProfile, ...
    'Units', 'Seconds', ...
    'ScaleFactor', '1e-06', ...
    'NumericFormat', '%0.3f')
```

The report displays time in seconds only if the timer is calibrated, that is, the number of timer ticks per second is established. On a Windows machine, the software determines this value for a SIL simulation. On a Linux machine, you must manually calibrate the timer. For example, if your processor speed is 1 GHz, specify the number of timer ticks per second:



```
executionProfile.TimerTicksPerSecond = 1e9;
```

For more information, see “Analyze Execution Time Data” on page 25-7.

To display measured execution times for a function, click the Simulation Data Inspector icon  on the corresponding row. You can use the Simulation Data Inspector to manage and compare plots from various executions. For information, see “Inspect Signal Data with Simulation Data Inspector”.

The following table lists the information provided in the code section profiles.

Column	Description
Section	Name of function from which code is generated.
Maximum Execution Time	Maximum time between start and end of function execution. Includes time spent in child functions.
Average Execution Time	Average time between start and end of execution. Includes time spent in child functions.
Maximum Self Time	Longest time spent in function. Excludes time spent in child functions.
Average Self Time	Average time spent in function. Excludes time spent in child functions.
Calls	Number of calls made to function.

Column	Description
	Icon that you click to display the profiled code section in the Command Window.
	Icon that you click to display measured execution times with Simulation Data Inspector.

Analyze Execution Time Data

After a software-in-the-loop (SIL) or processor-in-the-loop (PIL) execution, you can analyze execution time data using methods from the `coder.profile.ExecutionTime` and `coder.profile.ExecutionTimeSection` classes.

In the following example, you run a SIL execution and apply supplied methods to execution time data.

Extract Execution Time Data for Kalman Estimator Code

1 Run SIL execution to generate execution time data

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'), '.')
```

For a description of the Kalman estimator, see “C Code Generation at the Command Line”.

Create a `coder.EmbeddedCodeConfig` object.

```
config = coder.config('lib');
config.GenerateReport = true; % HTML report
```

Configure the object for SIL and enable execution time profiling.

```
config.VerificationMode = 'SIL';
config.CodeExecutionProfiling = true;
```

Generate library code for the `kalman01` MATLAB function and the SIL interface.

```
codegen('-config', config, '-args', {zeros(2,1)}, 'kalman01');
```

Run the MATLAB test file `test01_ui` with `kalman01_sil`. `kalman01_sil` is the SIL interface for `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.
Current plot released
```

Terminate the SIL execution process. Click the link [clear kalman01_sil](#).

```
### Stopping SIL execution for 'kalman01'
Execution profiling report: report(getCoderExecutionProfile('kalman01'))
```

2 Create workspace variable that holds execution time data

Use the `getCoderExecutionProfile` function to create a workspace variable that holds execution time profiling data.

```
executionProfile=getCoderExecutionProfile('kalman01');
```

3 Extract code sections

Use the `Sections` method.

```
allSections = executionProfile.Sections
```

The software displays the number of code sections and a list of properties.

```
allSections =
```

```
1x3 ExecutionTimeTaskSection array with properties:
```

```
Name
Number
ExecutionTimeInTicks
SelfTimeInTicks
TurnaroundTimeInTicks
TotalExecutionTimeInTicks
TotalSelfTimeInTicks
TotalTurnaroundTimeInTicks
MaximumExecutionTimeInTicks
MaximumExecutionTimeCallNum
MaximumSelfTimeInTicks
```



```

MaximumSelfTimeCallNum
MaximumTurnaroundTimeInTicks
MaximumTurnaroundTimeCallNum
NumCalls
ExecutionTimeInSeconds
Time

```

4 Extract execution time data from specific code section

Specify the code section that you want to examine.

```
secondSectionProfile = executionProfile.Sections(2)
```

The software displays profile data for the code section.

```
secondSectionProfile =
```

```
    ExecutionTimeTaskSection with properties:
```

```

        Name: 'kalman01'
        Number: 2
        ExecutionTimeInTicks: [1x300 uint64]
        SelfTimeInTicks: [1x300 uint64]
        TurnaroundTimeInTicks: [1x300 uint64]
        TotalExecutionTimeInTicks: 6641016
        TotalSelfTimeInTicks: 6641016
        TotalTurnaroundTimeInTicks: 6641016
        MaximumExecutionTimeInTicks: 48864
        MaximumExecutionTimeCallNum: 158
        MaximumSelfTimeInTicks: 48864
        MaximumSelfTimeCallNum: 158
        MaximumTurnaroundTimeInTicks: 48864
        MaximumTurnaroundTimeCallNum: 158
        NumCalls: 300
        ExecutionTimeInSeconds: [1x300 double]
        Time: [300x1 double]

```

You can extract specific properties, for example, the name of a profiled function.

```
nameOfSection = secondSectionProfile.Name
```

The software displays the name.

```
nameOfSection =
```

```
kalman01
```

The following table lists the information that you can extract from each code section.

Property	Description
Name	Name of entry-point function
Number	Code section number
ExecutionTimeInTicks	Vector of execution times, measured in timer ticks. Each element contains the difference between the timer reading at the start and at the end of the code section. The data type is the same data type as the data type of the timer used on the target, which allows you to infer the maximum range of the timer measurements.
SelfTimeInTicks	Vector of timer tick numbers. Each element contains the number of ticks recorded for the code section, excluding the time spent in calls to child functions.
TurnaroundTimeInTicks	Vector of timer tick numbers. Each element contains the number of ticks recorded between the start and the finish of the code section. Unless the code is preempted, this number is the same number as the execution time.
TotalExecutionTimeInTicks	Total number of timer ticks recorded for the code section over the entire execution.
TotalSelfTimeInTicks	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.
TotalTurnaroundTimeInTicks	Total number of timer ticks recorded between the start and the finish of the profiled code section over the entire execution. Unless the code is preempted, this number is the same as the total execution time.
MaximumExecutionTimeInTicks	Maximum number of timer ticks recorded for a single invocation of the code section over the execution.
MaximumExecutionTimeInTicksIndex	Number of call in which <code>MaximumExecutionTimeInTicks</code> occurs.
MaximumSelfTimeInTicks	Maximum number of timer ticks recorded for a single code section invocation, but excluding the time spent in calls to child functions.

Property	Description
<code>MaximumSelfTimeCalls</code>	Number of call in which <code>MaximumSelfTimeInTicks</code> occurs.
<code>MaximumTurnaroundTime</code>	Maximum number of timer ticks recorded between the start and the finish of a single invocation of the profiled code section over the execution. Unless the code is preempted, this time is the same as the maximum execution time.
<code>MaximumTurnaroundTimeInTicks</code>	Number of call in which <code>MaximumTurnaroundTimeInTicks</code> occurs.
<code>NumCalls</code>	Total number of calls to the code section over the entire execution.
<code>ExecutionTimeInSeconds</code>	Vector of execution times, measured in seconds. Each element contains the difference between the timer reading at the start and at the end of the code section. Produced only if <code>TimerTicksPerSecond</code> is set.
<code>Time</code>	Vector of execution time measurements for the code section.

Data Copy Reduction

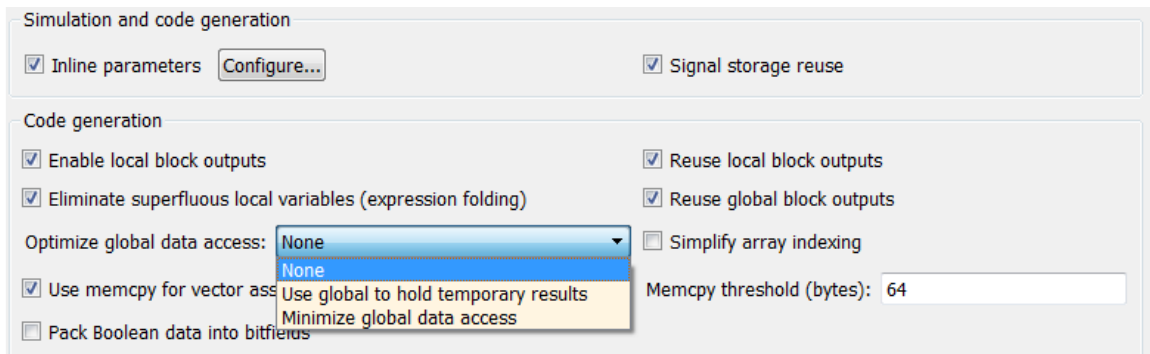
- “Optimize Global Variable Usage” on page 26-2
- “Reuse Block Outputs in the Generated Code” on page 26-12
- “Virtualized Output Ports Optimization” on page 26-16
- “Control Signal Storage” on page 26-18
- “Signal Reuse for Root-Level Model Inputs and Outputs” on page 26-19

Optimize Global Variable Usage

In this section...

“Minimize Global Data Access” on page 26-3

“Use Global to Hold Temporary Results” on page 26-8



To tune your application and choose tradeoffs for execution speed and memory usage, you can choose a global variable reference optimization for the generated code.

In the Configuration Parameters dialog box, select **Optimization > Signals and Parameters**. In the **Optimize global data access** drop-down list, three parameter options control global variable usage optimizations.

- **None**. Use default optimizations. This choice works well for most models. The code generator balances the use of local and global variables. It generates code which balances RAM and ROM consumption and execution speed.
- **Use global to hold temporary results**. Reusing global variables improves code efficiency and readability. This optimization reuses global variables, which results in fewer variables defined by the code generator. It reduces RAM and ROM consumption and data copies.
- **Minimize global data access**. Using local variables to cache global data reduces ROM consumption by reducing code size in certain cases, such as when the global variables are scalars. This optimization improves execution speed because the code uses fewer instructions for local variable references than for global variable references.

Minimize Global Data Access

This example shows how the code generator uses global and local variables when you select `None` versus when you select `Minimize global data access`.

- `None`

Use default optimizations. This choice works well for most models. The code generator balances the use of local and global variables. It produces balanced optimizations of RAM and ROM consumption and execution speed.

- `Minimize global data access`

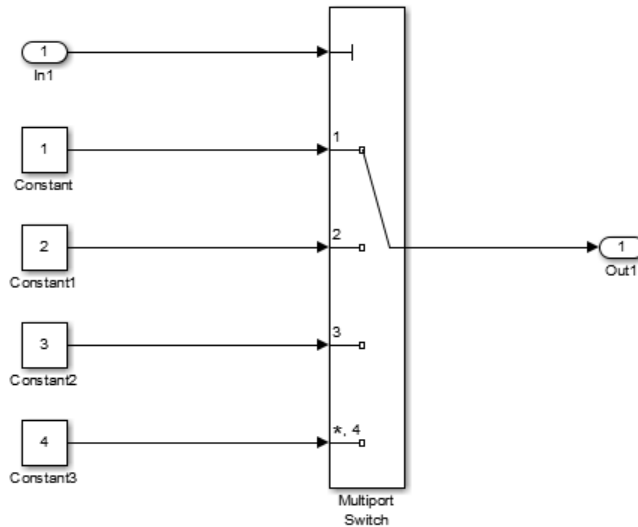
Minimize the use of global variables by using local variables to hold intermediate values. This optimization reduces ROM and RAM consumption by reducing code size in certain cases, such as when the global variables are scalars.

Minimizing the use of global variables by using local variables interacts with stack usage control. For example, stack size can determine the number of local and global variables that the code generator can allocate in the generated code. For more information, see “Customize Stack Space Allocation” and “Control Signal Storage” on page 26-18.

Example Model

In the Command Window, type `rtwdemo_optimize_global`.

Save the model to a local folder.



Set up the configuration for the model.

In the Configuration Parameters dialog box, on the **Optimization > Signals and Parameters** pane, verify that the **Signal Storage Reuse** check box is selected. From the **Optimize global data access** list, select **None**.

Generate Code

To generate the code, on the **Code Generation** pane:

- 1 Select **Generate code only**.
- 2 Click **Apply**.
- 3 Click **Generate Code**.

The code generator places the generated code in the `rtwdemo_optimize_global_ert_rtw` folder. For this example, the file `rtwdemo_optimize_global.c` contains the relevant code.

The code assigns values to the global vector `rtY.Out1` in each case statement without using a local variable. This assignment improves ROM and RAM consumption and

reduces data copies. The code places the value in the destination variable for each assignment instead of copying the value at the end.

```
switch ((int32_T)rtU.In1) {
  case 1:
    /* Output: '<Root>/Out1' incorporates:
     * Constant: '<Root>/Constant'
     */
    rtY.Out1 = 1.0;
    break;

  case 2:
    /* Output: '<Root>/Out1' incorporates:
     * Constant: '<Root>/Constant1'
     */
    rtY.Out1 = 2.0;
    break;

  case 3:
    /* Output: '<Root>/Out1' incorporates:
     * Constant: '<Root>/Constant2'
     */
    rtY.Out1 = 3.0;
    break;

  default:
    /* Output: '<Root>/Out1' incorporates:
     * Constant: '<Root>/Constant3'
     */
    rtY.Out1 = 4.0;
    break;
}

/* End of MultiPortSwitch: '<Root>/Multiport Switch' */
}
```

In the Static Code Metrics Report, examine the **Global Variables** section.

- 1 In the Code Generation report window, click **Static Code Metrics Report**.
- 2 Scroll down to the **Global Variables** section.
- 3 Click **[+]** before each variable to expand it.

Global variables defined in the generated code.

Global Variable	Size (bytes)	Reads / Writes	Reads / Writes in a Function
[-] rtU	8	1	1
In1	8	1	1
[-] rtY	8	4	4
Out1	8	4	4
[-] rtM	4	0*	0*
errorStatus	4	0	0
Total	20	5	

* The global variable is not directly used in any function.

The total number of reads and writes for global variables is 5.

Enable Optimization

On the **Optimization > Signals and Parameters** pane, from the **Optimize global data access** list, select **Minimize global data access**. Click **Apply**.

Generate Code with Optimization

On the **Code Generation** pane, click **Generate Code**.

In the `rtwdemo_optimize_global.c` listing, the code assigns the constant value to the local variable `tmp_Out1` in each case statement. The last statement in the code listing copies the value of `tmp_Out1` to the global variable `rtY.Out1`. Fewer global variable references result in fewer instructions and improved execution speed.

```
switch ((int32_T)rtU.In1) {
case 1:
    /* Outport: '<Root>/Out1' incorporates:
     * Constant: '<Root>/Constant'
     */
    tmp_Out1 = 1.0;
    break;
```

```

case 2:
  /* Outport: '<Root>/Out1' incorporates:
   * Constant: '<Root>/Constant1'
   */
  tmp_Out1 = 2.0;
  break;

case 3:
  /* Outport: '<Root>/Out1' incorporates:
   * Constant: '<Root>/Constant2'
   */
  tmp_Out1 = 3.0;
  break;

default:
  /* Outport: '<Root>/Out1' incorporates:
   * Constant: '<Root>/Constant3'
   */
  tmp_Out1 = 4.0;
  break;
}

/* End of MultiPortSwitch: '<Root>/Multiport Switch' */

/* Outport: '<Root>/Out1' */
rtY.Out1 = tmp_Out1;
}

```

In the Static Code Metrics Report, examine the **Global Variables** section.

Global variables defined in the generated code.

Global Variable	Size (bytes)	Reads / Writes	Reads / Writes in a Function
[-] rtU	8	1	1
In1	8	1	1
[-] rtY	8	1	1
Out1	8	1	1
[-] rtM	4	0 ⁺	0 ⁺
errorStatus	4	0	0
Total	20	2	

⁺ The global variable is not directly used in any function.

As a result of minimizing global data accesses, the total number of reads and writes for global variables has decreased from 5 to 2.

Use Global to Hold Temporary Results

This example shows how the code generator uses global and local variables when you select **None** versus when you select **Use global to hold temporary results**.

- **None**

Use default optimizations. This choice works well for most models. The code generator balances the use of local and global variables. It generates code which balances RAM and ROM consumption and execution speed.

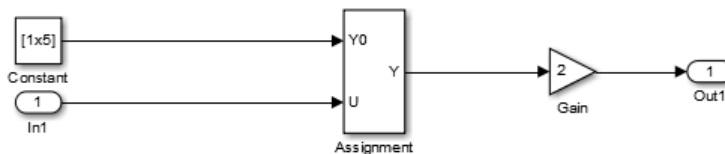
- **Use global to hold temporary results**

Maximize use of global variables. Reusing global variables improves code efficiency and readability. This optimization reuses global variables, which results in fewer variables defined by the code generator, reducing RAM and ROM consumption and data copies.

Example Model

In the Command Window, type `rtwdemo_optimize_global_ebf`.

Save the model to a local folder.



Set up the configuration for the model. In the Configuration Parameters dialog box, on the **Optimization > Signals and Parameters** pane, verify that the **Signal Storage Reuse** check box is selected. From the **Optimize global data access** list, select **None**.

Generate Code

To generate the code, on the **Code Generation** pane:

- 1 Select **Generate code only**.

- 2 Click **Apply**.
- 3 Click **Generate Code**.

The code generator places the generated code in the `rtwdemo_reuse_global_ebf_ert_rtw` folder. The file `rtwdemo_optimize_global_ebf.c` contains the relevant code.

The code assigns values to the local vector `rtb_Assignment`. The last statement copies the values in the local vector `rtb_Assignment` to the global vector `rtY.Out1`. Fewer global variable references result in improved execution speed. The code uses more instructions for global variable references than for local variable references.

```

/* Model step function */
void rtwdemo_optimize_global_ebf_step(void)
{
    real_T rtb_Assignment[5];
    int32_T i;

    /* Assignment: '<Root>/Assignment' incorporates:
     * Constant: '<Root>/Constant'
     * Inport: '<Root>/In1'
     */
    for (i = 0; i < 5; i++) {
        rtb_Assignment[i] = rtCP_Constant_Value[i];
    }

    rtb_Assignment[1] = rtU.In1;

    /* End of Assignment: '<Root/Assignment' */

    /* Outport: '<Root>/Out1' incorporates:
     * Gain: '<Root>/Gain'
     */
    for (i = 0; i < 5; i++) {
        rtY.Out1[i] = 2.0 * rtb_Assignment[i];
    }

    /* End of Outport: '<Root>/Out1' */
}

```

In the Static Code Metrics Report, examine the **Global Variables** section.

- 1 In the Code Generation report window, click **Static Code Metrics Report**.

- 2 Scroll down to the Global Variables section.
- 3 Click **[+]** before each variable to expand it.

Global variables defined in the generated code.

Global Variable	Size (bytes)	Reads / Writes	Reads / Writes in a Function
[-] rtY	40	1	1
Out1	40	1	1
[-] rtU	8	1	1
In1	8	1	1
[-] rtM	4	0*	0*
errorStatus	4	0	0
Total	52	2	

* The global variable is not directly used in any function.

The total number of reads and writes for global variables is 2.

Enable Optimization

On the **Optimization > Signals and Parameters** pane, from the **Optimize global data access** list, select **Use global** to hold temporary results. Click **Apply**.

Generate Code with Optimization

On the **Code Generation** pane, click **Generate Code**.

rtwdemo_optimize_global_ebf.c contains the following code.

The code assigns values to the global vector `rtY.Out1` without using a local variable. This assignment improves ROM and RAM consumption and reduces data copies. The code places the value in the destination variable for each assignment instead of copying the value at the end.

```
for (i = 0; i < 5; i++) {
    rtY.Out1[i] = rtCP_Constant_Value[i];
}

rtY.Out1[1] = rtU.In1;
```

```

/* End of Assignment: '<Root>/Assignment' */

/* Outport: '<Root>/Out1' incorporates:
 * Gain: '<Root>/Gain'
 */
for (i = 0; i < 5; i++) {
    rtY.Out1[i] *= 2.0;
}

```

In the Static Code Metrics Report, examine the **Global Variables** section.

Global variables defined in the generated code.

Global Variable	Size (bytes)	Reads / Writes	Reads / Writes in a Function
[-] rtY	40	4	4
Out1	40	4	4
[-] rtU	8	1	1
In1	8	1	1
[-] rtM	4	0*	0*
errorStatus	4	0	0
Total	52	5	

* The global variable is not directly used in any function.

As a result of using global variables to hold local results, the total number of reads and writes for global variables has increased from 2 to 5. This optimization reduces data copies by reusing global variables.

Reuse Block Outputs in the Generated Code

In the Configuration Parameters dialog box, on the **Optimization > Signals and Parameters** pane, the **Reuse global block outputs** parameter controls the reuse of global block outputs in the generated code.

Reuse Global Block Outputs

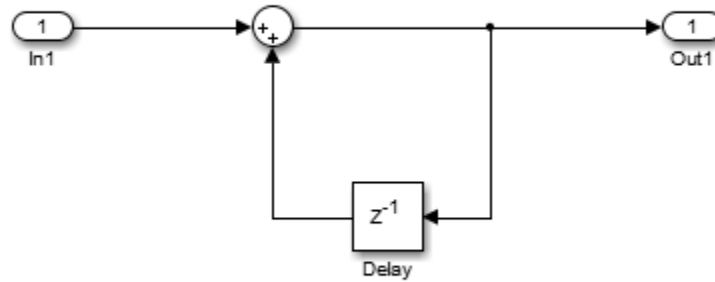


This example shows the results of enabling the **Reuse global block outputs** parameter.

Example Model

In the Command Window, type `rtwdemo_reuse_global`.

Save the model to a local folder.



- 1 In the Configuration Parameters dialog box, on the **Optimization > Signals and Parameters** pane, verify that **Signal Storage Reuse** is selected.
- 2 Clear **Reuse global block outputs**.
- 3 On the **Code Generation > Report** pane select **Static code metrics**.

Generate Code

To generate the code, on the **Code Generation** pane:

- Select **Generate code only**.
- Click **Apply**.
- Click **Generate Code**.

The code generator places the generated code in the `rtwdemo_reuse_global_ert_rtw` folder. The file `rtwdemo_reuse_global.c` contains the relevant code listed. The code assigns the value of:

- The calculation to the global variable `rtY.Out1`.
- The global variable `rtY.Out1` to the global variable `rtDW.Delay_DSTATE`.

```
rtY.Out1 = rtU.In1 + rtDW.Delay_DSTATE;
```

```
/* Update for Delay: '<Root>/Delay' */
rtDW.Delay_DSTATE = rtY.Out1;
```

Global variables defined in the generated code.

Global Variable	Size (bytes)	Reads / Writes	Reads / Writes in a Function
[~] rtDW	8	3	2
Delay_DSTATE	8	3	2
[~] rtU	8	2	1
In1	8	2	1
[~] rtY	8	3	2
Out1	8	3	2
[~] rtM_	4	0*	0*
errorStatus	4	0	0
Total	28	8	

* The global variable is not directly used in any function.

The total number of reads and writes for global variables is 8. The total size in bytes is 28.

Enable Optimization

On the **Optimization > Signals and Parameters** pane, select **Reuse global block outputs**. Click **Apply**.

Generate Code with Optimization

On the **Code Generation** pane, click **Generate Code**.

The code generator reduces two statements to one statement and three global variables to two global variables. This optimization reduces ROM and RAM consumption and improves execution speed.

```
rtY.Out1 += rtU.In1;
```

Global variables defined in the generated code.

Global Variable	Size (bytes)	Reads / Writes	Reads / Writes in a Function
[-] rtU	8	2	1
In1	8	2	1
[-] rtY	8	3	2
Out1	8	3	2
[-] rtM	4	0+	0+
errorStatus	4	0	0
Total	20	5	

* The global variable is not directly used in any function.

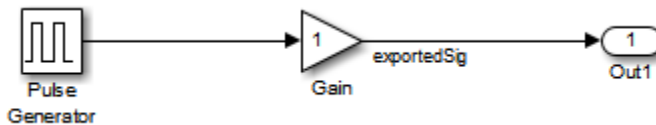
The optimization reduces the total number of reads and writes for global variables from 8 to 5 and the total size in bytes from 28 to 20.

For more information, see “Optimize Global Variable Usage” on page 26-2.

Virtualized Output Ports Optimization

The *virtualized output ports* optimization lets you store the signal entering the root output port as a global variable. Clearing the **MAT-file logging** option and setting the TLC variable `FullRootOutputVector` to 0, both defaults for Embedded Coder, eliminate code and data storage associated with root output ports.

Consider the model in the following block diagram. The signal `exportedSig` has `exportedGlobal` storage class.



In the default case, the output of the Gain block is written to the signal storage location, `exportedSig`. The code generator does not generate code or data for the `Out1` block, which has become a virtual block.

```
/* Gain Block: <Root>/Gain */
exportedSig = rtb_PulseGen * VirtOutPortLogOFF_P.Gain_Gain;
```

In cases where you enable **MAT-file logging** or set `FullRootOutputVector = 1`, the generated code represents root output ports as members of an external outputs vector.

The following code fragment was generated with **MAT-file logging** enabled. The output port is represented as a member of the external outputs vector `VirtOutPortLogON_Y`. The Gain block output value is copied to `exportedSig` and to the external outputs vector.

```
/* Gain Block: <Root>/Gain */
exportedSig = rtb_PulseGen * VirtOutPortLogON_P.Gain_Gain;

/* Output Block: <Root>/Out1 */
VirtOutPortLogON_Y.Out1 = exportedSig;
```

Data maintenance in the external outputs vector can be significant for smaller models that perform benchmarks.

You can force root output ports to be stored in the external outputs vector (regardless of the setting of **MAT-file logging**) by setting the TLC variable `FullRootOutputVector` to 1. Add the statement

```
%assign FullRootOutputVector = 1
```

to the Embedded Coder system target file. Alternatively, you can enter the assignment from the MATLAB command line using the `set_param` command, the model parameter `TLCOptions`, and the TLC option `-a`. For more information, see “Specify TLC Options” and “Configure TLC”.

For more information on how to control signal storage in generated code, see “Signals”.

Control Signal Storage

You can control how signals in your model are stored and represented in the generated code with a number of options. You can also control where signal storage is declared.

You can choose to store signals in global memory space or locally in functions, in stack variables. For more information, see “Signals”.

If you want to store signals in stack space, you must enable the **Enable local block outputs** option.

- 1 In the Configuration Parameters dialog box, select **Optimization > Signals and Parameters**. Select **Signal storage reuse**.
- 2 Select the **Enable local block outputs** option. Click **Apply**.

Signal Reuse for Root-Level Model Inputs and Outputs

Signal reuse allows for further optimizations that can reduce data copies, global variables, and ROM/RAM consumption. The code generator can reuse the root input and output signals for a model in the generated code. When you prepare to generate code for a model, you can specify that pairs of root-level model input and output signals are available to be reused. Use the **Reusable** custom storage class.

- 1 Choose one root-level input signal and one root-level output signal. Assign them the same name. The name is used for the reused variable name in the generated code. Specify that the signal name must resolve to a **Simulink.Signal** object.
- 2 Create a **Simulink.Signal** object and specify the same name as you chose for the signals in the previous step. Set the storage class to **Custom**. Set the custom storage class to **Reusable** or to a custom storage class that you derived from **Reusable**. Verify that **Is Reusable** is set to **Yes**.

In some cases, code generation creates an extra buffer to satisfy requested buffer use for the specified signal.

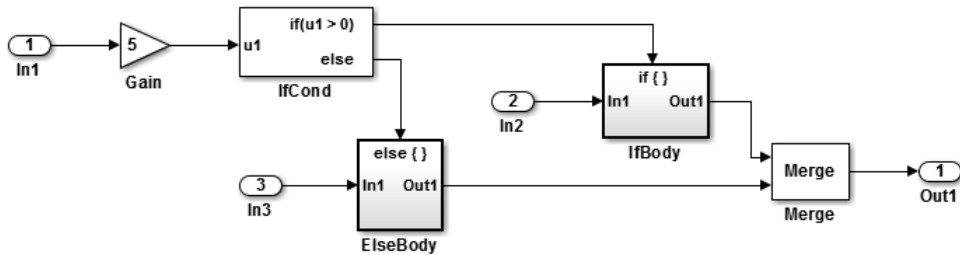
When you run an executable produced by code generation, and you reuse a root I/O pair, it is important that when the root input value is zero that the root output value is also zero. If the output value is nonzero and you reuse the signals, then the results from the simulation can differ from the results produced by the executable.

Reuse Root-Level I/O Signals

This example shows how the code generator can reuse root-level input and output signals. Reuse reduces ROM and RAM consumption, data copies, and global variables.

In the Command Window, open `rtwdemo_merge`.

Save the model to a local folder.



Generate Code Without Optimization

- 1 In the Configuration Parameters dialog box, on the **Code Generation** pane, set **System target file** to `ert.tlc`.
- 2 Select **Generate code only**.
- 3 Click **Apply**.
- 4 Click **Generate Code**.

In the `rtwdemo_merge.h` header file the code generator declares these variables: `In1`, `In2`, `In3`, and `Out1`, for a total of four variables.

```

/* External inputs (root inport signals with auto storage) */
typedef struct {
    real32_T In1;           /* '<Root>/In1' */
    real32_T In2;           /* '<Root>/In2' */
    real32_T In3;           /* '<Root>/In3' */
} ExtU_rtwdemo_merge_T;

/* External outputs (root outports fed by signals with auto storage) */
typedef struct {
    real32_T Out1;         /* '<Root>/Out1' */
} ExtY_rtwdemo_merge_T;

```

In `rtwdemo_merge.c`, in the `rtwdemo_merge_U` structure, the code generator uses `In1`, `In2`, and `In3` for the input values. In the `rtwdemo_merge_Y` structure, it uses `Out1` for the output value.

```

if (5.0F * rtwdemo_merge_U.In1 > 0.0F) {
    /* Outputs for IfAction SubSystem: '<Root>/IfBody' incorporates:
     * ActionPort: '<S2>/IfAction'
     */

```



```

/* Output: '<Root>/Out1' incorporates:
 * Gain: '<S2>/ifgain'
 * Inport: '<Root>/In2'
 */
rtwdemo_merge_Y.Out1 = 5.0F * rtwdemo_merge_U.In2;

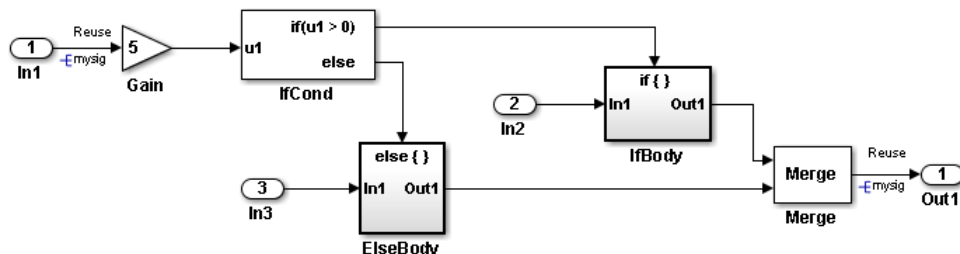
/* End of Outputs for SubSystem: '<Root>/IfBody' */
} else {
/* Outputs for IfAction SubSystem: '<Root>/ElseBody' incorporates:
 * ActionPort: '<S1>/IfAction'
 */
/* Output: '<Root>/Out1' incorporates:
 * Gain: '<S1>/elsegain'
 * Inport: '<Root>/In3'
 */
rtwdemo_merge_Y.Out1 = 10.0F * rtwdemo_merge_U.In3;

```

Enable Optimization

- 1 Choose a pair of root-level input and outputs. For this example, choose In1 for the root-level input and Out1 for the root-level output.
- 2 For the input signal:
 - a Right-click the signal line. From the context menu, choose Properties.
 - b Set **Signal name** to mysig. Press **Tab**.
 - c Select **Signal name must resolve to Simulink signal object**.
 - d Click **OK**.
- 3 Repeat step 2 for the corresponding output signal.

The model, after being configured, with the root I/O signals labelled with Reuse, looks like this:



- 4 Create a reusable signal object. Call it `mysig`, to match the name of the input and output signals. In the Command Window, enter:

```
mysig=Simulink.Signal;
mysig.CoderInfo.StorageClass = 'Custom';
mysig.CoderInfo.CustomStorageClass = 'Reusable';
```

Generate Code With Optimization

On the **Code Generation** pane, click **Generate Code**.

In the `rtwdemo_merge.h` header file, the code generator declares `In2`, `In3`, and `mysig`, resulting in three variables.

```
/* External inputs (root inport signals with auto storage) */
typedef struct {
    real32_T In2;                /* '<Root>/In2' */
    real32_T In3;                /* '<Root>/In3' */
} ExtU_rtwdemo_merge_T;
...

/* Declaration for custom storage class: Reusable */
extern real32_T mysig;
```

The code in `rtwdemo_merge.c`, in the structure `rtwdemo_merge_U`, uses the variables `In2` and `In3` for input values. The code reuses the variable `mysig` for input and output, instead of using `In1` and `Out1`.

```
if (5.0F * mysig > 0.0F) {
    /* Outputs for IfAction SubSystem: '<Root>/IfBody' incorporates:
     * ActionPort: '<S2>/IfAction'
     */
    mysig = 5.0F * rtwdemo_merge_U.In2;

    /* End of Outputs for SubSystem: '<Root>/IfBody' */
} else {
    /* Outputs for IfAction SubSystem: '<Root>/ElseBody' incorporates:
     * ActionPort: '<S1>/IfAction'
     */
    mysig = 10.0F * rtwdemo_merge_U.In3;
```

With reuse, the number of variables created and referenced is reduced by half. Each input/output signal pair results in one variable rather than two.

More About

- “Introduction to Custom Storage Classes”
- “Simulink Package Custom Storage Classes”
- “Signal Objects”

Execution Speed

- “Remove Initialization Code” on page 27-2
- “Generate Pure Integer Code If Possible” on page 27-3
- “Disable MAT-File Logging” on page 27-4
- “Simplify Multiply Operations In Array Indexing” on page 27-5

Remove Initialization Code

Consider selecting the **Remove internal state zero initialization** and **Remove root level I/O zero initialization** options on the **Optimization > General** pane.

These options (both off by default) control whether internal data (block states and block outputs) and external data (root inports and outports whose value is zero) are initialized. Initializing the internal and external data whose value is zero is a precaution and your application might not require it. Many embedded application environments initialize RAM to zero at startup, making generation of initialization code redundant.

However, be aware that if you select **Remove internal state zero initialization**, memory might not be in a known state each time the generated code begins execution. If you turn the option on, running a model (or a generated S-function) multiple times can result in different answers for each run.

This behavior is sometimes desirable. For example, you can turn on **Remove internal state zero initialization** if you want to test the behavior of your design during a warm boot (that is, a restart without full system reinitialization).

In cases where you have turned on **Remove internal state zero initialization** but still want to get the same answer on every run from a S-function generated by the Embedded Coder software, you can use either of the following MATLAB commands before each run:

```
clear SFcnName
```

where *SFcnName* is the name of the S-function, or

```
clear mex
```

A related option, **Use memset to initialize floats and doubles**, lets you control the representation of zero used during initialization. See “Use memset to initialize floats and doubles to 0.0” in the Simulink reference documentation.

Note that the code still initializes data structures whose value is not zero when **Remove internal state zero initialization** and **Remove root level I/O zero initialization** are selected.

Note also that data of `ImportedExtern` or `ImportedExternPointer` storage classes are not initialized, regardless of the settings of these options.

Generate Pure Integer Code If Possible

If your application uses only integer arithmetic, clear the **Support floating-point numbers** option in the **Software environment** section of the **Interface** pane so that the generated code contains no floating-point data or operations. When this option is cleared, an error is raised if noninteger data or expressions are encountered during code generation. The error message reports the offending blocks and parameters.

Disable MAT-File Logging

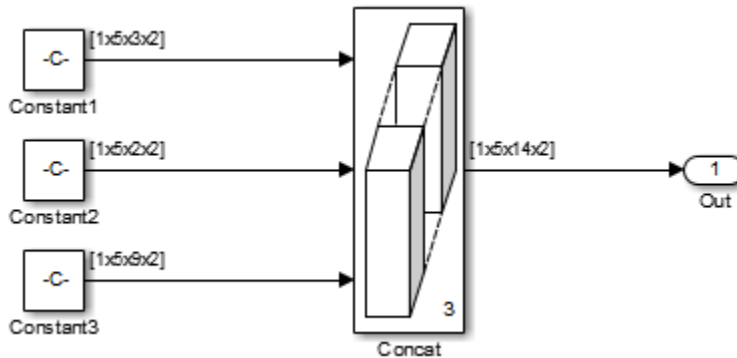
Clear the **MAT-file logging** option in the **Verification** section of the **Interface** pane. This setting is the default, and is recommended for embedded applications because it eliminates the extra code and memory usage required to initialize, update, and clean up logging variables. In addition to these efficiencies, clearing the **MAT-file logging** option lets you exploit further efficiencies under certain conditions. See “Virtualized Output Ports Optimization” on page 26-16 for information.

Note also that code generated to support MAT-file logging invokes `malloc`, which may be undesirable for your application.

Simplify Multiply Operations In Array Indexing

The generated code might have multiply operations when indexing an element of an array. You can select the optimization parameter “Simplify array indexing” to replace multiply operations in the array index with a temporary variable. To modify this parameter open the Configuration Parameters dialog box and select the **Optimization > Signals and Parameters** pane. This optimization can improve execution speed by reducing the number of times the multiply operation is executed.

If you have the following model:



The Constant blocks have the following **Constant value**:

- Const1: `reshape(1:30,[1 5 3 2])`
- Const2: `reshape(1:20,[1 5 2 2])`
- Const3: `reshape(1:90,[1 5 9 2])`

The Concatenate block parameter **Mode** is set to `Multidimensional array`.

Generated Code Results

Building the model with the **Simplify array indexing** parameter turned off generates the following code:

```
int32_T i;
int32_T i_0;
int32_T i_1;
```

```

for (i = 0; i < 2; i++) {
    for (i_1 = 0; i_1 < 3; i_1++) {
        for (i_0 = 0; i_0 < 5; i_0++) {
            ex_arrayindex_Y.Out[(i_0 + 5 * i_1) + 70 * i] =
                ex_arrayindex_P.Constant1_Value[(5 * i_1 + i_0) + 15 * i];
        }
    }
}

for (i = 0; i < 2; i++) {
    for (i_1 = 0; i_1 < 2; i_1++) {
        for (i_0 = 0; i_0 < 5; i_0++) {
            ex_arrayindex_Y.Out[(i_0 + 5 * (i_1 + 3)) + 70 * i] =
                ex_arrayindex_P.Constant2_Value[(5 * i_1 + i_0) + 10 * i];
        }
    }
}

for (i = 0; i < 2; i++) {
    for (i_1 = 0; i_1 < 9; i_1++) {
        for (i_0 = 0; i_0 < 5; i_0++) {
            ex_arrayindex_Y.Out[(i_0 + 5 * (i_1 + 5)) + 70 * i] =
                ex_arrayindex_P.Constant3_Value[(5 * i_1 + i_0) + 45 * i];
        }
    }
}

```

After selecting the **Simplify array indexing** parameter and building the model again, a multiply operation in the array index, $[(i_0 + 5 * i_1) + 70 * i]$, is replaced with $[(i_0 + tmp_1) + tmp]$. The generated code is now:

```

int32_T i;
int32_T i_0;
int32_T i_1;
int32_T tmp;
int32_T tmp_0;
int32_T tmp_1;

tmp = 0;
tmp_0 = 0;
for (i = 0; i < 2; i++) {
    tmp_1 = 0;
    for (i_1 = 0; i_1 < 3; i_1++) {

```

```
    for (i_0 = 0; i_0 < 5; i_0++) {
        ex_arrayindex_Y.Out[(i_0 + tmp_1) + tmp] =
            ex_arrayindex_P.Constant1_Value[(i_0 + tmp_1) + tmp_0];
    }

    tmp_1 += 5;
}

tmp += 70;
tmp_0 += 15;
}

tmp = 0;
tmp_0 = 0;
for (i = 0; i < 2; i++) {
    tmp_1 = 0;
    for (i_1 = 0; i_1 < 2; i_1++) {
        for (i_0 = 0; i_0 < 5; i_0++) {
            ex_arrayindex_Y.Out[((i_0 + tmp_1) + tmp) + 15] =
                ex_arrayindex_P.Constant2_Value[(i_0 + tmp_1) + tmp_0];
        }

        tmp_1 += 5;
    }

    tmp += 70;
    tmp_0 += 10;
}

tmp = 0;
tmp_0 = 0;
for (i = 0; i < 2; i++) {
    tmp_1 = 0;
    for (i_1 = 0; i_1 < 9; i_1++) {
        for (i_0 = 0; i_0 < 5; i_0++) {
            ex_arrayindex_Y.Out[((i_0 + tmp_1) + tmp) + 25] =
                ex_arrayindex_P.Constant3_Value[(i_0 + tmp_1) + tmp_0];
        }

        tmp_1 += 5;
    }

    tmp += 70;
    tmp_0 += 45;
}
```

}

Memory Usage

- “Optimize Generated Code Using Specified Minimum and Maximum Values” on page 28-2
- “Reduce Global Variables in Nonreusable Subsystem Functions” on page 28-8

Optimize Generated Code Using Specified Minimum and Maximum Values

To optimize the generated code for your model, you can choose an option to use input range information, also known as *design minimum and maximum*, that you specify on signals and parameters. These minimum and maximum values usually represent environmental limits, such as temperature, or mechanical and electrical limits, such as output ranges of sensors.

When you select the **Optimize using specified minimum and maximum values** configuration parameter, the software uses the minimum and maximum values to derive range information for downstream signals in the model. It then uses this derived range information to determine if it is possible to streamline the generated code. For example, by:

- Reducing expressions to constants
- Removing dead branches of conditional statements
- Eliminating unnecessary mathematical operations

This optimization results in:

- Reduced ROM and RAM consumption
- Improved execution speed

How to Configure Your Model

To make optimization more likely:

- Provide as much design minimum and maximum information as possible. Specify minimum and maximum values for signals and parameters in the model for:
 - Inport and Outport blocks
 - Block outputs
 - Block inputs, for example, for the MATLAB Function and Stateflow Chart blocks
 - `Simulink.Signal` objects
- Before generating code, test the minimum and maximum values for signals and parameters. Otherwise, optimization might result in numerical mismatch with

simulation. You can simulate your model with simulation range checking enabled. If errors or warnings occur, fix these issues before generating code.

How to Enable Simulation Range Checking

- 1 In your model, select **Simulation > Model Configuration Parameters** to open the Configuration Parameters dialog box.
 - 2 In the Configuration Parameters dialog box, select **Diagnostics > Data Validity**.
 - 3 On the **Data Validity** pane, under **Signals**, set **Simulation range checking** to **warning** or **error**.
- Provide design minimum and maximum information upstream of blocks as close to the inputs of the blocks as possible. If you specify minimum and maximum values for a block output, these values are most likely to affect the outputs of the blocks immediately downstream. For more information, see “Optimize Generated Code Using Specified Minimum and Maximum Values” on page 28-3.

How to Enable Optimization

- 1 Set the **Code Generation > System target file** configuration parameter to select an Embedded Real-Time (ERT) target (requires a Embedded Coder license).
- 2 Specify design minimum and maximum values for signals and parameters in your model using the tips in “How to Configure Your Model” on page 28-2.
- 3 Select the **Optimization > General Optimize using specified minimum and maximum values** configuration parameter.

For more information, see “Optimize using the specified minimum and maximum values” in the Simulink documentation.

Optimize Generated Code Using Specified Minimum and Maximum Values

This example demonstrates how the software uses specified input range information to determine whether it can eliminate unnecessary utility functions from the generated code. It uses the `rtwdemo_minmax` model.

Generate Code Without Using Specified Minimum and Maximum Values

First, generate code without taking into account the minimum and maximum values for the inputs to the Sum and Gain blocks or the minimum and maximum values for the Gain block parameter to see the code generated without the optimization.

- 1 Open the model. At the MATLAB command line, enter:

```
rtwdemo_minmax
```

- 2 Double-click the **View Optimization Configuration** button.

The **Optimization** pane of the Configuration Parameters dialog box appears.

On the **Code generation** panel, note that the **Optimize using specified minimum and maximum values** parameter is cleared.

- 3 Double-click the **Generate Code** button.

The code generation report appears.

- 4 In the left pane of the report, click the `rtwdemo_minmax.c` link.

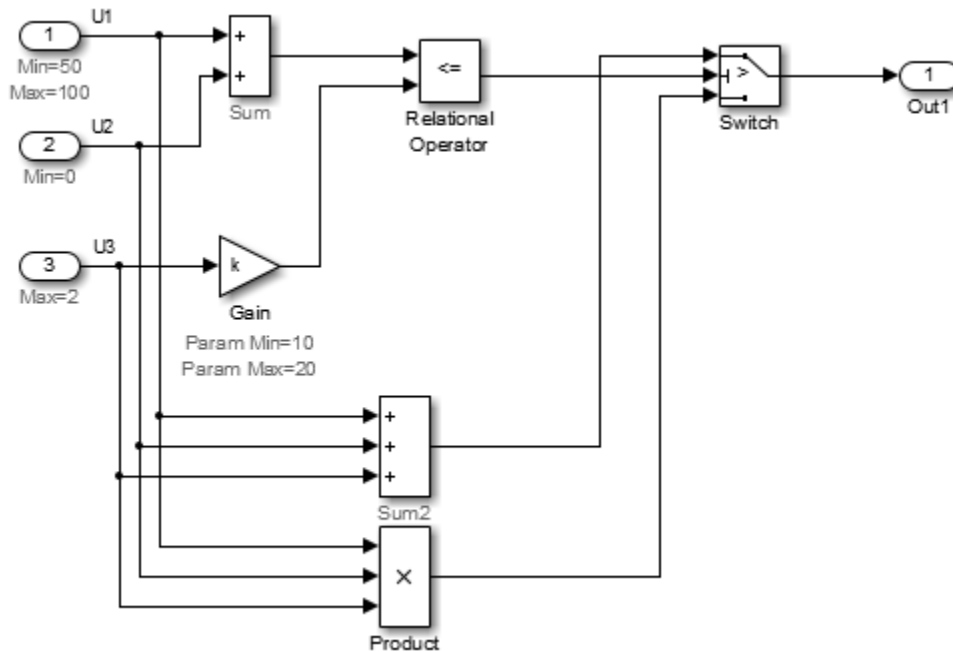
The report displays the C code in the right pane.

The generated code for this model includes a branch for each of the Relational Operator block inputs.

```
void rtwdemo_minmax_step(void)
{
    if (U1 + U2 <= k * U3) {
        rtY.Out1 = (U1 + U2) + U3;
    } else {
        rtY.Out1 = U1 * U2 * U3;
    }
}
```

Generate Code Using Minimum and Maximum Values

Next, enable the optimization and generate code for the model again, this time taking into account the design minimum and maximum values for the inputs to the Sum and Gain blocks and the minimum and maximum values for the Gain block parameter.



Note that:

- The minimum value of the first input to the Relational Operator block is 50 because this value is the minimum output from the Sum block.
- The maximum value of the second input to the Relational Operator block is 40 because this value is the maximum output of the Gain block.

Therefore, the output of the Relational Operator block is false, and the output of the Switch block is always the product of the three inputs.

- 1 Double-click the **View Optimization Configuration** button.

The Optimization pane of the Configuration Parameters dialog box appears.

- 2 On the **Code generation** panel, select the **Optimize using specified minimum and maximum values** parameter and click **Apply**.
- 3 Double-click the **Generate Code** button.

The code generation report appears.

- 4 In the left pane of the report, click the `rtwdemo_minmax.c` link and inspect the generated code. Using the minimum and maximum values, the software optimized the generated code by eliminating the conditional statement.

```
void rtwdemo_minmax_step(void)
{
    rtY.Out1 = U1 * U2 * U3;
}
```

Limitations

- This optimization does not take into account minimum and maximum values for:
 - Merge block inputs. To work around this issue, use a `Simulink.Signal` object on the Merge block output and specify the range on this object.
 - Bus elements.
 - Conditionally-executed subsystem (such as a triggered subsystem) block outputs that are directly connected to an Output block.

Output blocks in conditionally-executed subsystems can have an initial value specified for use only when the system is not triggered. In this case, the optimization cannot use the range of the block output because the range might not cover the initial value of the block.

- If you use Polyspace software to verify code generated using this optimization, it might mark code that was previously green as orange. For example, if your model contains a division where the range of the denominator does not include zero, the generated code does not include protection against division by zero. Polyspace might mark this code orange because it does not have information about the minimum and maximum values for the inputs to the division.

Polyspace Code Prover automatically captures some minimum and maximum values specified in the MATLAB workspace, for example, for `Simulink.Signal` and `Simulink.Parameter` objects. In this example, to provide range information to the Polyspace software, use a `Simulink.Signal` object on the input of the division and specify a range that does not include zero.

Polyspace Code Prover stores these values in a Data Range Specification (DRS) file. However, they do not capture all minimum and maximum values in your Simulink model. To provide additional minimum and maximum information to Polyspace, you can manually define a DRS file.

- If you are using double-precision data types and the **Code Generation > Interface > Support non-finite numbers** configuration parameter is selected, this optimization does not occur.
- If your model contains multiple instances of a reusable subsystem and each instance uses input signals with different minimum and maximum values, this optimization might result in different generated code for each subsystem so code reuse does not occur. Without this optimization, code is generated once for the subsystem and shares this code among the multiple instances of the subsystem.
- The Model Advisor **Check safety-related optimization settings** check generates a warning if this option is selected. For many safety-critical applications, removing dead code automatically is unacceptable because doing so might make code untraceable. For more information, see “Check safety-related optimization settings”.

Reduce Global Variables in Nonreusable Subsystem Functions

In this section...

“Generate void-void Function” on page 28-8

“Generate Function with Arguments” on page 28-9

Global variables can increase memory requirements and reduce execution speed. To reduce global RAM for a nonreusable subsystem, you can generate a function interface that passes data through arguments instead of global variables. The Subsystem block parameter “Function interface” provides this option. To compare the outputs for the **Function interface** options, first generate a function for a subsystem with a **void-void** interface, and then generate a function with arguments.

Generate void-void Function

By default, when you configure a Subsystem block as a nonreusable function, it generates a **void-void** interface.

- 1 Open the example model `rtwdemo_roll`.
- 2 Right-click the subsystem `RollAngleReference`. From the list select **Block Parameter (Subsystem)**.
- 3 In the Block Parameter dialog box, confirm that the **Treat as atomic unit** check box is selected.
- 4 Click the **Code Generation** tab and set the **Function packaging** parameter to **Nonreusable function**.
- 5 The **Function interface** parameter is already set to **void-void**.
- 6 Click **Apply** and **OK**.
- 7 Repeat steps 2–6, for the other subsystems `HeadingMode` and `BasicRollMode`.
- 8 Generate code and the static code metrics report for `rtwdemo_roll`. This model is configured to generate a code generation report and to open the report automatically. For more information, see “Generate Static Code Metrics Report for Simulink Model”.

In the code generation report, in `rtwdemo_roll.c`, the generated code for subsystem `RollAngleReference` contains a **void-void** function definition:

```
void rtwdemo_roll_RollAngleReference(void)
```

```
{
  ...
}
```

In the static code metrics report, navigate to **Global Variables**. With the void-void option, the number of bytes for global variables is 59.

2. Global Variables [\[hide\]](#)

Global variables defined in the generated code.

Global Variable	Size (bytes)	Reads / Writes	Reads / Writes in a Function
[+] rtwdemo_roll_U	26	14	6
[+] rtwdemo_roll_B	16	16	8
[+] rtwdemo_roll_DW	9	18	15
[+] rtwdemo_roll_M	4	0*	0*
[+] rtwdemo_roll_Y	4	2	2
Total	59	50	

* The global variable is not directly used in any function.

Next, generate the same function with the `Allow arguments` option to compare the results.

Generate Function with Arguments

To reduce global RAM, improve ROM usage and execution speed, generate a function that allows arguments:

- 1 Open the Subsystem Block Parameter dialog box for `RollAngleReference`.
- 2 Click the **Code Generation** tab. Set the **Function interface** parameter to `Allow arguments`.
- 3 Click **Apply** and **OK**.
- 4 Repeat steps 2 and 3, for the other subsystems `HeadingMode` and `BasicRollMode`.
- 5 Generate code and the static code metrics report for `rtwdemo_roll`.

In the code generation report, in `rtwdemo_roll.c`, the generated code for subsystem `RollAngleReference` now has arguments:

```

real32_T rtwdemo_roll_RollAngleReference(real32_T rtu_Turn_Knob,...
                                         boolean_T rtu_AP_Eng,...
                                         real32_T rtu_Phi)
{
  ...
}

```

In the static code metrics report, navigate to **Global Variables**. With the **Allow arguments** option set, the total number of bytes for global variables is now 47 bytes.

2. Global Variables [\[hide\]](#)

Global variables defined in the generated code.

Global Variable	Size (bytes)	Reads / Writes	Reads / Writes in a Function
[+] rtwdemo_roll_U	26	11	8
[+] rtwdemo_roll_DW	9	18	15
[+] rtwdemo_roll_B	4	2	1
[+] rtwdemo_roll_M	4	0*	0*
[+] rtwdemo_roll_Y	4	2	2
Total	47	33	

* The global variable is not directly used in any function.

Verification

Code Tracing

- “About Code Tracing” on page 29-2
- “Format of Traceability Tags” on page 29-3
- “Trace Code to Model Objects Using Hyperlinks” on page 29-4
- “Trace Model Objects to Generated Code” on page 29-6
- “Reload Existing Traceability Information” on page 29-9
- “Customize Traceability Reports” on page 29-10
- “Generate a Traceability Matrix” on page 29-12
- “Traceability Limitations” on page 29-13

About Code Tracing

The Simulink Coder code generator produces the HTML code generation report for a model. The report includes the following resources that support code traceability:

- Code element hyperlinks (indicated with underlining) that you can use to trace through and toggle between generated source and header files.
- Tags in code comments that identify blocks from which lines of code are generated.

For more information, see

- “Format of Traceability Tags” on page 29-3
- Tracing code to blocks using “`hilite_system`”
- “Generate a Code Generation Report”

The Embedded Coder product extends traceability capabilities to support:

- “Trace Code to Model Objects Using Hyperlinks” on page 29-4
- “Trace Model Objects to Generated Code” on page 29-6
- “Reload Existing Traceability Information” on page 29-9
- “Customize Traceability Reports” on page 29-10

Format of Traceability Tags

A traceability tag appears in a comment above the corresponding line of generated code. The format of the tags is `<system>/block_name`, where

- *system* is one of the following:
 - The string `Root`
 - A unique system number assigned by the Simulink engine
- *block_name* is the name of the source block

The code generator documents the tags for a model in the comments section of the generated header file `model.h`. For example, the following comment appears in the header file for model, `foo`, that has a subsystem `Outer` and a nested subsystem `Inner`:

```
/* Here is the system hierarchy for this model.
 *
 * <Root> : foo
 * <S1>   : foo/Outer
 * <S2>   : foo/Outer/Inner
 */
```

Examples of Tagged Code

The following code shows a tag comment adjacent to a line of code. This code is generated from a Gain block at the root level of a source model:

```
/* Gain: '<Root>/UnDeadGain1' */
rtb_UnDeadGain1_h = dead_gain_U.In1 *
    dead_gain_P.UnDeadGain1_Gain;
```

The following code shows a tag comment adjacent to a line of code. This code is generated from a Gain block within a subsystem one level below the root level of the source model:

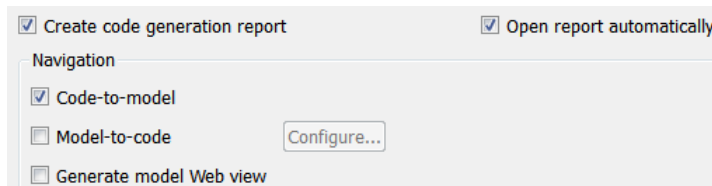
```
/* Gain: '<S1>/Gain' */
dead_gain_B.temp0 *= (dead_gain_P.s1_Gain_Gain);
```

Trace Code to Model Objects Using Hyperlinks

When using the Simulink Coder product, you can trace code to model objects using the `hilite_system` command. The Embedded Coder product simplifies traceability with the use of hyperlinks in HTML code generation reports. The reports display hyperlinks in comment lines in generated code. You can highlight the corresponding block or subsystem in the model diagram by clicking the hyperlinks.

To use hyperlinks for tracing code to model objects:

- 1 Open the model and make sure it is configured for an ERT target.
- 2 In the Configuration Parameters dialog box, on the **Code Generation > Report** pane, select **Create code generation report**. The parameter is selected by default. When selected, this parameter enables and selects **Open report automatically** and **Code-to-model**.



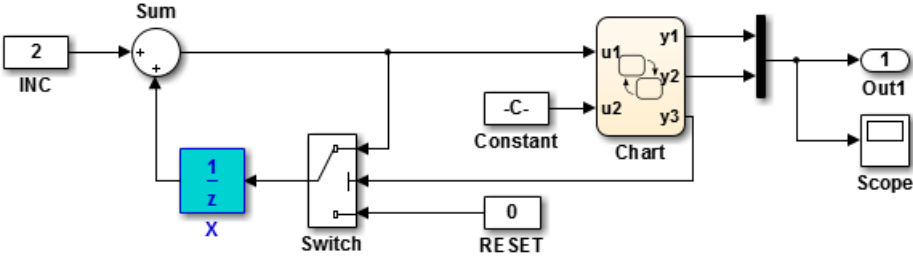
- 3 Build or generate code for the model. An HTML code generation report is displayed.
- 4 In the HTML report window, click hyperlinks to highlight source blocks. For example, generate an HTML report for model `rtwdemo_hyperlinks`. In the generated code for the model step function in `rtwdemo_hyperlinks.c`, click the first `UnitDelay` block hyperlink.

```

130 /* Model step function */
131 void rtwdemo_hyperlinks_step(void)
132 {
133     static real_T tmp[5] = { 10.0, -20.0, 30.0, -40.0, 50.0 };
134
135     /* Sum: '<Root>/Sum' incorporates:
136      * Constant: '<Root>/INC'
137      * Switch: '<Root>/Switch'
138      * UnitDelay: '<Root>/X'
139      * Update for UnitDelay: '<Root>/X'
140     */
141     rtDWork.X += 2.0;

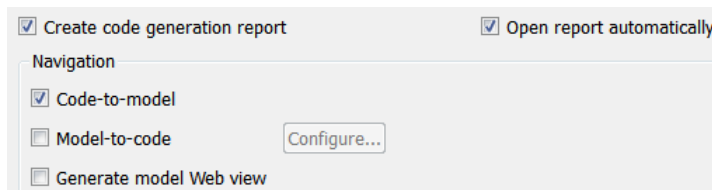
```

In the model window, the corresponding `UnitDelay` block is highlighted.

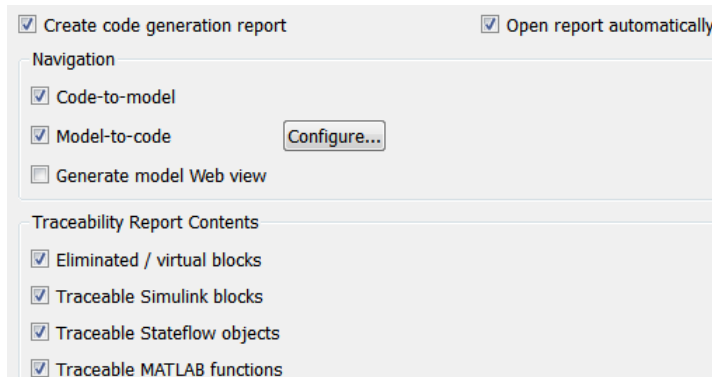


Trace Model Objects to Generated Code

- 1 Open the model and make sure that it is configured for an ERT target.
- 2 In the Configuration Parameters dialog box, select **Code Generation > Report > Create code generation report**. This parameter is selected by default. When selected, the parameter enables and selects the **Open report automatically** and **Code-to-model** parameters.



- 3 Select **Model-to-code**.



This parameter:

- Enables the **Configure** button, which opens a dialog box for loading existing trace information.
 - Enables and selects parameters for customizing the content of a traceability report.
- 4 Build or generate code for the model. An HTML code generation report is displayed.
 - 5 In the model window, right-click a model object.

- 6 In the context menu, select **C/C++ Code > Navigate to C/C++ Code**. In the HTML code generation report, you see the first instance of highlighted code that is generated for the model object. In the left pane of the report, numbers that appear to the right of generated file names indicate the total number of highlighted lines in each file. The following figure shows the result of tracing the Unit Delay block in model `rtwdemo_hyperlinks`.

```

Highlight code for block: '<Root>/X'
227 }
228 }
229
230 /* End of Chart: '<Root>/Chart' */
231
232 /* Outport: '<Root>/Out1' */
233 rtY.Out1[0] = rtDWork.y1;
234 rtY.Out1[1] = rtDWork.y2;
235
236 /* Switch: '<Root>/Switch' */
237 if (rtDWork.y3 >= 0.5) {
238 /* Update for UnitDelay: '<Root>/X' incorporates:
239 * Constant: '<Root>/INC'
240 * Sum: '<Root>/Sum'
241 * UnitDelay: '<Root>/X'
242 */
243 rtDWork.X += 2.0;
244 } else {
245 /* Update for UnitDelay: '<Root>/X' incorporates:
246 * Constant: '<Root>/RESET'
247 */
248 rtDWork.X = 0.0;
249 }
250
251 /* End of Switch: '<Root>/Switch' */
252 }

```

At the top of the code window, use the navigation toolbar to move forward and backward through multiple instances of highlighted lines. Use the navigation sidebar to go directly to a line of code.

If you close and reopen a model, the **Navigate to Code** context menu option might not be available because Embedded Coder cannot find a build folder for your model in the current working folder. Do one of the following:

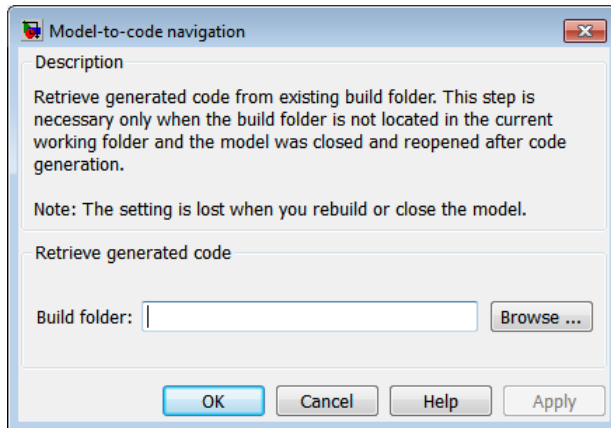
- Reset the current working folder to the parent folder of the existing build folder.
- Select **Model-to-code** and rebuild the model. Rebuilding the model regenerates the build folder into the current working folder.

- Click **Configure** and in the Model-to-code navigation dialog box, reload the existing trace information.

Reload Existing Traceability Information

To reload existing traceability information for a model:

- 1 In the Configuration Parameters dialog box, click **Code Generation > Report > Configure**. The Model-to-code navigation dialog box opens.



- 2 In the **Build folder** field, type or browse to the build folder that contains the existing traceability information.

If you close and reopen a model, the **Navigate to Code** context menu option might not be available. This occurs because Embedded Coder cannot find a build folder for your model in the current working folder. To fix this without having to reset the current working folder or rebuild the model, do the following:

- 1 Click **Configure** to open the Model-to-code navigation dialog box.
- 2 In the Model-to-code navigation dialog box, click **Browse**.
- 3 Browse to the build folder for your model, and select the folder. The build folder path is displayed in the **Build folder** field, as shown in the preceding figure.
- 4 Click **Apply** or **OK**. This loads traceability information from the earlier build into your Simulink session, provided that you selected **Model-to-code** for the build.
- 5 Right-click a model object and select **C/C++ Code > Navigate to C/C++ Code** to open the context menu and trace a model object to corresponding code.

Customize Traceability Reports

In the Configuration Parameters dialog box, the **Code Generation > Report > Traceability Report Contents** section lists parameters you can select and clear to customize the content of your traceability reports. By default, all parameters are selected.

Select or clear any combination of the following:

- **Eliminated / virtual blocks** (account for blocks that are untraceable)
- **Traceable Simulink blocks**
- **Traceable Stateflow objects**
- **Traceable MATLAB functions**

If you select all parameters, you get a complete mapping between model elements and the generated code.

The following figure shows the top section of the traceability report generated by selecting all traceability content parameters for model `rtwdemo_hyperlinks`.

Traceability Report for rtwdemo_hyperlinks

Generate
Traceability Matrix

Table of Contents

1. [Eliminated / Virtual Blocks](#)
2. [Traceable Simulink Blocks / Stateflow Objects / MATLAB Functions](#)
 - o [rtwdemo_hyperlinks](#)
 - o [rtwdemo_hyperlinks/Chart](#)
 - o [rtwdemo_hyperlinks/Chart:43](#)

Eliminated / Virtual Blocks

Block Name	Comment
<Root>/Build ERT	Empty SubSystem
<Root>/Mux	Mux
<Root>/Scope	Unused code path elimination
<Root>/View Code Generation Report	Empty SubSystem

Traceable Simulink Blocks / Stateflow Objects / MATLAB Functions

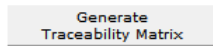
Root system: [rtwdemo_hyperlinks](#)

Object Name	Code Location
<Root>/Chart	rtwdemo_hyperlinks.c:20, 43, 85, 103, 112, 143, 257 rtwdemo_hyperlinks.h:39, 40, 41, 43, 45, 46, 47, 48, 49, 52, 53
<Root>/Constant	rtwdemo_hyperlinks.c:144

Generate a Traceability Matrix

If you are licensed for either DO Qualification Kit software or IEC Certification Kit software and are using a Windows host, you can generate a traceability matrix into Microsoft Excel format directly from the traceability report described in “Customize Traceability Reports” on page 29-10.

To do this, go to the **Traceability Report** section of the HTML code generation report and click the **Generate Traceability Matrix** button.



When you click the button, a Generate Traceability Matrix dialog box appears. Use this dialog to select an existing matrix file to update or specify a new matrix file to create. Optionally, you can use this dialog to select and order the columns that appear in the generated matrix. For more information, see “Generating a Traceability Matrix” in either the “DO Qualification Kit documentation” or the “IEC Certification Kit documentation”.

Traceability Limitations

The following limitations apply to reports generated by Embedded Coder software.

- Under the following conditions, model-to-code traceability is disabled for a block if the block name contains:
 - A single quote (').
 - An asterisk (*), that causes a name-mangling ambiguity relative to other names in the model. This name-mangling ambiguity occurs if in a block name or at the end of a block name, an asterisk precedes or follows a slash (/).
 - The character ÿ (char(255)).
- If a block name contains a newline character (\n), in the generated code comments, the block path name hyperlink replaces the newline character with a space for readability.
- You cannot trace blocks representing the following types of subsystems to generated code:
 - Virtual subsystems
 - Masked subsystems
 - Nonvirtual subsystems for which code has been optimized away

If you cannot trace a subsystem at subsystem level, you might be able to trace individual blocks within the subsystem.

Component Verification

- “Component Verification in the Target Environment” on page 30-2
- “Goals of Component Verification” on page 30-3
- “Maximizing Code Portability and Configurability” on page 30-4
- “Simplifying Code Integration and Maximizing Code Efficiency” on page 30-5
- “Running Component Tests” on page 30-6

Component Verification in the Target Environment

After you generate production code for a component design, you need to integrate, compile, link, and deploy the code as a complete application on the embedded system. One approach is to manually integrate the code into an existing software framework that consists of an operating system, device drivers, and support utilities. The algorithm can include externally written legacy or custom code.

An easier approach to verifying a component in a target environment is to use processor-in-the-loop (PIL) simulation. For information about PIL simulations, see “About SIL and PIL Simulations”.

Goals of Component Verification

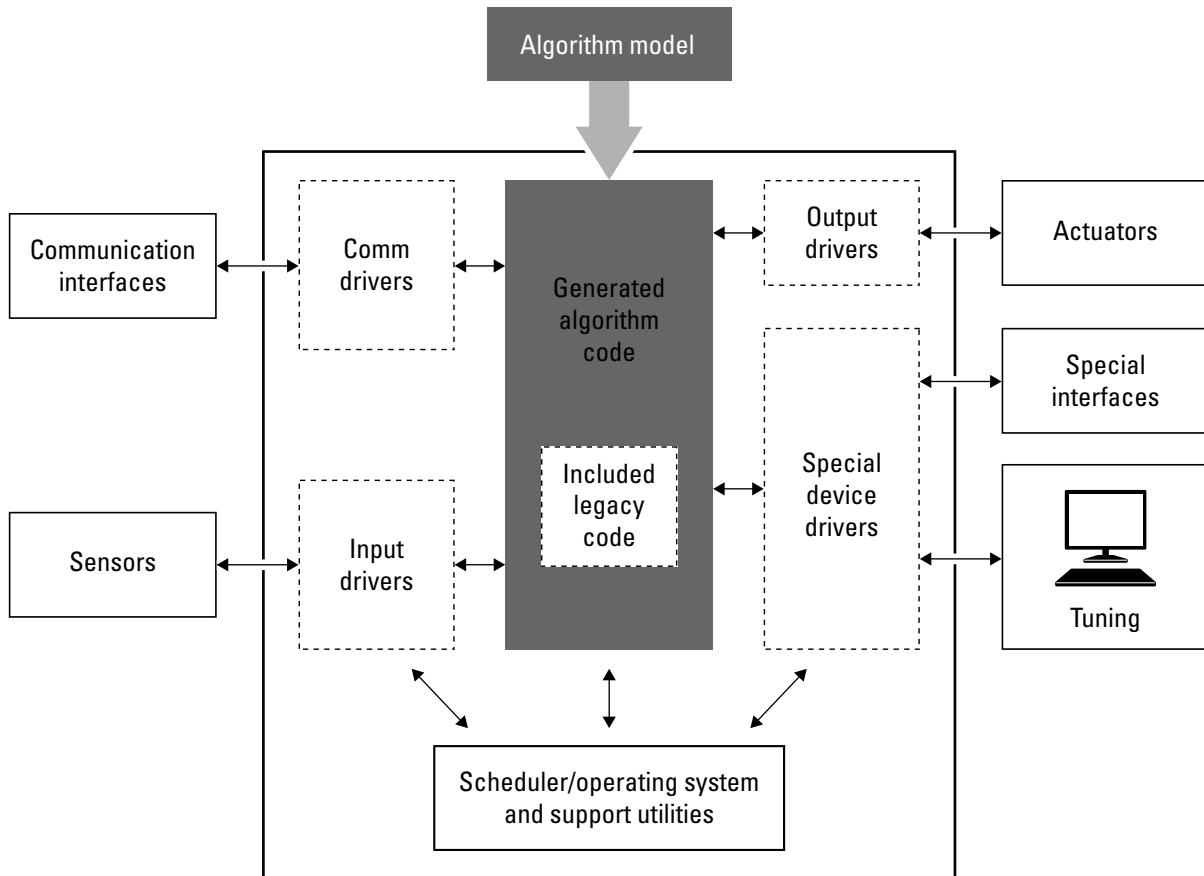
Assuming that you have generated production source code and integrated required externally written code, such as drivers and a scheduler, you can verify that the integrated software operates as expected by testing it in the target environment. During testing, you can achieve either of the following goals, depending on whether you export code that is strictly ANSI C/C++ or mixes ANSI C/C++ with code optimized for a target environment.

Goal	Type of Code Export
Maximize code portability and configurability	ANSI C/C++
Simplify integration and maximize use of processor resources and code efficiency	Mixed code

Regardless of your goal, you must integrate required external drivers and scheduling software. To achieve real-time execution, you must integrate the real-time scheduling software.

Maximizing Code Portability and Configurability

To maximize code portability and configurability, limit the application code to ANSI/ISO C or C++ code only, as the following figure shows.

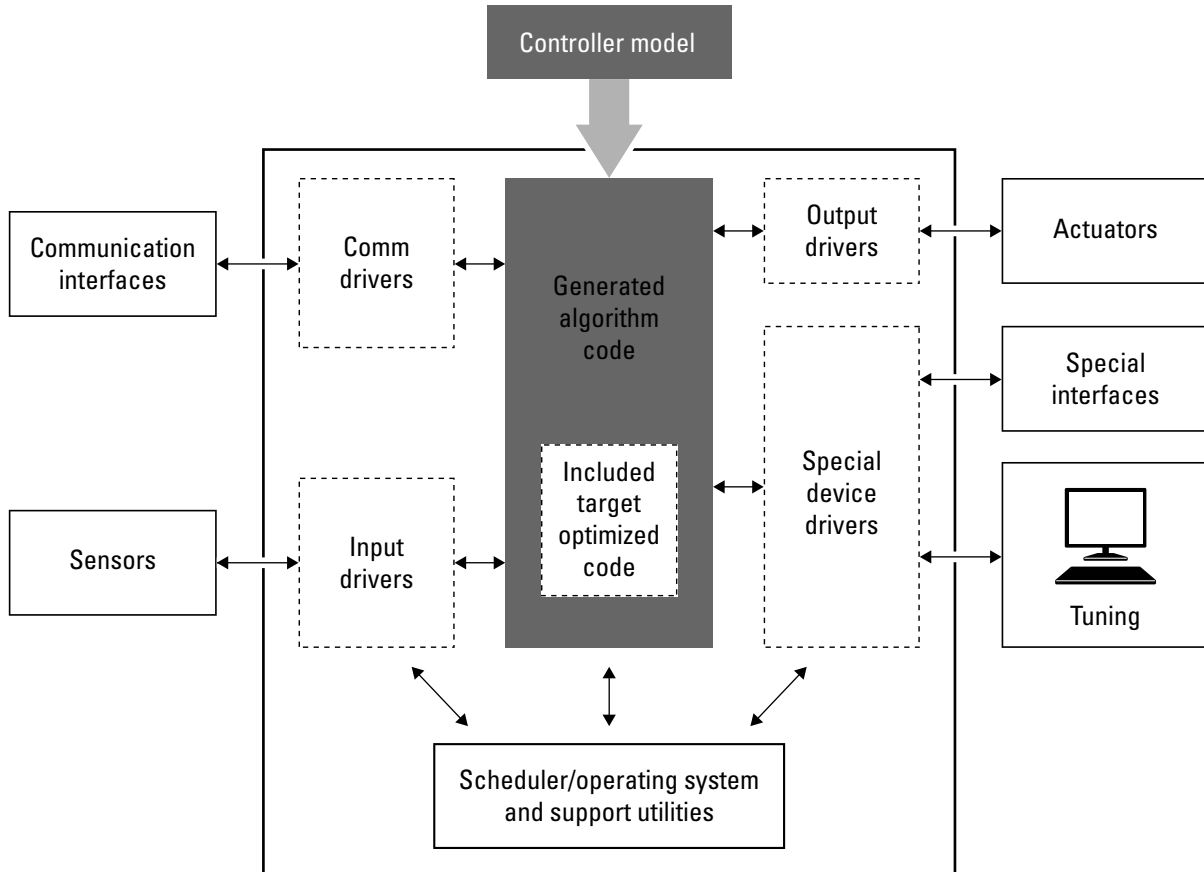


Simplifying Code Integration and Maximizing Code Efficiency

To simplify code integration and maximize code efficiency for a target environment, use Embedded Coder features for:

- Controlling code interfaces
- Exporting subsystems
- Including target-specific code, including compiler optimizations

The following figure shows a mix of ANSI C/C++ code with code that is optimized for a target environment.



Running Component Tests

The workflow for running software component tests in the target environment is:

- 1 Integrate external code, for example, for device drivers and a scheduler, with the generated C or C++ code for your component model. For more information, see “Insert S-Function Code” in the Simulink Coder documentation. For more specific references that depend on your verification goals, see the following table.

For	See
ANSI C/C++ code integration	“Integrate C Functions Using Legacy Code Tool” in the Simulink documentation. Also, open <code>rtwdemos</code> and navigate to the Custom Code folder.
Mixed code integration	<ul style="list-style-type: none"> • “Export Function-Call Subsystems” and example <code>rtwdemo_exporting_functions</code> • “Function Prototype Control”, “C++ Class Interface Control”, and example <code>rtwdemo_fcnprotoctrl</code> • “What Is Code Replacement?”, “What Is Code Replacement Customization?”, and example <code>rtwdemo_crl_script</code>

- 2 Simulate the integrated component model.
- 3 Generate code for the integrated component model.
- 4 Connect to data interfaces for the generated C code data structures. See “Data Interchange Using the C API” and “ASAP2 Data Measurement and Calibration” in the Simulink Coder documentation. Also see examples `rtwdemo_capi` and `rtwdemo_asap2`.
- 5 Customize and control the build process, if required. See “Customize Post-Code-Generation Build Processing” in the Simulink Coder documentation, and example `rtwdemo_buildinfo`.
- 6 Create a zip file that contains generated code files, static files, and dependent data to build the generated code in an environment other than your host computer. See “Relocate Code to Another Development Environment”, in the Simulink Coder documentation, and example `rtwdemo_buildinfo`.

Component Verification With a Real-Time Target Environment

- “About Real-Time Software Component Verification” on page 31-2
- “Real-Time Software Component Testing” on page 31-4

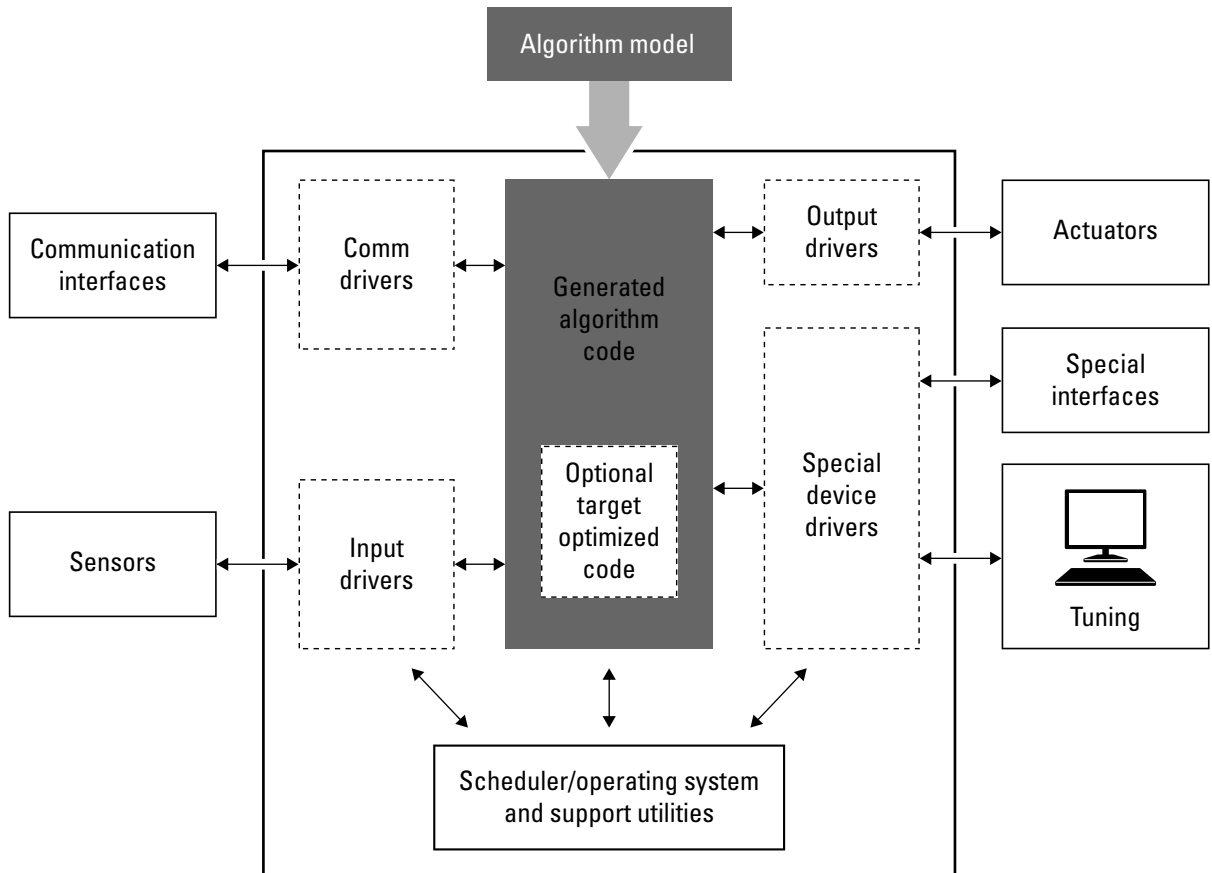
About Real-Time Software Component Verification

One approach to verifying a software component is to build the component into a complete software system that can execute in real time in the target environment. A complete software system includes:

- Algorithm for the software component
- Scheduling algorithms
- Calls to drivers for board-specific devices

This single build approach is more time consuming to set up, but makes it easier to get the complete application running in the target environment.

The following figure shows code generated for an algorithm being built into a complete system executable for the target environment.



Real-Time Software Component Testing

The workflow for testing component software as part of a complete real-time target environment is:

- 1 Develop a component model and generate source code for production.

For information on building in scheduling and real-time system support, see:

- “Scheduling” in the Simulink Coder documentation. For an example, open `rtwdemos` and navigate to the **Multirate Support** folder.
 - “Handle Asynchronous Events” in the Simulink Coder documentation and example `rtwdemo_async`
 - “Standalone Programs (No Operating System)”
 - “Workflows for AUTOSAR” and examples `rtwdemo_autosar_legacy_script` and `rtwdemo_autosar_multirunnables_script`.
- 2 Optimize generated code for a specific run-time environment, using specialized function libraries. For more information, see and example `rtwdemo_tfl_script`.
 - 3 Customize post code generation build processing to accommodate third-party tools and processes, as required. See “Customize Post-Code-Generation Build Processing” in the Simulink Coder documentation and example `rtwdemo_buildinfo`.
 - 4 Integrate external code, for example, for device drivers and a scheduler, with the generated C or C++ code for your component model. For more information, see “Insert S-Function Code” in the Simulink Coder documentation. For more specific references depending on your verification goals, see the following table.

For...	See...
ANSI C/C++ code integration	“Integrate C Functions Using Legacy Code Tool” in the Simulink documentation. Also, open <code>rtwdemos</code> and navigate to the Custom Code folder.
Mixed code integration	<ul style="list-style-type: none"> • “Export Function-Call Subsystems” and example <code>rtwdemo_exporting_functions</code> • “Function Prototype Control”, “C++ Class Interface Control”, and example <code>rtwdemo_fcnprotoctrl</code> • and example <code>rtwdemo_crl_script</code>

- 5** Simulate the integrated model.
- 6** Generate code for the integrated model.
- 7** Connect to data interfaces for the generated C code data structures. See “Data Interchange Using the C API” and “ASAP2 Data Measurement and Calibration” in the Simulink Coder documentation. Also see examples `rtwdemo_capi` and `rtwdemo_asap2`.
- 8** Customize and control the build process, as required. See “Customize Post-Code-Generation Build Processing”, in the Simulink Coder documentation, and example `rtwdemo_buildinfo`.
- 9** Create a zip file that contains generated code files, static files, and dependent data to build the generated code in an environment other than your host computer. See “Relocate Code to Another Development Environment”, in the Simulink Coder documentation, and example `rtwdemo_buildinfo`.

Numerical Equivalence Checking

- “About SIL and PIL Simulations” on page 32-2
- “Choose a SIL or PIL Approach” on page 32-7
- “Configure a SIL or PIL Simulation” on page 32-10
- “Top Model Simulation Using SIL or PIL” on page 32-17
- “Referenced Model Simulation Using SIL or PIL” on page 32-18
- “Verify Internal Signals of a Component” on page 32-20
- “Simulation Mode Override Behavior in Model Reference Hierarchy” on page 32-21
- “Code Interfaces for SIL and PIL” on page 32-23
- “Configure Hardware Implementation Settings for SIL” on page 32-25
- “Debug Code During SIL Simulations” on page 32-30
- “PIL Customization for Target Environment” on page 32-32
- “Create PIL Target Connectivity Configuration” on page 32-35
- “View Test Harness in Code Generation Report” on page 32-43
- “SIL and PIL Simulation Support and Limitations” on page 32-45
- “Programmatic Code Generation Verification” on page 32-69

About SIL and PIL Simulations

In this section...

“What are SIL and PIL Simulations?” on page 32-2

“Why Use SIL and PIL” on page 32-3

“How SIL and PIL Simulations Work” on page 32-4

“Comparison of SIL and PIL Simulation” on page 32-5

What are SIL and PIL Simulations?

The Embedded Coder product supports software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulations.

A SIL simulation involves compiling and running production source code on your host computer to verify the source code. SIL provides a convenient alternative to processor-in-the-loop (PIL) simulation as no target hardware (for example, an evaluation board or instruction set simulator) is required. For examples of SIL verification, see `rtwdemo_sil_pil_script`.

A PIL simulation involves cross-compiling and running production object code on a target processor or an equivalent instruction set simulator.

You can run a SIL or PIL simulation using:

- The **Software-in-the-Loop (SIL)** or **Processor-in-the-Loop (PIL)** simulation mode for top models and Model blocks
- A SIL or PIL block

For more information, see “Choose a SIL or PIL Approach” on page 32-7.

The following features enable you to verify the generated code:

- Ability to compare the output of regular simulation modes, for example, Normal or Accelerator, against the output of SIL and PIL simulation modes.
- Easy switching between regular simulation, SIL, and PIL modes.

You can model and test your embedded software component in Simulink and then reuse your test suites across simulation and compiled production code. This approach avoids the time-consuming process of leaving the Simulink software environment and verifying production code on a separate test infrastructure.

For information about how you verify the configuration of your model for a SIL or PIL simulation, see “Check the SIL or PIL Configuration”.

Why Use SIL and PIL

Through SIL and PIL, you can achieve early verification and fixing of defects. The following table describes situations where you can use SIL and PIL.

Situation	Use
You want to reuse test vectors developed for Normal mode simulation to verify numerical output of generated (or legacy) code. For example, reusing test cases generated by Simulink Design Verifier™. See “What Is Test Case Generation?” in the Simulink Design Verifier documentation.	SIL and PIL
You want to collect metrics for generated code: <ul style="list-style-type: none"> • Code coverage. See “Configure SIL and PIL Code Coverage”. • Execution profiling. See “Code Execution Profiling for SIL and PIL” • Stack profiling. See “Perform Stack Profiling with IDE and Toolchain Targets”. 	SIL and PIL
You want to achieve IEC 61508, ISO 26262, and DO-178 certification. See “Embedded Coder Reference Workflow Overview” in the IEC Certification Kit documentation and “Testing of Outputs of Integration Process” in the DO Qualification Kit documentation.	SIL and PIL
You do not have target hardware and want a convenient alternative to PIL.	SIL
You have target hardware, for example, an evaluation board or instruction set simulator, and you want to: <ul style="list-style-type: none"> • Verify behavior of target-specific code, for example, code replacement optimizations, and legacy code. See “What Is Code Replacement?” and “What Is Code Replacement Customization?”. • Optimize the execution speed and memory footprint of your code. See the row in this table about collecting execution profiling and stack profiling metrics. 	PIL

Situation	Use
<ul style="list-style-type: none"> Investigate effects of compiler settings and optimizations, for example, deviation from ANSI C overflow behavior. <p>Normal simulation techniques do not account for restrictions and requirements that the hardware imposes, such as limited memory resources or behavior of target-specific optimized code.</p> <p>See “Sample Custom Targets” in the Simulink Coder documentation, which gives information about running PIL simulations on specific targets.</p>	

Note: The SIL and PIL simulation modes are not designed for reducing model simulation times. If you want to speed up the simulation of your model, use the Rapid Accelerator mode. For more information, see:

- “Acceleration”
- Rapid Accelerator Simulations Using PARFOR

How SIL and PIL Simulations Work

In a SIL/PIL simulation, code is generated for either the top model or part of the model. With SIL, this code is compiled for, and executed on the host computer. With PIL, the code is cross-compiled for the target hardware and runs on the target processor.

Through a communication channel, Simulink sends stimulus signals to the code on the host or target processor for each sample interval of the simulation:

- For a top model, Simulink uses stimulus signals from the base or model workspace.
- If you have designated only part of the model to simulate in SIL/PIL mode, then a part of the model remains in Simulink without the use of code generation. Typically, you configure this part of the model to provide test vectors for the software executing on the hardware. This part of the model can represent other parts of the algorithm or the environment model in which the algorithm operates.

When the host/target processor receives signals from Simulink, the processor executes the SIL/PIL algorithm for one sample step. The SIL/PIL algorithm returns output signals computed during this step to Simulink through a communication channel. One sample

cycle of the simulation is complete, and Simulink proceeds to the next sample interval. The process repeats and the simulation progresses. SIL/PIL simulations do not run in real time. In each sample period, Simulink and the object code exchange I/O data.

Comparison of SIL and PIL Simulation

Use SIL or PIL simulation to verify automatically generated code by comparing the results with a Normal mode simulation. With SIL, you can easily verify the behavior of production source code on your host computer. However, you cannot verify the same code that is subsequently compiled for your target hardware because the code is compiled for your host computer (that is, a different compiler and different processor architecture than the target). With PIL simulation, you can verify the same code that you intend to deploy in production, and you can run the code on either real target hardware or an instruction set simulator. See “What are SIL and PIL Simulations?” on page 32-2.

You can use the following approaches to verification.

Approach	SIL	PIL
Simulation mode (for top model or Model block)	Generated production code compiled and executed on host computer as separate process, independent of the MATLAB process. Execution is host/host and nonreal time.	Test the generated code as cross-compiled object code on target processor or instruction set simulator. Exercises same object code used in production software. Execution is host/target and nonreal time.
Block	Create SIL block. Software runs compiled object code through S-function wrapper. S-function communicates with object code executing as standalone application on host computer. SIL block execution is independent of the MATLAB process. Execution is host/host and nonreal time. See “Use a SIL or PIL Block” on page 32-13.	Create PIL block. Software runs cross-compiled object code through S-function wrapper on host computer. S-function communicates with object code executing as standalone application on target processor or instruction set simulator. Execution is host/target and nonreal time. See “Use a SIL or PIL Block” on page 32-13.

To decide which verification approach you want to use, see “Choose a SIL or PIL Approach” on page 32-7 .

Choose a SIL or PIL Approach

In this section...

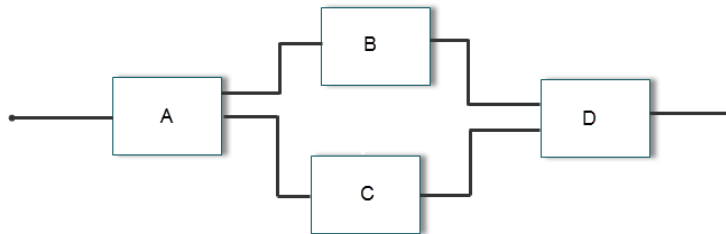
“Verify Top Model Code” on page 32-8

“Verify Referenced Model Code” on page 32-9

“Verify Subsystem Code” on page 32-9

Consider a top model that consists of components A, B, C, and D:

- A and B are existing components for which code has previously been generated and tested.
- C, a referenced model, and D, a subsystem, are new components.



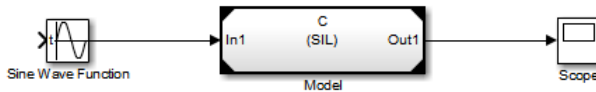
You can use the following approaches to code verification:

- Verify code from all components together. See “Verify Top Model Code” on page 32-8.
- Verify new components separately (before verifying code from all components). See “Verify Referenced Model Code” on page 32-9 and “Verify Subsystem Code” on page 32-9.

For some forms of code verification, you require a test harness model. The test harness model:

- Generates test vectors or stimulus inputs that feed the block under test.
- Makes it possible for you to observe or capture output from the block.

The following example shows a simple test harness model.



The block under test is a Model block. The Sine Wave block generates the input for the Model block. Through the Scope block, you can observe the output from the Model block.

For examples that use the various approaches to SIL/PIL code verification, see `rtwdemo_sil_pil_script`.

For information about running top-model SIL/PIL, Model block SIL/PIL, and SIL/PIL block simulations, see “Configure a SIL or PIL Simulation” on page 32-10.

Verify Top Model Code

To verify code generated from the top-model components together (A, B, C and D), you can use top-model SIL/PIL or Model block SIL/PIL.

- Top-model SIL/PIL:
 - 1 Create test vectors or stimulus inputs in the MATLAB workspace.
 - 2 Run the top model in Normal, SIL, and PIL simulation modes. The software loads the test vectors or stimulus inputs from the MATLAB workspace.
 - 3 For each simulation mode, observe or capture outputs.
 - 4 Verify the generated code by comparing the Normal and SIL and PIL outputs.
- Model block SIL/PIL:
 - 1 Create a Model block that contains the top-model components.
 - 2 Insert the Model block in your test harness model.
 - 3 Run simulations, switching the Model block between Normal, SIL, and PIL modes. For the SIL and PIL simulation modes, set the **Code under test** Model block parameter to `Top model`.
 - 4 Verify the generated code by comparing the Normal and SIL and PIL outputs.

Verify Referenced Model Code

To verify code generated from the component **C** as part of a model reference hierarchy, use the Model block SIL/PIL approach:

- Insert the Model block **C** in your test harness model.
- Run simulations, switching the Model block between Normal, SIL, and PIL modes. For the SIL and PIL simulation modes, set the **Code under test** Model block parameter to **Model reference**.
- Verify the generated code by comparing the Normal and SIL and PIL outputs.

Verify Subsystem Code

To verify code generated from the subsystem **D**, use the SIL or PIL block approach:

- 1** Insert the subsystem in your test harness model.
- 2** Run a Normal mode simulation, capturing the outputs.
- 3** Create a SIL or PIL block from the subsystem.
- 4** In the test harness model, replace the subsystem with the SIL or PIL block.
- 5** Run a simulation of the test harness model, capturing the outputs.
- 6** Verify the generated code by comparing the SIL or PIL block outputs against the Normal mode subsystem outputs.

For information about how you create a SIL or PIL block, see “Use a SIL or PIL Block” on page 32-13.

Configure a SIL or PIL Simulation

In this section...

- “Top-Model SIL or PIL Simulation” on page 32-10
- “Model Block SIL or PIL Simulation” on page 32-12
- “Use a SIL or PIL Block” on page 32-13
- “Check the SIL or PIL Configuration” on page 32-14

Top-Model SIL or PIL Simulation

To configure and run a top-model SIL or PIL simulation:

- 1 Open your model.
- 2 Select either **Simulation > Mode > Software-in-the-Loop (SIL)** or **Simulation > Mode > Processor-in-the-Loop (PIL)**.

Note: This option is available only if the model is configured for an ERT or AUTOSAR target. See “Code Generation Pane: General” and “Export AUTOSAR Component XML and C Code” for configuration information.

- 3 If you have not already done so, in the Configuration Parameters dialog box, on the **Data Import/Export** pane:
 - In the **Input** check box and field, specify stimulus signals (or test vectors) for your top model.
 - Configure logging for model outputs, using either *output logging* or *signal logging*:
 - In the **Output** check box and field, specify *output logging*.
 - In the **Signal logging** check box and field, specify *signal logging*.

See “Internal Signal Logging Support” and “Top-Model Root-Level Logging Limitations”.

- Disable logging of Data Store Memory variables. The software does not support this option for this simulation mode. If you do not clear the **Data stores** check box, the software produces a warning when you run the simulation.

- 4 If you are configuring a SIL simulation, specify the portable word sizes option. This option allows you to switch seamlessly between the SIL and PIL modes. Select **Code Generation > Verification > Enable portable word sizes**. See “Configure Hardware Implementation Settings for SIL” on page 32-25.
- 5 If required, configure:
 - Code coverage. See “Configure SIL and PIL Code Coverage”.
 - Code execution profiling. See “Configure Code Execution Profiling for SIL and PIL”.
 - Creation of code generation report and static code metrics. See “View Test Harness in Code Generation Report” on page 32-43.
- 6 Start the simulation.

Note: On a Windows operating system, the Windows Firewall might block your SIL simulation. To allow the SIL simulation, use the Windows Security Alert dialog box . For example, in Windows 7, click **Allow access**.

You cannot:

- Close the model while the simulation is running. To interrupt the simulation, in the Command Window, press **Ctrl+C**.
- Alter the model during the simulation. You can move blocks and lines as long as it does not alter the behavior of the model.

You can run a top-model SIL or PIL simulation using the command `sim(model)`. The software supports the `sim` command options `SrcWorkspace` and `DstWorkspace` for only the following values:

- `SrcWorkspace` — 'base'
- `DstWorkspace` — 'base' or 'current'

For more information on the `sim` command and its options, see “Run Simulation Using the `sim` Command”.

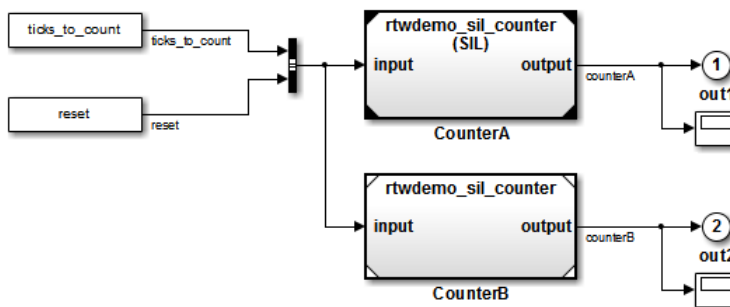
For information about how a simulation behaves when the top model contains a Model block (and this Model block is a parent Model block containing Model blocks at lower levels of its reference hierarchy), see “Simulation Mode Override Behavior in Model Reference Hierarchy” on page 32-21.

For a PIL simulation, you control the way code compiles and executes in the target environment through connectivity configurations. See “Create PIL Target Connectivity Configuration” on page 32-35.

Model Block SIL or PIL Simulation

To configure a Model block for a SIL or PIL simulation:

- 1 Open your model, for example, `rtwdemo_sil_modelblock`.
- 2 Right-click your Model block, for example, **Counter A**. In the context menu, select **Block Parameters (ModelReference)**, which opens the Function Block Parameters dialog box.
- 3 From the **Simulation Mode** drop-down list, select the required mode, for example, **Software-in-the-loop (SIL)**.
- 4 From the **Code under test** drop-down list, specify the code that you want to test, for example, `Model reference`.
- 5 Click **OK**. The software displays the simulation mode as a block label.



If you select `Top model`, the software displays the block label (`SIL: Top`).

- 6 If you are configuring a SIL simulation, specify the portable word sizes option. This option allows you to switch seamlessly between the SIL and PIL modes. Select **Code Generation > Verification > Enable portable word sizes**. See “Configure Hardware Implementation Settings for SIL” on page 32-25.
- 7 If required, configure:
 - Code coverage. See “Configure SIL and PIL Code Coverage”.

- Code execution profiling for your Model block, by configuring execution profiling for the top model. See “Configure Code Execution Profiling for SIL and PIL”.
- Creation of code generation report and static code metrics. See “View Test Harness in Code Generation Report” on page 32-43.

8 Start the simulation.

Note: On a Windows operating system, the Windows Firewall might block your SIL simulation. To allow the SIL simulation, use the Windows Security Alert dialog box . For example, in Windows 7, click **Allow access**.

For a PIL simulation, you control the way code compiles and executes in the target environment through connectivity configurations. See “Create PIL Target Connectivity Configuration” on page 32-35.

Use a SIL or PIL Block

You can automatically create a SIL or PIL block from a subsystem and use this block to test the code generated from the subsystem:

- 1** In the Configuration Parameters dialog box, select **Code Generation > Verification**.
- 2** From the **Create block** drop-down list, select either **SIL** or **PIL**.
- 3** If required, configure code execution profiling. For details, see “Code Execution Profiling for SIL and PIL”.
- 4** Click **OK**.
- 5** In your model window, right-click the subsystem that you want to simulate.
- 6** Select **C/C++ Code > Build This Subsystem**.
- 7** Click **Build** to start a subsystem build that generates a SIL or PIL block for the generated subsystem code.
- 8** Add the generated block to an environment or test harness model that supplies test vectors or stimulus input.
- 9** Run simulations with the environment or test harness model to perform SIL or PIL tests.
- 10** Verify that the generated code in the SIL or PIL block provides the same results as the original subsystem.

Note: On a Windows operating system, the Windows Firewall might block your SIL simulation. To allow the SIL simulation, use the Windows Security Alert dialog box . For example, in Windows 7, click **Allow access**.

You cannot create a SIL or PIL block if you do one of the following:

- Disable the `CreateSILPILBlock` property.
- Select a code coverage tool.

Create block appears dimmed.

For a PIL simulation, you control the way code compiles and executes in the target environment through connectivity configurations. See “Create PIL Target Connectivity Configuration” on page 32-35.

For an example of how you can use the SIL block in testing, see `rtwdemo_sil_pil_script`.

Check the SIL or PIL Configuration

To run a SIL or PIL simulation, you might need to change some model settings. To find out what settings you must change, use the `cgv.Config` class. Using the `cgv.Config` class, you can review your model configuration and determine which settings you must change to configure the model for SIL or PIL. By default, `cgv.Config` changes configuration parameter values to the value that it recommends, but does not save the model. Alternatively, you can specify that `cgv.Config` use one of the following approaches:

- Change configuration parameter values to the values that `cgv.Config` recommends, and save the model. Specify this approach using the `SaveModel` property.
- List the values that `cgv.Config` recommends for the configuration parameters, but do not change the configuration parameters or the model. Specify this approach using the `ReportOnly` property.

Note:

- To execute the model in the target environment, you might need to make additional modifications to the configuration parameter values or the model.

- Do not use referenced configuration sets in models that you are changing using `cgv.Config`. If the model uses a referenced configuration set, update the model with a copy of the configuration set. Use the `Simulink.ConfigSetRef.getRefConfigSet` method. For more information, see `Simulink.ConfigSetRef` in the Simulink documentation.
 - If you use `cgv.Config` on a model that executes a callback function, the callback function might change configuration parameter values each time the model loads. The callback function might revert changes that `cgv.Config` made. When this change occurs, the model might no longer be set up for SIL or PIL. For more information, see “Callbacks for Customized Model Behavior”.
-

For more information about the `cgv.Config` class, see `cgv.Config`.

How To Verify a SIL or PIL Configuration

To verify that your model is configured for SIL or PIL:

- 1 Construct a `cgv.Config` object that changes the configuration parameter values without saving the model. For example, to configure your model for SIL:

```
c = cgv.Config('vdp', 'connectivity', 'sil');
```

Tip

- You can obtain a list of changes without changing the configuration parameter values. When you construct the object, include the `'ReportOnly', 'on'` property name and value pair.
 - You can change the configuration parameter values and save the model. When you construct the object, include the `'SaveModel', 'on'` property name and value pair.
-

- 2 Determine and change the configuration parameter values that the object recommends using the `configModel` method. For example:

```
c.configModel();
```

- 3 Display a report of the changes that `configModel` makes. For example:

```
c.displayReport();
```

- 4** Review the changes.
- 5** To apply the changes to your model, save the model.

Top Model Simulation Using SIL or PIL

With a top-model SIL or PIL simulation:

- Simulink generates and executes code that uses the same code interface produced by the standalone build process. See “Code Interfaces for SIL and PIL” on page 32-23.
- You can load data from the MATLAB workspace to specify stimulus signals, and you can log output signals, which allows you to verify object code generated from a complete model without creating a separate test harness model. Running the SIL or PIL simulation is a simple operation.

You can use the Model block approach as an alternative to top-model SIL or PIL simulation. See “Choose a SIL or PIL Approach” on page 32-7.

For the top-model SIL/PIL approach, Simulink creates a hidden *wrapper* model. When you run a top-model SIL simulation, the software generates code for the model and creates a hidden wrapper model to call this code at each time step.

As a result, in some circumstances, logged signals might have a `_wrapper` suffix. See “Top-Model Root-Level Logging Limitations”.

During a SIL/PIL simulation, the software can generate warnings that refer to the hidden wrapper model. For example:

```
Warning: The model 'modelName_wrapper' has the 'Configuration Parameters' ...
```

Referenced Model Simulation Using SIL or PIL

In addition to the Normal and Accelerator simulation modes, `Model` blocks support the Software-in-the-loop (SIL) and Processor-in-the-loop (PIL) simulation modes.

You can switch easily between the simulation modes. This feature allows you to verify the generated code by executing the referenced model as compiled code on the host computer or target platform. You can model and test your embedded software component in Simulink and reuse your regression test suites across simulation and compiled object code. With this capability, you can avoid the time-consuming process of leaving the Simulink software environment to run tests on object code compiled for your production hardware.

When you set the **Simulation mode** (`SimulationMode`) parameter to **Software-in-the-loop (SIL)** or **Processor-in-the-loop (PIL)**, you can specify the code under test through the Function Block Parameters dialog box.

Code under test (CodeUnderTest) Setting	Description
Top model	Code generated from top model with the standalone code interface. Code generation uses the <code>slbuild('model')</code> command. On the Model block, under the model name, the text label is (SIL: Top) or (PIL: Top).
Model reference	Code generated from referenced model as part of a model reference hierarchy. Code generation uses the <code>slbuild('model', 'ModelReferenceRTWTarget')</code> command. On the Model block, under the model name, the text label is (SIL) or (PIL).

For more information, see:

- “Verify Referenced Model Code” on page 32-9
- “Configure a SIL or PIL Simulation” on page 32-10
- “Simulation Mode Override Behavior in Model Reference Hierarchy” on page 32-21

- “Code Interfaces for SIL and PIL” on page 32-23

Verify Internal Signals of a Component

Outputs of the SIL or PIL component are available for verification. If you want to examine an internal signal, you can:

- Enable internal signal logging for top-model or Model block SIL or PIL, but check the limitations. See “Internal Signal Logging Support”.
- Manually route the signal to the top level.
- Use global data stores to access internal signals:
 - 1** Inside the component, connect a Data Store Write block to the required signal.
 - 2** Outside the component, use a Data Store Read block to access the signal value.See “About Data Stores” and “Global Data Store Example”.
- Use MAT-file logging. See “Logging”. For PIL, the target environment must support MAT-file logging.

For more information on signal support, see “I/O Support” on page 32-56.

Simulation Mode Override Behavior in Model Reference Hierarchy

This section describes simulation behavior when the top model contains a Model block. This Model block can also be a parent block containing child Model blocks at lower levels of its reference hierarchy.

Note: You can view your model hierarchy in the Model Dependency Viewer. In the Referenced Model Instances view, the software displays Model blocks differently to indicate their simulation modes, for example, Normal, Accelerator, SIL, and PIL. In this view, the software does not indicate the simulation mode of the top model.

You can specify the simulation mode of a top model to be Normal, Accelerator, Rapid Accelerator, SIL, or PIL. With a Model block, you can specify all modes *except* Rapid Accelerator. The configured simulation mode of a Model block can be overridden by the parent simulation mode. The following table shows how the software determines the effective simulation mode of a Model block in the hierarchy.

Mode of top model or parent block	Mode of parent or child block in reference hierarchy			
	Normal	Accelerator	SIL	PIL
Normal	Equivalent	Compatible	Compatible	Compatible
Accelerator	Override	Equivalent	Error	Error
Rapid Accelerator	Override	Override	Error	Error
SIL	Override	Override	Equivalent	Error
PIL	Override	Override	Error	Equivalent

The following list explains the different types of simulation behavior:

- Equivalent – Both parent and child Model block run in the same simulation mode.
- Compatible – If the simulation mode of the top model or parent block is Normal, then the software simulates the child block in the mode specified for it.
- Error – The simulation produces an error. For example, if a top model or parent Model block has simulation mode Accelerator but contains a child block in SIL or PIL mode, then running a simulation produces an error: the Accelerator mode cannot override the SIL and PIL mode of child blocks. This behavior avoids the risk of “false positives”, that is, the simulation of a model in Accelerator mode will not lead to the

conclusion that generated source or object code of child Model blocks has been tested or verified.

- **Override** – The simulation mode of the top model or parent Model block overrides the simulation mode of the child block. For example, if a top model or parent Model block that is configured for a SIL simulation contains a child Model block with simulation mode Normal or Accelerator, then the software simulates the child block in SIL mode. This override behavior:
 - Allows a Model block in the reference hierarchy to have the SIL or PIL mode.
 - Makes lower-level referenced models execute in SIL or PIL mode if you simulate the top model or parent Model block in SIL or PIL mode. You do not have to switch manually the simulation mode of every model component in the hierarchy.

Note: For a model reference hierarchy that consists of multiple sub-hierarchies, if the top-model simulation mode is Normal, the software can run only one sub-hierarchy in PIL mode. For example, if your Normal mode top model contains multiple Model blocks, you can specify the PIL mode for only one of the Model blocks.

Code Interfaces for SIL and PIL

In this section...

“Code Interface for Top-Model SIL or PIL” on page 32-23

“Code Interface for Model Block SIL or PIL” on page 32-24

You generate standalone code when you perform, for example, a top-model or right-click subsystem build for a single deployable component. You can compile and link standalone code into a standalone executable or integrate it with other code. For more information on the standalone code interface, see “Entry-Point Functions and Scheduling”.

When you generate code for a referenced model hierarchy, the software generates standalone executable code for the top model, and a library module called a *model reference target* for each referenced model. When the code executes, the standalone executable invokes the applicable model reference targets to compute the referenced model outputs. For more information, see “Build Model Reference Targets”.

Note: If you intend to integrate automatically generated code with legacy code, use standalone code because the standalone code interface is documented. See “Entry-Point Functions and Scheduling”.

SIL/PIL Feature	Standalone Code Interface	Model Reference Code Interface
Top-model	Yes	No (but you can include Model blocks inside your top model)
Model block	Yes (if you set Code under test to Top model)	Yes (if you set Code under test to Model reference)
SIL or PIL block	Yes	No

Code Interface for Top-Model SIL or PIL

Top-model SIL or PIL generates the *standalone code interface* for the model.

When you run a top-model SIL or PIL simulation, the software calls the standalone code for the model if it already exists. The software generates the standalone code if it does not exist.

Code Interface for Model Block SIL or PIL

For Model Block SIL or PIL, the value of the **Code under test** block parameter determines the code interface:

- **Top model** — The software generates the *standalone code interface* for the model. When you run a simulation, the software calls the standalone code for the model if it already exists. The software generates the standalone code if it does not exist.
- **Model reference** — The software generates the *model reference* code interface. When you run a simulation with a Model block in SIL or PIL mode, the software calls the model reference target for the Model block if it already exists, or generates the model reference target. If the model reference target does not yet exist, there are three ways to generate it:
 - Run the simulation.
 - Press **Ctrl+B** to build the top model containing the Model block.
 - Use the command `slbuild`, specifying the model reference option, for example:

```
slbuild('model', 'ModelReferenceRTWTargetOnly')
```

Configure Hardware Implementation Settings for SIL

In this section...

“Choose Hardware Implementation Approach” on page 32-25

“Portable Word Sizes” on page 32-25

“Test Hardware” on page 32-28

“Production hardware” on page 32-29

Choose Hardware Implementation Approach

Approach	Use when
Portable word sizes	You want to switch between the SIL and PIL modes without regenerating code.
Test hardware	You want to work around a limitation of portable word sizes.
Production hardware	Production hardware settings match your host computer architecture.

For information about test and production targets, “Platform Options for Development and Deployment” in the Simulink Coder documentation.

Portable Word Sizes

Embedded Coder provides an option to specify portable word sizes. If you select this option for a model, you can use the same generated source code files for:

- Software-in-the-loop (SIL) simulation on the host computer
- Production deployment on the target platform

To configure a model to use portable word sizes, set the following model configuration parameters.

Parameter	Setting
Hardware Implementation > Test hardware > Test hardware is the same as production hardware	Selected

Parameter	Setting
Code Generation > Verification > Enable portable word sizes	Selected

When you generate code for a model with the preceding parameter settings, the code generator conditionalizes data type definitions:

- `tmwtypes.h` supports SIL simulation on the host system
- Code generation types support deployment on the target system

For example, in the following generated code, the first two lines define types for SIL simulation on a host system. The **bold** lines define types for target deployment.

```
#ifdef PORTABLE_WORDSIZES    /* PORTABLE_WORDSIZES defined */
# include "tmwtypes.h"
#else                          /* PORTABLE_WORDSIZES not defined */
#define __TMWTYPES__
#include <limits.h>
...
typedef signed char int8_T;
typedef unsigned char uint8_T;
typedef int int16_T;
typedef unsigned int uint16_T;
typedef long int32_T;
typedef unsigned long uint32_T;
typedef float real32_T;
typedef double real64_T;
...
#endif                          /* PORTABLE_WORDSIZES */
```

Compile Generated Code That Supports Portable Word Sizes

When you compile generated code that supports portable word sizes for SIL testing, pass the definition `PORTABLE_WORDSIZES` to the compiler.

For example:

```
-DPORTABLE_WORDSIZES
```

To build the same code for target deployment, compile the code without the `PORTABLE_WORDSIZES` definition.

Code that the Host Cannot Compile

Consider the case where your target uses code that the host cannot compile. When you switch from the PIL mode to the SIL mode and try to simulate the model, you see compilation errors. You might be able to work around this problem by adding the source code files to the `SkipForSil` group in the build information object `RTW.BuildInfo`. The SIL build on the host platform does not compile source files present in the `SkipForSil` group. For information about how you add source code files to a group in the build information object, see:

- “addSourceFiles” in the Simulink Coder reference documentation
- “Customize Post-Code-Generation Build Processing” in the Simulink Coder documentation

Portable Word Sizes Limitations

The following limitations apply when using portable word sizes in SIL simulation:

- Numerical results might differ between generated code executing in a SIL simulation and generated code executing on the production hardware under one of the following conditions:
 - Your model contains blocks implemented in TLC, for which C integral promotion in expressions might behave differently between the MATLAB host and the production hardware target. Normal and PIL simulation results will match, but SIL simulation results might be different.
 - Your production hardware implements rounding to `Floor` for signed integer division, and divisions in your model use rounding mode `Floor` or `Simplest`. Normal and PIL simulation results will match, but SIL simulation results might be different.
 - The precision of floating-point operations differs between the MATLAB host and the production hardware target. In this case, Normal and SIL simulation results will match, but PIL simulation results might be different.
 - You use custom code with the Stateflow product. In this case, type conversion statements will not be inserted into the custom code, which might be required to achieve target overflow behavior on the host. Normal and PIL simulation results will match, but SIL simulation results might be different.
- Compilation warnings might occur for code generated using portable word sizes if all of the following conditions exist:

- The combination of MATLAB host and production hardware target word sizes causes `rtwtypes.h` to redefine the word sizes using preprocessor macros. For example, when the production hardware has a 16-bit `int` data type and the MATLAB host has a 16-bit `short` data type, `int16_T` is redefined to be `short` on the host and `int` on the target.
- The data types are used in pointer arguments to function calls.
- The called functions are host-based precompiled functions (not compiled using `rtwtypes.h`).

Under these conditions, the compiler typically issues a warning similar to the following:

```
warning: passing argument 2 of 'frexp' from incompatible pointer type
```

Executing the generated code on the MATLAB host could lead to memory corruption. For example, the function `"double frexp (double value, int *exp);"` expects `'int *'` as the second argument, for which `'int16_T *'` is passed in the generated code. But on the MATLAB host, `int16_T` is redefined to `short`, and during SIL execution, `frexp` will attempt to write 4 bytes to a 2 byte location.

A potential workaround for the SIL workflow is to provide a custom code replacement library for functions that write to address locations obtained through pointer arguments. In the above example, the function `frexp` is called by the reciprocal square root operation (`rSqrt`) and `rSqrt` is replaceable using a code replacement library. Therefore, you can provide a custom version of `rSqrt` to support SIL execution. The replacement function would perform the change in memory allocation for the data accessed by the pointer variable, perhaps by introducing a temporary variable and transferring the data to and from that variable. For more information, see “What Is Code Replacement?” and “What Is Code Replacement Customization?”.

Test Hardware

Use this approach only if you need to work around a limitation of portable word sizes.

To configure a model for test hardware, set the following model configuration parameters.

Parameter	Setting
Code Generation > Verification > Enable portable word sizes	Not selected
Hardware Implementation > Test hardware > Test hardware is the same as production hardware	Not selected
Hardware Implementation > Test hardware > Device vendor	Generic
Hardware Implementation > Test hardware > Device type	MATLAB Host Computer

For an example of how to configure a model to maintain bit-true agreement between host simulation and target deployment, and generate code that is portable between the host and target systems, see `rtwdemo_sil_hardware_config`.

Production hardware

You can use this approach only when the production hardware settings match your host computer architecture.

Set the following model configuration parameters.

Parameters	Settings
Code Generation > Verification > Enable portable word sizes	Not selected.
Hardware Implementation > Test hardware > Test hardware is the same as production hardware	Selected.
Hardware Implementation > Production hardware section parameters	Select settings that match your host computer architecture.

Debug Code During SIL Simulations

If you notice differences between the results of a Normal mode and SIL mode simulation, you can rerun the SIL simulation with a debugger enabled. By inserting breakpoints, you can observe the behavior of code sections, which might help you to understand the cause of the differences in results.

The software supports the following debuggers;

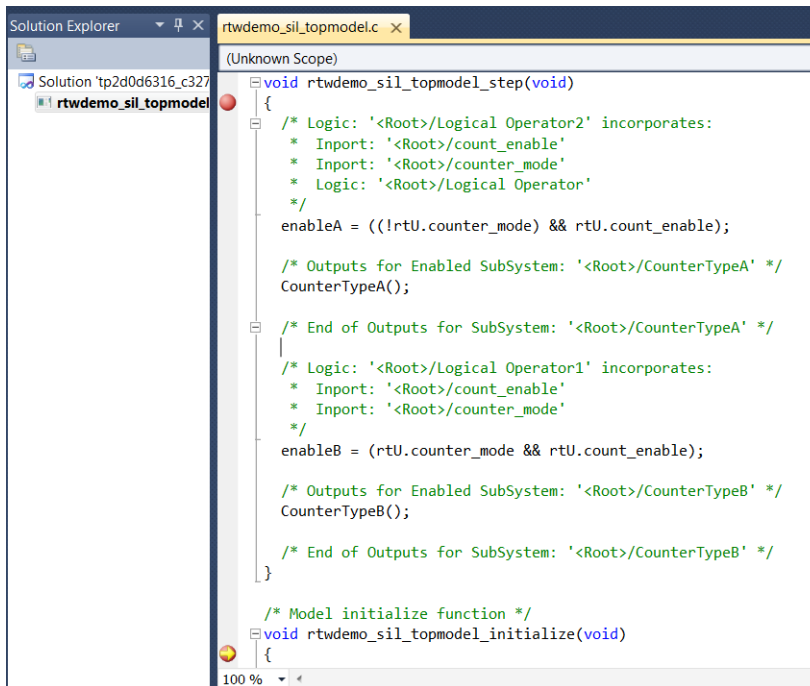
- On Windows, Microsoft Visual Studio® debugger.
- On Linux, GNU Data Display Debugger (DDD).

Note: You can perform SIL debugging only if your Microsoft Visual C++ or GNU GCC compiler is supported by the Simulink product family. For more information, see supported compilers.

To enable your debugger for a SIL simulation, on the **Configuration Parameters > Code Generation > Verification** pane, select the **Enable source-level debugging for SIL simulations** check box.

If your top model has Model blocks where the **Code under test** block parameter is set to `Top model`, then the **Enable source-level debugging for SIL simulations** parameters for the top model and referenced models must have the same settings. Otherwise, the software produces an error.

When you run the SIL simulation, for example on a Windows computer, your `model.c` or `model.cpp` file opens in the Microsoft Visual Studio IDE with debugger breakpoints at the start of the `model_initialize` and `model_step` functions.



```
Solution Explorer  rtwdemo_sil_topmodel.c x
(Unknown Scope)
void rtwdemo_sil_topmodel_step(void)
{
    /* Logic: '<Root>/Logical Operator2' incorporates:
     * Inport: '<Root>/count_enable'
     * Inport: '<Root>/counter_mode'
     * Logic: '<Root>/Logical Operator'
     */
    enableA = ((!rtU.counter_mode) && rtU.count_enable);

    /* Outputs for Enabled SubSystem: '<Root>/CounterTypeA' */
    CounterTypeA();

    /* End of Outputs for SubSystem: '<Root>/CounterTypeA' */
    /* Logic: '<Root>/Logical Operator1' incorporates:
     * Inport: '<Root>/count_enable'
     * Inport: '<Root>/counter_mode'
     */
    enableB = (rtU.counter_mode && rtU.count_enable);

    /* Outputs for Enabled SubSystem: '<Root>/CounterTypeB' */
    CounterTypeB();

    /* End of Outputs for SubSystem: '<Root>/CounterTypeB' */
}

/* Model initialize function */
void rtwdemo_sil_topmodel_initialize(void)
{
}
```

You can now use the debugger features to observe code behavior. For example, you can step through code and examine variables.

To end the debugging session:

- 1 Remove all breakpoints.
- 2 Click the Continue button (**F5**).

The SIL simulation runs to completion and the Microsoft Visual Studio IDE closes automatically.

Note: In the Microsoft Visual Studio IDE, if you select **Debug > Stop Debugging**, the SIL simulation times out with the following error message:

The timeout of 1 seconds for receiving data from the rtiostream interface has been exceeded. There are multiple possible causes for this failure.

...
...

PIL Customization for Target Environment

In this section...

“Target Connectivity Configurations for PIL” on page 32-32

“Target Connectivity API Components” on page 32-32

“Communications `rtiostream` API” on page 32-33

Target Connectivity Configurations for PIL

Use target connectivity configurations and the target connectivity API to customize processor-in-the-loop (PIL) verification for target environments.

Through a target connectivity configuration, you specify:

- A target connectivity configuration name for a target connectivity API implementation.
- Settings that define the set of Simulink models that the configuration is compatible with, for example, the set of models that have a particular system target file, template makefile, and hardware implementation.

You must associate a connectivity configuration name with a connectivity API implementation. You can have many different connectivity configurations, each configuration being available for PIL simulation. Register a connectivity configuration with Simulink by creating an `sl_customization.m` file and placing it on the MATLAB search path.

To run a PIL verification, the software must first determine which of the available connectivity configurations to use. The software looks for a connectivity configuration that is compatible with the model under test. If the software finds multiple or no compatible connectivity configurations, the software generates an error message with information about resolving the problem.

For more information, see:

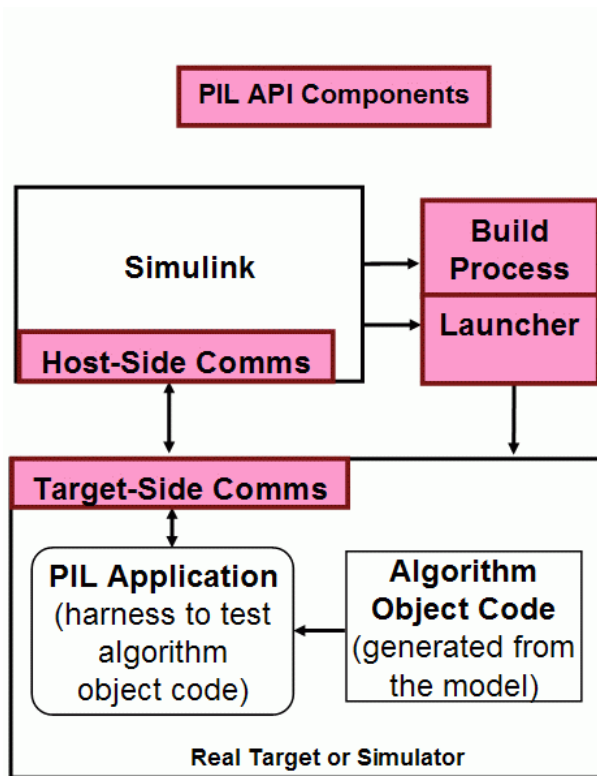
- “Target Connectivity API Components”
- “Create PIL Target Connectivity Configuration”

Target Connectivity API Components

Use the target connectivity API to integrate third-party tools for:

- Building the processor-in-the loop (PIL) application, an executable for the target hardware
- Downloading, starting, and stopping the application on the target
- Communicating between Simulink and the target

The following diagram shows the components of the target connectivity PIL API.



Communications `rtiostream` API

The `rtiostream` API supports communications for the target connectivity API. Use the `rtiostream` API to implement a communication channel that enables data exchange between different processes.

PIL verification requires a host-target communications channel. This communications channel comprises driver code that runs on the host and target. The `rtiostream` API

defines the signature of target-side and host-side functions that must be implemented by this driver code.

The API is independent of the physical layer that sends the data. Possible physical layers include RS232, Ethernet, or Controller Area Network (CAN).

A full `rtiostream` implementation requires both host-side and target-side drivers. Code generation software includes host-side drivers for the default TCP/IP implementation as well as a version for serial communications. To use:

- The TCP/IP `rtiostream` communications channel, you must provide, or obtain from a third party, target-specific TCP/IP device drivers.
- The serial communications channel, you must provide, or obtain from a third party, target-specific serial device drivers.

For other communication channels and platforms, the code generation software does not provide default implementations. You must provide both the host-side and the target-side drivers.

The `rtiostream` API comprises the following functions:

- `rtIOStreamOpen`
- `rtIOStreamSend`
- `rtIOStreamRecv`
- `rtIOStreamClose`

For information about:

- Using `rtiostream` functions in a connectivity implementation, see “Create a Connectivity API Implementation” on page 32-35.
- Testing the `rtiostream` shared library methods from MATLAB code, see `rtiostream_wrapper`.
- Debugging and verifying the behavior of custom `rtiostream` interface implementations, see “Test an `rtiostream` Driver” on page 32-36.

Create PIL Target Connectivity Configuration

In this section...

“Create a Connectivity API Implementation” on page 32-35

“Test an `rtiostream` Driver” on page 32-36

“Synchronize Host and Target” on page 32-38

“Specify Hardware Timer” on page 32-39

“Register a Connectivity API Implementation” on page 32-41

“Target Connectivity API Examples” on page 32-41

Create a Connectivity API Implementation

To create a target connectivity API implementation, you must create a subclass of `rtw.connectivity.Config`.

- You must instantiate `rtw.connectivity.MakefileBuilder`. This class configures the build process.
- You must create a subclass of `rtw.connectivity.Launcher`. This class downloads and executes the application using a third-party tool.
- Configure your `rtiostream` communications implementation:
 - On the target-side, integrate the driver code implementing `rtiostream` functions directly into the build process by creating a subclass of `rtw.pil.RtIOStreamApplicationFramework`.
 - On the host-side, compile the driver code into a shared library. You load and initialize this shared library by instantiating (or optionally, customizing) `rtw.connectivity.RtIOStreamHostCommunicator`.
- For code execution profiling, you must create a timer object that provides details of the hardware-specific timer and associated source files. See “Specify Hardware Timer”.

Note: Each time you modify a connectivity implementation, close and reopen the models to refresh them.

See also:

- “Creating Subclasses — Syntax and Techniques” in MATLAB documentation.
- `rtwdemo_custom_pil_script` for an example that helps you to create a target connectivity configuration using the target connectivity API

Test an `rtiostream` Driver

Use a test suite to debug and verify the behavior of custom `rtiostream` interface implementations.

The test suite has the following advantages:

- Reduces time for integrating custom hardware that does not have built-in `rtiostream` support.
- Reduces time for testing custom `rtiostream` drivers.
- Helps analyze the performance of custom `rtiostream` drivers.

The test suite has two parts. One part of the test suite runs on the target.

Note: After building the target application, download it to the target and run it.

To launch this part, compile and link the following files, which are in `matlabroot/toolbox/coder/rtiostream/src/rtiostreamtest`.

- `rtiostreamtest.c`
- `rtiostreamtest.h`
- `rtiostream.h` (located at `matlabroot/rtw/c/src/`)
- `rtiostream` implementation under investigation (for example, `rtiostream_tcpip.c`)
- `main.c`

To run the MATLAB part of the test suite, invoke `rtiostreamtest`. The syntax is as follows:

```
rtiostreamtest(connection,param1,param2)
```

- `connection` is a string indicating the communication method. It can have values `'tcp'` or `'serial'`.

- param1 and param2 have different values depending on the value of connection.
 - If connection is 'tcp', then param1 and param2 are hostname and port, respectively. For example, `rtiostreamtest('tcp', 'localhost', 2345)`.
 - If connection is 'serial', then param1 and param2 are COM port and baud rate, respectively. For example, `rtiostreamtest('serial', 'COM1', 9600)`.

You can run the MATLAB part of the test suite as follows:

```
rtiostreamtest('tcp','localhost','2345')
```

An output in the following format appears in the MATLAB window:

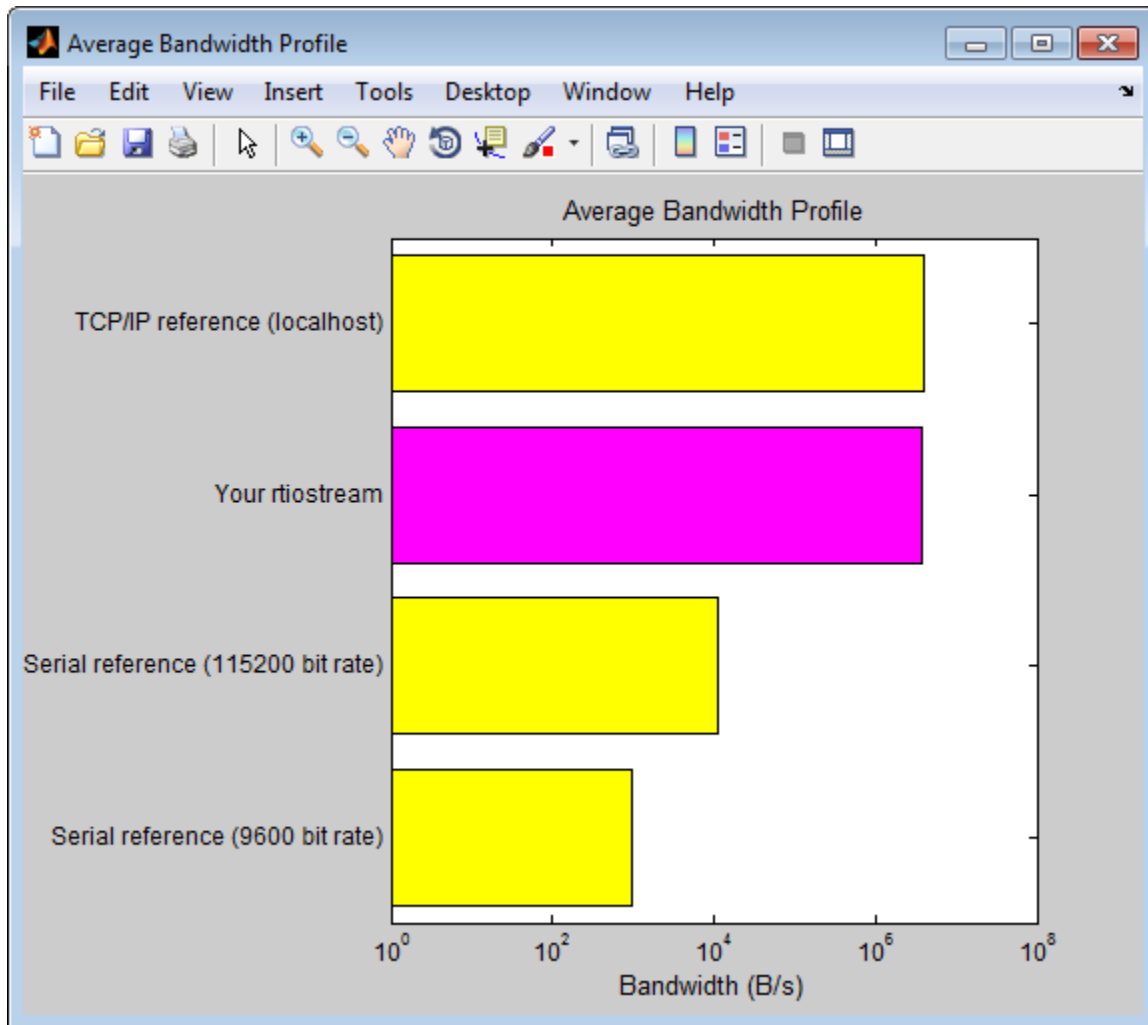
```
### Test suite for rtiostream ###
Initializing connection with target...

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

### rtiostream characteristics discovered
Round trip time  : 0.96689 ms
rtIOStreamRecv  behavior : non-blocking

### Test results
Test 1 (fixed size data exchange): ..... PASS
Test 2 (varying size data exchange): ..... PASS

### Test suite for rtiostream finished successfully ###
Furthermore, the following profile appears.
```



Synchronize Host and Target

If you use the `rtiostream` API to implement the communications channel, the host and target must be synchronized, which prevents Simulink from transmitting and receiving data before the target application is fully initialized.

To synchronize the host and target for TCP/IP `rtiostream` implementations, use the `setInitCommsTimeout` method from `rtw.connectivity.RtIOStreamHostCommunicator`. This approach works well for connection-oriented TCP/IP `rtiostream` implementations because Simulink automatically waits until the target server is running.

With other `rtiostream` implementations, for example, serial, the Simulink side of the `rtiostream` connection will open without waiting for the target to be fully initialized. In this case, you must make your `Launcher` implementation wait until the target application is fully initialized. Use one of the following approaches to synchronize your host and target:

- Add a pause at the end of the `Launcher` implementation that makes the `Launcher` wait until target initialization is complete.
- In the `Launcher` implementation, use third-party downloader or debugger APIs that wait until target initialization is complete.
- Implement a handshaking mechanism in the `Launcher / rtiostream` implementation to confirm that target initialization is complete.

Specify Hardware Timer

For code execution profiling, you must create a timer object that provides details of the hardware-specific timer and associated source files. You can use the Code Replacement Tool or the code replacement library API to specify this hardware-specific timer.

To specify the timer with the Code Replacement Tool:

- 1 Open the Code Replacement Tool. In the Command Window, enter `crtool`.
- 2 Create a new code replacement table. Select **File > New table**.
- 3 Create a new function entry. Under **Tables List**, right-click the new table. Then, from the context-menu, select **New entry > Function**.
- 4 In the middle view, select the new unnamed function.
- 5 On the **Mapping Information** pane:
 - a From the **Function** drop-down list, select `code_profile_read_timer`.
 - b Specify the count direction for your timer. For example, from the **Count direction** drop-down list, select **Up**.
 - c In the **Ticks per second** field, specify the number of ticks per second for your timer, for example, `1e+09`.

The default value is 0. In this case, the software reports time measurements in terms of ticks, not seconds.

- d In the **Name** field, specify a replacement function name, for example, MyTimer.
- e Click **Apply**.

- f To validate the function entry, click **Validate entry**.
- 6 On the **Build Information** pane, specify the required build information. See “Specify Build Information for Replacement Code”.
 - 7 Save the table (**Ctrl+S**). When you save the table for the first time, use the Save As dialog box to specify the file name and location.

You must save the table in a location that is on the MATLAB search path. For example, you can save this file in the folder for your subclass of `rtw.connectivity.Config`.

The software stores your timer information as a code replacement library table.

- 8 Assuming you save the table as *MyCr1Table.m*, in your subclass of `rtw.connectivity.Config`, add the following line:

```
setTimer(this, MyCr1Table)
```

For more information, see “What Is Code Replacement?” and “What Is Code Replacement Customization?”.

Register a Connectivity API Implementation

Register the new connectivity API implementation to Simulink as a connectivity configuration, by creating or adding to an `sl_customization.m` file. By doing so, you also define the set of Simulink models that the new connectivity configuration is compatible with.

For details, see `rtw.connectivity.ConfigRegistry`.

Target Connectivity API Examples

For step-by-step examples, see:

- `rtwdemo_custom_pil_script`

This example shows you how to create a custom PIL implementation using the target connectivity APIs. You can examine the code that configures the build process to support PIL, a downloading and execution tool, and a communication channel between host and target. Follow the steps in the example to activate a full host-based PIL configuration.

- `rtwdemo_rtiostream_script`

This example shows you how to implement a communication channel for use with the Embedded Coder product and your embedded target. This communication channel enables exchange of data between different processes. PIL simulation requires exchange of data between the Simulink software running on your host computer and deployed code executing on target hardware.

The `rtiostream` interface provides a generic communication channel that you can implement in the form of target connectivity drivers for a range of connection types.

The example shows how to configure your own target-side driver for TCP/IP, to operate with the default host-side TCP/IP driver. The default TCP/IP communications allow high-bandwidth communication between host and target, which you can use for transferring data such as video.

Note: If you customize the `rtiostream` TCP/IP implementation for your PIL simulations, you must turn off Nagle's algorithm for the server side of the connection. If Nagle's algorithm is not turned off, your PIL simulations might run at a significantly slower speed. The `matlabroot/rtw/c/src/rtiostream/rtiostreamtcpip/rtiostream_tcpip.c` file shows how you can turn off Nagle's algorithm:

```
/* Disable Nagle's Algorithm*/
option = 1;
sockStatus = setsockopt(lFd, IPPROTO_TCP, TCP_NODELAY, (char*)&option, sizeof(option));
```

For your custom TCP/IP implementation, you might have to modify this code.

The example also shows how to implement custom target connectivity drivers, for example, using serial, CAN, or USB for both host and target sides of the communication channel.

View Test Harness in Code Generation Report

With top-model and Model block SIL and PIL simulations, you can produce a code generation report and static code metrics that cover SIL and PIL test harness files. The information helps you to:

- Understand and review the SIL and PIL verification process.
- See how your registered custom target connectivity files fit into the target application that runs during a SIL or PIL simulation.

This feature is not supported for simulations that you run with the PIL block.

To configure the creation of a code generation report and static code metrics, on the **Configuration Parameters > Code Generation > Report** pane, select the **Create code generation report** and **Static code metrics** check boxes. Then click **OK**. For more information about:

- Code generation reports, see “HTML Code Generation Report Extensions”
- Static code metrics, see “Static Code Metrics”

At the end of the simulation, the software displays test harness files and the corresponding static code metrics in the code generation report.

The screenshot shows the 'Code Generation Report' window. On the left, the 'Contents' pane lists various reports, with 'Static Code Metrics Report' highlighted in yellow and circled in red. A red arrow points from this report to the main content area. The main content area is titled '3. Function Information [hide]' and contains a table of function metrics. The table has five columns: 'Function Name', 'Accumulated Stack Size (bytes)', 'Self Stack Size (bytes)', 'Lines of Code', and 'Lines'. The table lists various functions, including 'main', 'xilInit', 'xilTerminateComms', 'xilRun', 'processData', 'finalizeCommandResponse', 'xilReadData', 'xilInitialize', 'xilInitializeConditions', 'xilGetDataTypeInfo', 'xilProcessParams', 'xilOutput', 'rtwdemo_sil_topmodel_step', 'CounterTypeA', 'CounterTypeB', 'xilTerminate', 'xilEnable', 'xilDisable', and 'xilUpdate'.

Function Name	Accumulated Stack Size (bytes)	Self Stack Size (bytes)	Lines of Code	Lines
[-] main	1,552	12	22	32
[+] xilInit	1,540	4	7	10
[+] xilTerminateComms	540	8	7	9
[-] xilRun	430	29	191	222
[+] processData	401	9	49	61
[+] finalizeCommandResponse	393	1	14	21
[+] xilReadData	384	24	22	28
[+] xilInitialize	4	4	8	12
xilInitializeConditions	4	4	7	11
xilGetDataTypeInfo	0	0	1	3
xilProcessParams	0	0	4	7
[-] xilOutput	0	0	11	14
[-] rtwdemo_sil_topmodel_step	0	0	5	26
CounterTypeA	0	0	11	43
CounterTypeB	0	0	8	31
xilTerminate	0	0	4	8
xilEnable	0	0	5	9
xilDisable	0	0	5	9
xilUpdate	0	0	5	9

The software displays the test harness files in the **Interface files** category.

Note: You must not use files from the SIL/PIL test harness in code development as these files can change over releases. Use supplied APIs for code development.

SIL and PIL Simulation Support and Limitations

In this section...

“About SIL and PIL Simulation Support and Limitations” on page 32-45

“Code Source Support” on page 32-45

“Block Support” on page 32-48

“Configuration Parameters Support” on page 32-50

“I/O Support” on page 32-56

“Hardware Implementation Support” on page 32-66

“Other Feature Support” on page 32-67

About SIL and PIL Simulation Support and Limitations

Top-model and Model block software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulation modes, and SIL and PIL blocks are Embedded Coder features.

The following tables summarize the support provided for top-model SIL and PIL, Model block SIL and PIL and the SIL or PIL block. “Yes” indicates a supported feature.

Information on selected aspects of SIL and PIL is also provided, especially unsupported features and limitations.

Code Source Support

Code Source	Code Interface	Top-Model SIL/PIL	Model Block SIL/PIL	SIL/PIL Block
Top model	Standalone	Yes	Yes. See “Top-Model Code Testing with Model Block SIL/PIL” on page 32-46.	Yes
Atomic subsystem	Standalone	No	No	Yes
Virtual subsystem	Standalone	No	No	Yes, but recommend atomic subsystem. See “Algebraic Loop

Code Source	Code Interface	Top-Model SIL/PIL	Model Block SIL/PIL	SIL/PIL Block
				Issues” on page 32-51.
Model block	Model reference target	No, but you can include Model blocks inside your top model.	Yes. See “Conditionally Executed Subsystem” on page 32-47.	No, but you can include Model blocks inside your model.
Enabled/ Triggered subsystem	Standalone	No	No	Yes
Exported function-call subsystem	Export Functions	N/A	N/A	Yes
Export-function model	Export Functions	Yes	Yes	N/A
Legacy code	Custom	See “Custom Code Interfaces” on page 32-47.	See “Custom Code Interfaces” on page 32-47.	See “Custom Code Interfaces” on page 32-47.
MATLAB Coder	MATLAB Coder	See “Custom Code Interfaces” on page 32-47.	See “Custom Code Interfaces” on page 32-47.	See “Custom Code Interfaces” on page 32-47.

For more information, see “Code Interfaces for SIL and PIL” on page 32-23.

Top-Model Code Testing with Model Block SIL/PIL

The following limitations apply:

- The `Model Variants` block does not support the block parameter `CodeUnderTest`. The software behaves as if `CodeUnderTest` is set to `'Model reference'`. To work around this limitation, use the `Variant Subsystem` block. Through this block, you can incorporate `Model` blocks for which `CodeUnderTest` is set to `'Top model'`.
- Because model arguments do not apply to a top model, when the **Code under test** block parameter is set to `Top model`, the software does not support the **Model arguments** block parameter.
- Conditional execution does not apply to a top model. If a `Model` block is set up to execute conditionally and the **Code under test** block parameter is set to `'Top model'`, the software produces an error when you run a SIL or PIL simulation.

- For sample time independent models, you must set **Configuration Parameters > Solver > Periodic sample time constraint** to **Ensure sample time independent**.
- Simulation results from top-model code and model reference code might differ when a root-level Inport is connected to a root-level Outport by a signal that has a signal object with an initial value.

For top-model code, the software associates the signal object with the Inport. The software might apply the initial value for the signal object to the Inport. See “Initialization Behavior Summary for Signal Objects”.

For model reference code, the software associates the signal object with the Outport. The software does not apply the initial value for the signal object to the Inport.

Conditionally Executed Subsystem

You see an error if:

- You place your Model block (in either SIL or PIL simulation mode) in a conditionally executed subsystem and the referenced model is multirate (that is, has multiple sample times). Single rate referenced models (with only a single sample time) are not affected.
- Your Model block (in either SIL or PIL simulation mode) has blocks that depend on absolute time **and** is conditionally executed.

Custom Code Interfaces

MathWorks does not provide direct SIL/PIL support for code interfaces such as legacy code and MATLAB Coder. However, you can incorporate these interfaces into Simulink as an S-function (for example, using the Legacy Code Tool, S-Function Builder, or handwritten code), and then verify them using SIL/PIL.

SIL/PIL Does Not Check Simulink Coder Error Status

SIL/PIL does not check the Simulink Coder error status of the generated code under test. This error status flags exceptional conditions during execution of the generated code.

The Simulink Coder error status can also be set by blocks in the model (for example, custom blocks developed by a user). It is a limitation that SIL/PIL cannot check this error status and report back errors.

Block Support

Blocks Supported Within SIL/PIL Component	Top-Model SIL/PIL	Model Block SIL/PIL	SIL/PIL Block
Driver blocks	Yes, but not recommended.	Yes, but not recommended.	Yes, but not recommended.
Function Caller block	Yes	Yes	No. Use the Model block SIL/PIL approach, with the Code under test block parameter set to Top model .
MATLAB Function block	Yes	Yes	Yes
Model block	Yes	Yes	Yes
Merge blocks	Yes	Yes. Cannot connect SIL/PIL outputs to Merge blocks. See “Merge Block Issue” on page 32-49.	Yes. Cannot connect SIL/PIL outputs to Merge blocks. See “Merge Block Issue” on page 32-49.
Scope blocks, and all types of run-time display. For example, display of port values and signal values.	No	No	No
Simulink Function block	Yes	Yes	No. Use the Model block SIL/PIL approach, with the Code under test block parameter set to Top model .
Stop blocks	No. SIL/PIL ignores the Stop Simulation	No. SIL/PIL ignores the Stop Simulation block and continues simulating.	No. SIL/PIL ignores the Stop Simulation

Blocks Supported Within SIL/PIL Component	Top-Model SIL/PIL	Model Block SIL/PIL	SIL/PIL Block
	block and continues simulating.		block and continues simulating.
To File blocks	Yes, if MAT-file logging is on. MAT-file logging might not be available in PIL mode.	Yes, if MAT-file logging is on. MAT-file logging might not be available in PIL mode.	Yes, if MAT-file logging is enabled. For PIL block, MAT-file logging must be supported.
To Workspace blocks	Yes, if MAT-file logging is on. MAT-file logging might not be available in PIL mode.	No, MAT-file logging is not supported.	Yes, if MAT-file logging is supported and on.

Merge Block Issue

If you connect SIL/PIL outputs to a Merge block, you see an error because S-function memory is not reusable.

Model in Compiled State During Top-Model SIL/PIL

During a top-model SIL/PIL simulation, the software places the model in a compiled state – see `model`. This action might result in a conflict over global resources between the model and the generated SIL/PIL code. In this case, you might see differences between Normal mode and SIL/PIL simulation outputs.

For example, you might see this limitation with a model that uses UDP blocks from the DSP System Toolbox. These blocks open UDP sockets, which can lead to resource contention between the model and the generated SIL/PIL code.

Other Top-Model SIL/PIL Limitations

SIL/PIL does not support the callbacks (model or block) `StartFcn` and `StopFcn`.

Note: Top-model SIL/PIL supports the callback `InitFcn`.

Configuration Parameters Support

Configuration Parameters	Top-Model SIL/PIL	Model Block SIL/PIL	SIL/PIL Block
ERT-based system target file	Yes	Yes	Yes
AUTOSAR system target file	Yes. See “Use the SIL and PIL Simulation Modes”.	Yes. See “Use the SIL and PIL Simulation Modes”.	Yes. See “Use a SIL or PIL Block for AUTOSAR Verification”.
GRT-based system target file	No	No	No
Classic call interface	No; see “Missing Code Interface Description File Errors” on page 32-51.	No; see “Missing Code Interface Description File Errors” on page 32-51.	No; see “Missing Code Interface Description File Errors” on page 32-51.
Function Prototype Control	Yes	Yes	Yes
Reusable code format	Yes, but see the special cases in “Imported Data Definitions” on page 32-62.	N/A	Yes, but see the special cases in “Imported Data Definitions” on page 32-62.
Standard math library	Yes	Yes	Yes
Code replacement library	Yes	Yes	Yes
C++	Yes	Yes	Yes
C++ class	Yes. See “C++ Class Code and SIL/PIL” on page 32-56.	Yes. See “C++ Class Code and SIL/PIL” on page 32-56.	Yes. See “C++ Class Code and SIL/PIL” on page 32-56.
Generate ASAP2 file	Yes	Yes	Yes
Generate example main	N/A	N/A	N/A
MAT-file logging	Yes. For PIL, the target environment might not support MAT-file logging.	Yes. For PIL, the target environment might	Yes. For PIL, the target environment might

Configuration Parameters	Top-Model SIL/PIL	Model Block SIL/PIL	SIL/PIL Block
		not support MAT-file logging.	not support MAT-file logging.
Signal logging	Yes, for internal signals and for signals connected to root-level inports and outputs. See “Internal Signal Logging Support”, “Top-Model Root-Level Logging Limitations”.	Yes. See “Internal Signal Logging Support”.	No, but see “Verify Internal Signals of a Component” on page 32-20.
'Simplified' model initialization	Yes	Yes	Yes
Single output/update	Yes, but see “Algebraic Loop Issues” on page 32-51.	Yes, but see “Algebraic Loop Issues” on page 32-51.	Yes, but see “Algebraic Loop Issues” on page 32-51.
Configuration set reference	Yes	Yes	Yes

- “Missing Code Interface Description File Errors” on page 32-51
- “Algebraic Loop Issues” on page 32-51
- “Internal Signal Logging Support” on page 32-53
- “Top-Model Root-Level Logging Limitations” on page 32-54
- “C++ Class Code and SIL/PIL” on page 32-56

Missing Code Interface Description File Errors

SIL/PIL requires a code interface description file, which is generated during the code generation process for the component under test. If the code interface description file is missing, the SIL/PIL simulation cannot proceed and you see an error reporting that the file does not exist. This error can occur if you select the unsupported option **Classic call interface** in your configuration parameters. Do not select this option.

Algebraic Loop Issues

For more information on algebraic loops, see:

- “Algebraic Loops” in the Simulink documentation.
- The Algebraic Loops section in “Modeling Semantic Considerations”.
- “Code Generation of Subsystems”.

There are three ways that PIL simulation can introduce algebraic loops that do not exist for a normal simulation:

- “Algebraic Loops Caused by Code Generation for a Virtual Subsystem” on page 32-52
- “Algebraic Loops Caused by “Single output/update function”” on page 32-52
- “Algebraic Loops Caused by SIL/PIL Scheduling Limitations” on page 32-52

Algebraic Loops Caused by Code Generation for a Virtual Subsystem

If you generate code for a virtual subsystem, code generation treats the subsystem as atomic and generates the code accordingly. The resulting code can change the execution behavior of your model, for example, by applying algebraic loops, and introduce inconsistencies to the simulation behavior.

Declare virtual subsystems as atomic subsystems to enable consistent simulation and execution behavior for your model. For more information, see “Code Generation of Subsystems”

Algebraic Loops Caused by “Single output/update function”

The “single output/update function” in code generation optimization can introduce algebraic loops because it introduces direct feedthrough via a combined output and update function.

This option is not compatible with the **Minimize algebraic loop occurrences** option (in the Subsystem Parameters dialog box and **Model Referencing** pane of the Configuration Parameters dialog box). This option allows code generation to remove algebraic loops by partitioning generated code between output and update functions to avoid direct feedthrough.

Algebraic Loops Caused by SIL/PIL Scheduling Limitations

The S-function scheduling mechanism that the software uses to execute the SIL/PIL component has the following limitations:

- Direct feedthrough is set to true.

- Separate output and update functions in the SIL/PIL component are executed from the `mdlOutputs` S-function callback.

These limitations mean that SIL/PIL can introduce algebraic loops that do not exist in normal simulation, and you might get incorrect results. If this happens, you see a warning or error about the introduced algebraic loop and SIL/PIL results can differ from simulation results. You do not see a warning or error if **Configuration Parameters > Diagnostics > Algebraic loop** is set to none.

A workaround is to break the algebraic loop by inserting a Unit Delay block so that the algebraic loop does not occur. You can then use SIL/PIL.

Internal Signal Logging Support

You can use Simulink signal logging with the SIL and PIL simulation modes, with both top-model SIL or PIL and Model block (referenced model) SIL or PIL. This feature allows you to:

- Collect signal logging outputs (e.g., `log sout`) during SIL and PIL simulations.
- Log the internal signals and the root-level outputs of a SIL or PIL component.
- Manage the SIL and PIL signal logging settings using the Simulink Signal Logging Selector.
- Compare logged signals between normal, SIL, and PIL simulations, for example, using the Simulation Data Inspector.

SIL or PIL signal logging requires the following model configuration settings:

- On the **Data Import/Export** pane of the Configuration Parameters dialog box, set **Signal logging format** to **Dataset**.
- On the **Code Generation > Interface** pane of the Configuration Parameters dialog box, set **Interface** to **C API**.

The C API is used to determine the addresses of the internal signals that need to be logged. See also “Internal Signal Logging Limitations” on page 32-53.

Internal Signal Logging Limitations

- The C API requires that support for floating-point numbers is selected (see **Configuration Parameters > Code Generation > Interface > Support > floating-point numbers**).

- Only signals that are included in the C API are logged during SIL/PIL simulation. You might need to configure signals as test points (see **Signal Properties > Test point**) to check that they are observable in the generated code.
- Logging of signals in models referenced by the SIL/PIL component is not supported. Only signals within the top level of the SIL/PIL component are logged.
- Virtual signals (e.g. MUX) are not supported.
- Buses are not supported.
- Custom storage classes are not supported.
- Continuous, asynchronous and triggered sample times are not supported.
- Logging of Stateflow States and Local Data is not supported.

With top-model internal signal logging, some additional limitations apply:

- Variable-size signals, Function-call signals, and Action signals: error for normal simulation and warning for SIL/PIL.
- State port signals: error for normal simulation; no warning for SIL/PIL.
- Signals feeding merge blocks are not supported for logging in normal simulation but are logged in SIL/PIL mode. The logged values during SIL/PIL will be the same as the logged values for the output of the merge block.
- Under the following circumstances, top-model Normal simulation logs data at a periodic rate, but top-model SIL/PIL simulation logs data at the constant rate:
 - **Inline parameters** is not selected.
 - A constant sample time signal from a Model block is logged in the top model.
 - The logged signal is not directly connected to a root-level output port.

To avoid this behavior, select **Inline parameters** to log at constant rate in all simulation modes.

Top-Model Root-Level Logging Limitations

Top-model SIL/PIL supports signal logging for signals connected to root-level inports and outports. Both `ModelDataLogs` and `Dataset` signal logging formats are supported, and the C API is not required. Root-level logging has the following limitations:

- The characteristics of the logged data such as data type and dimensions match the characteristics of the root-level inports and outports rather than the characteristics of the connected signal.

In some cases, there can be differences in data type and dimensions between the signal being logged and the root inport or outport that the signal is connected to. Consider the following examples.

- If a signal being logged has matrix dimensions [1x5] but the outport connected to the signal has vector dimensions (5), then the data logged during a SIL or PIL simulation has vector dimensions (5).
- If a signal being logged has scalar dimensions but the outport connected to the signal has matrix dimensions [1x1], then the data logged during a SIL or PIL simulation has matrix dimensions [1x1].
- Signals connected to duplicated inports are not logged during SIL/PIL simulation. No warning is issued.

During normal simulation, signals connected directly to duplicated inports are logged.

- The Signal Logging Selector / `DataLoggingOverride` override mechanism is not supported.
- There is no support for signals that are not named.
- The Normal and SIL/PIL simulations for a top model produce logged bus signals with names that are different when all of the following conditions apply:
 - The signal logging format is `Dataset`.
 - The names of the elements in the bus signal are different from the corresponding names in the bus object.
- The software inserts the suffix, `_wrapper`, in the following cases:
 - For *signal logging*, if you specify the signal logging format to be `ModelDataLogs`, the software adds `_wrapper` to the block path for signals in `logstdout`. For example:

```
>> logstdout.SignalLogging
      Name: 'SignalLogging'
      BlockPath: 'sillogging_wrapper/sillogging'
      PortIndex: 1
      SignalName: 'SignalLogging'
      ParentName: 'SignalLogging'
      TimeInfo: [1x1 Simulink.TimeInfo]
      Time: [11x1 double]
      Data: [11x1 double]
```

If the Simulation Data Inspector is recording data, the software adds `_wrapper` to the run name.

To avoid this behavior, use the `Dataset` signal logging format. See “`Simulink.SimulationData.Dataset`”.

- For *output logging*, if the save format is `Structure` or `Structure with time` and you run the `sim` command without specifying the single-output format, the software adds `_wrapper` to the block name for signals in `yout`. For example:

```
>> yout.signals
```

```
ans =
      values: [11x1 double]
 dimensions: 1
      label: 'SignalLogging'
 blockName: 'sillogging_wrapper/OutputLogging'
```

If the save format is `Array`, the software does not add the suffix.

To avoid this behavior, run command-line simulations with the `sim` command specifying the single-output format. See “Run Simulation Using the `sim` Command”.

C++ Class Code and SIL/PIL

To run SIL/PIL simulations with C++ class code, you must set **External I/O access** to `None`.

I/O Support

I/O	Top-Model SIL/PIL	Model Block SIL/PIL	SIL/PIL Block
Tunable parameters (Model reference arguments)	N/A	Yes. See “Tunable Parameters and SIL/PIL” on page 32-59.	N/A
Tunable parameters (Workspace variables)	No	Yes. See “Tunable Parameters and SIL/PIL” on page 32-59.	Yes. See “Tunable Parameters and SIL/PIL” on page 32-59.
Virtual buses	No	Yes	Yes
Nonvirtual buses	Yes, but see “Top-Model SIL/PIL Bus	Yes	Yes

I/O	Top-Model SIL/PIL	Model Block SIL/PIL	SIL/PIL Block
	Limitations” on page 32-65.		
MUX/DEMUX	No	Yes	Yes, but see “PIL Block MUX Support Limitations” on page 32-65.
Vector/2D/ Multidimensional	Yes	Yes	Yes
Complex data	Yes	Yes	Yes
Fixed-point data	Yes. See “Fixed-Point Data Types Wider Than 32 Bits” on page 32-61.	Yes. See “Fixed-Point Data Types Wider Than 32 Bits” on page 32-61.	Yes. See “Fixed-Point Data Types Wider Than 32 Bits” on page 32-61.
Multiword fixed-point data	No. See “Tunable Parameters and SIL/PIL” on page 32-59.	No. See “Tunable Parameters and SIL/PIL” on page 32-59.	No. See “Tunable Parameters and SIL/PIL” on page 32-59.
Complex fixed-point data	Yes	Yes	Yes
Fixed-point data type override	Yes	Yes	Yes
Data type replacement	Yes, but see “Data Type Replacement Limitation” on page 32-65	Yes, but see “Data Type Replacement Limitation” on page 32-65	Yes, but see “Data Type Replacement Limitation” on page 32-65
Goto/From I/O	N/A	N/A	Yes
Global data store I/O	Yes. See “Global Data Store Support” on page 32-62 and “Imported Data Definitions” on page 32-62.	Yes. See “Global Data Store Support” on page 32-62 and “Imported Data Definitions” on page 32-62.	Yes. See “Global Data Store Support” on page 32-62 and “Imported Data Definitions” on page 32-62.

I/O	Top-Model SIL/PIL	Model Block SIL/PIL	SIL/PIL Block
Local data store I/O	No. See “Imported Data Definitions” on page 32-62.	No. See “Imported Data Definitions” on page 32-62.	No. See “Imported Data Definitions” on page 32-62.
Non-port-based sample times	Yes	Yes	Yes
Continuous sample times	Not at SIL or PIL component boundary.	No	Not at SIL or PIL component boundary.
Outputs with constant sample time	Yes	No	Yes
Non-auto-storage classes for data (such as signals, parameters, data stores)	Yes. See “Imported Data Definitions” on page 32-62.	Yes. See “Imported Data Definitions” on page 32-62.	Yes. See “Imported Data Definitions” on page 32-62.
Simulink data objects	Yes	Yes	Yes
Simulink numeric type and alias type	Yes	Yes	Yes
Simulink enumerated data	Yes	Yes	Yes
Custom storage classes	Yes, but see “Imported Data Definitions” on page 32-62, and “GetSet Custom Storage Class” on page 32-63.	Yes, but see “Imported Data Definitions” on page 32-62, and “GetSet Custom Storage Class” on page 32-63.	Yes, but see “Imported Data Definitions” on page 32-62, and “GetSet Custom Storage Class” on page 32-63.

I/O	Top-Model SIL/PIL	Model Block SIL/PIL	SIL/PIL Block
Variable-size signals	No. See “Variable-Size Signals and SIL/PIL” on page 32-64.	Yes. On the Configuration Parameters > Model Referencing pane, in the Propagate sizes of variable-size signals field, specify During execution . Otherwise, software generates error.	No. See “Variable-Size Signals and SIL/PIL” on page 32-64.
Noninlined S-functions	Yes	No	Yes

- “Tunable Parameters and SIL/PIL” on page 32-59
- “Fixed-Point Data Types Wider Than 32 Bits” on page 32-61
- “Global Data Store Support” on page 32-62
- “Imported Data Definitions” on page 32-62
- “GetSet Custom Storage Class” on page 32-63
- “Unsupported Implementation Errors” on page 32-63
- “Variable-Size Signals and SIL/PIL” on page 32-64
- “Data Type Overrides Unavailable for Most Blocks in Embedded Targets and Desktop Targets” on page 32-64
- “Data Type Replacement Limitation” on page 32-65
- “Top-Model SIL/PIL Bus Limitations” on page 32-65
- “PIL Block MUX Support Limitations” on page 32-65
- “Incremental Build for Top-Model SIL/PIL” on page 32-65
- “Exported Functions in Feedback Loops” on page 32-65

Tunable Parameters and SIL/PIL

You can tune parameters during a SIL/PIL mode simulation the same way that you tune parameters during a Normal mode simulation.

For more information, see “Global Tunable Parameters” and “Using Model Arguments” in the Simulink documentation.

The following table summarizes SIL/PIL support for tunable workspace parameters.

Action	Top-Model SIL/PIL	Model Block SIL/PIL	SIL/PIL Block
Define and initialize parameters	Yes. See "Limitations" on page 32-60.	Yes. See "Limitations" on page 32-60.	Yes. See "Limitations" on page 32-60.
Tune parameters	Yes. See "Limitations" on page 32-60.	Yes. See "Limitations" on page 32-60.	Yes. See "Limitations" on page 32-60.

Limitations

During a SIL/PIL simulation, the software cannot define, initialize, or tune the following types of tunable workspace parameters. The software produces warnings or errors.

Parameter description	Software response for		
	Top-Model SIL/PIL	Model Block SIL/PIL	SIL/PIL Block
Parameters with storage class that applies "static" scope or "const" keyword. For example, Custom, Const, or ConstVolatile	Warning	Warning	Warning
Parameters with multiword fixed-point data types	Warning	Error	Warning
Parameters with data types that have different sizes on host and target	Warning	Error	Warning

During a SIL/PIL block simulation, the software supports the tuning of *tunable workspace* parameters but not *tunable block dialog* parameters. You can view the list of tunable workspace parameters through the Function Block Parameters dialog box of the SIL/PIL block.

For C++ class code, SIL/PIL tunable workspace parameters can be tuned provided you do not use the following settings together:

- **Block parameter visibility:** private

- **Block parameter access** : Method or Inlined method

For top-model SIL/PIL and the SIL/PIL block, consider the case where all of the following conditions apply:

- **Code Generation > Interface > Code interface packaging** is set to Reusable function.
- **Optimization > Signals and Parameters > Inline parameters** is not selected.
- The model contains parameters with storage class **Auto** or **SimulinkGlobal**.

If the SIL/PIL component cannot dynamically initialize tunable parameters in the rtP model parameter structure, the software produces an error message like the following:

```
Parameter Dialog:InitialOutput in 'rtwdemo_sil_topmodel/CounterTypeA/count'
is part of the imported "rtP" structure in the generated code but cannot be
initialized by SIL or PIL. To avoid this error, make sure the parameter
corresponds to a tunable base workspace variable with a storage class such
as SimulinkGlobal and is supported for dynamic parameter initialization /
tuning with SIL/PIL.
```

Fixed-Point Data Types Wider Than 32 Bits

SIL/PIL supports fixed-point data types that are wider than 32 bits. For example:

- 64-bit long and long long
- 64-bit execution profiling timer data type
- int64 and uint64, which are used in MATLAB Coder SIL execution.

The following constraints apply:

- For 64-bit data type support, the data type must be representable as long or long long on the MATLAB host *and* the target. Otherwise, the software uses the multiword fixed-point approach, which SIL/PIL does not support.
- 32-bit Windows does not support 64-bit long or long long data types. In this case, the software uses the multiword fixed-point approach, which SIL/PIL does not support.
- The software does not support the 40-bit long data type of the TI's C6000™ target.

Through the **Configuration > Hardware Implementation** pane, you can enable support for the 64-bit long long data type. However, for data types with widths between 33 and 40 bits (inclusive), the software implements the data types using the 40-bit long data type, which SIL/PIL does not support.

Global Data Store Support

SIL/PIL supports global data stores. However, with components that are not export-function models, top-model SIL/PIL and SIL/PIL block simulations that access global data stores must be single rate. Otherwise, the software produces an error.

Imported Data Definitions

You can use, for example, signals, parameters, and data stores that specify storage classes with imported data definitions.

Top-Model SIL/PIL and SIL/PIL Block

Top-model SIL/PIL and the SIL/PIL block do not define imported signals that are internal with respect to the component or models referenced by the component.

Top-model SIL/PIL and the SIL/PIL block automatically define storage for imported data associated with:

- Signals at the root level of the component (on the I/O boundary)
- Global data stores
- Parameters

Top-model SIL/PIL and the SIL/PIL block do not define storage for other imported data. You must define the storage through custom code included by the component under test or through the PIL `rtw.pil.RtIOStreamApplicationFramework` API. For example, the SIL/PIL application does not define storage for imported data associated with:

- Internal signals (not on the I/O boundary)
- Local data stores

Model Block SIL/PIL

The SIL/PIL application automatically defines storage for imported data associated with:

- Signals at the root level of the component (on the I/O boundary)
- Parameters. See Tunable Parameters and SIL/PIL Limitations.
- Global data stores

Note: Model block SIL/PIL does not define imported signals that are internal with respect to the component or models referenced by the component.

A limitation is that SIL/PIL does not define storage for other imported data storage. You must define the storage through custom code included by the component under test or through the PIL `rtw.pil.RtIOStreamApplicationFramework` API. For example, the PIL application does not define imported data storage for data associated with:

- Internal signals (not on the I/O boundary)
- Local data stores

GetSet Custom Storage Class

The software supports the `GetSet` custom storage class for all types of SIL and PIL simulations. The SIL/PIL test harness automatically provides C definitions of the `Get` and `Set` functions that are used during SIL/PIL simulations. In addition, the software supports only *scalar* signals, parameters and global data stores.

Unsupported Implementation Errors

If you use a data store, signal, or parameter implementation that SIL/PIL does not support, you might see errors like the following:

The following *data interfaces* have implementations that are not supported by SIL or PIL.

data interfaces can be global data stores, inports, outports or parameters.

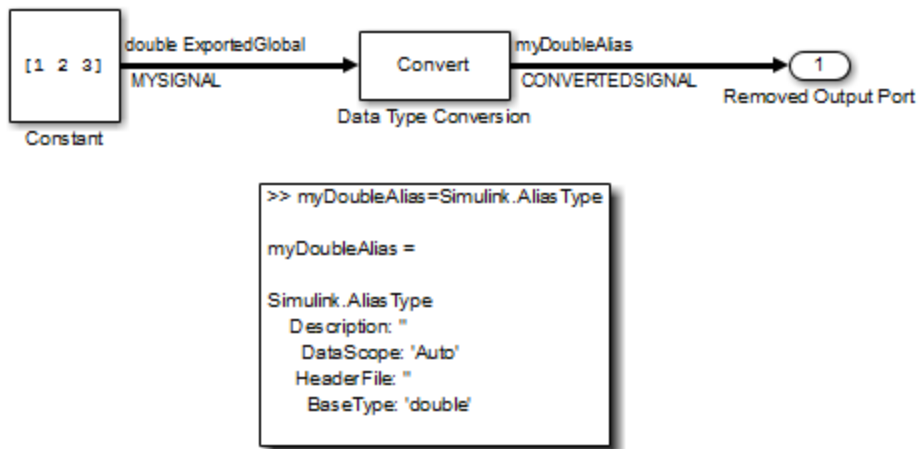
You see this error message because the model output port has been optimized through virtual output port optimization. See “Virtualized Output Ports Optimization”. The error occurs because the properties (for example, data type, dimensions) of the signal or signals entering the virtual root output port have been modified by routing the signals in one of the following ways:

- Through a Mux block
- Through a block that changes the signal data type. To check the consistency of data types in the model, display Port Data Types by selecting **Display > Signals & Ports > Port Data Types**.
- Through a block that changes the signal dimensions. To check the consistency of data types in the model, display dimensions by selecting **Display > Signal & Ports > Signal Dimensions**.

Note: Dimension changes from scalar (1) to matrix [1x1], and, matrix [1x1] to scalar (1), can lead to this error. Furthermore, it is difficult to inspect the model for

such changes because the **Display > Signal & Ports > Signal Dimensions** feature does not distinguish between (1) and [1x1] dimensions. The software shows both signals as scalar signals. Check your model and workspace objects carefully and see that scalar dimensions are specified consistently.

The following model causes this error by changing the output port signal data type.



Variable-Size Signals and SIL/PIL

SIL/PIL treats variable-size signals at the I/O boundary of the SIL/PIL component as fixed-size signals, which can lead to errors during propagation of signal sizes. To avoid such errors, use only fixed-size signals at the I/O boundary of the SIL/PIL component.

There might be cases where no error occurs during propagation of signal sizes. In these cases, the software treats variable-size input signals as zero-size signals.

Data Type Overrides Unavailable for Most Blocks in Embedded Targets and Desktop Targets

When you attempt to perform a datatype override on a block, you might get an error message similar to the following example:

```
Error reported by S-function 'sfun_can_frame_splitter' in
'c2000_host_CAN_monitor/CAN Message Unpacking/CAN Message
Unpacking': Incompatible DataType or Size specified.
```

Fixed-Point Tool data type overrides are not available for blocks in **Simulink Coder > Desktop Targets** and **Embedded Coder > Embedded Targets** libraries that support fixed-point data types.

There is no resolution for this issue.

Data Type Replacement Limitation

The software does not support replacement data type names that you define for the built-in data type `boolean` if these names map to either the `int` or `uint` built-in data type.

Top-Model SIL/PIL Bus Limitations

The software does not support grounded or unconnected signals at the outputs of a top model.

For top-model SIL/PIL, set the **Simulation > Configuration Parameters > Diagnostics > Connectivity+Mux blocks used to create bus signals** parameter to error. For details, see “Prevent Bus and Mux Mixtures”.

PIL Block MUX Support Limitations

The PIL block supports mux signals, except mixed data-type mux signals that expand into individual signals during a right-click subsystem build. You see an error for unsupported cases.

Incremental Build for Top-Model SIL/PIL

When you start a top-model SIL/PIL simulation, the software regenerates code if it detects changes to your model. The software detects changes by using a checksum for the model. However, the software does not detect changes that you make to:

- The `HeaderFile` property of a `Simulink.AliasType` object
- Legacy S-functions

Therefore, if you make these changes, you must build (**Ctrl-B**) your model again before starting the next PIL simulation.

Exported Functions in Feedback Loops

If your model has function-call subsystems and you export a subsystem that has context-dependent inputs (for example, feedback signals), then the results of a SIL/

PIL simulation with the generated code might not match the results of the Normal mode simulation of your model. One approach to make SIL/PIL and Normal mode simulations yield identical results is to use “Function-Call Feedback Latch” blocks in your model. This approach allows you to make context-dependent inputs become context-independent.

Note: The software generates a warning identifying context-dependent inputs of exported function-call subsystems if you set **Configuration Parameters > Diagnostics > Connectivity > Context-dependent inputs** to one of the following:

- Enable all as warnings
- Use local settings
- Disable all

For details, see “Context-dependent inputs”.

Hardware Implementation Support

Hardware Implementation	Embedded Coder
Different host and target data-type size	No. See “Hardware Implementation Settings” on page 32-66.
Word-addressable targets	Yes. Data type sizes smaller than target word sizes are supported.
Multiword data type word order different from target byte order	No. PIL simulation fails with undefined behavior.

Hardware Implementation Settings

PIL requires that, in the Simulink Configuration Parameters dialog box, you configure the right **Hardware Implementation** settings for the target environment. You must also specify byte ordering for targets.

Warning If you do not specify the correct **Hardware Implementation**, the PIL simulation fails, displaying undefined behavior.

Host/Target Data Type Size Mismatch

PIL supports only data types that have the same size on the host and the target at the PIL I/O boundary.

The data types used at the PIL I/O boundary are restricted based on the following rule: PIL supports the data type only if the data-type size on the host (Simulink) is the same as the data-type size on the target.

- For `boolean`, `uint8`, and `int8`, the size is 8-bits.
- For `uint16` and `int16`, the size is 16-bits.
- For `uint32` and `int32`, the size is 32-bits.
- For `single`, the size is 32-bits.
- For `double`, the size is 64-bits.

Examples of unsupported data types are:

- `single` and `double` on targets with 24-bit floating-point types
- `double` on targets with 32-bit double, that is, the same size as `single`

Other Feature Support

Other Features	Top-Model SIL/PIL	Model Block SIL/PIL	SIL/PIL Block
Multiplatform support (such as Linux)	Yes	Yes	Yes
Execution profiling	Yes	Yes	Yes
Stack profiling	SIL: No. PIL: Depends on target connectivity configuration and third-party product support.	SIL: No. PIL: Depends on target connectivity configuration and third-party product support.	SIL: No. PIL: Depends on target connectivity configuration and third-party product support.
C code coverage report	Yes. See also “Tips and Limitations”.	Yes. See also “Tips and Limitations”.	SIL: No. PIL: Depends on target connectivity configuration and third-party product support.

Other Features	Top-Model SIL/PIL	Model Block SIL/PIL	SIL/PIL Block
Debugging	SIL: Yes. PIL: No.	SIL: Yes. PIL: No.	SIL: Yes. PIL: No
Non-ASCII characters in name of current working folder	SIL: No. PIL: N/A	SIL: No. PIL: N/A	SIL: No. PIL: N/A

Programmatic Code Generation Verification

In this section...

“Code Generation Verification API Overview” on page 32-69

“Verify Numerical Equivalence with CGV” on page 32-69

“Verify Numerical Equivalence Between Two Modes of Execution of a Model” on page 32-70

“Plot Output Signals” on page 32-75

Code Generation Verification API Overview

When you execute a model in different modes of execution, you can use the Code Generation Verification (CGV) API to verify the numerical equivalence of results. CGV supports executing the model in simulation, Software-In-the-Loop (SIL), and Processor-In-the-Loop (PIL). For more information about SIL and PIL, see “About SIL and PIL Simulations”. The CGV example, `rtwdemo_cgv_script`, shows CGV configuration, execution, and comparison support.

Note: CGV helps you verify the numerical equivalence of results for a given set of inputs. CGV can detect numerical deviations for the given set of inputs only. The completeness of the input data that you provide to CGV determines the validity of the results.

Verify Numerical Equivalence with CGV

Before verifying numerical equivalence:

- Configure your model for SIL or PIL simulation. For more information, see “Configure a SIL or PIL Simulation” on page 32-10.
- Use the `cgv.Config` class of the CGV API to verify the model configuration for SIL or PIL simulation. For more information, see “Check the SIL or PIL Configuration”.
- Configure your model for code generation. For more information, see “Application Objectives”.
- Save your model. If you modify a model without saving it, CGV might issue an error.

To verify numerical equivalence:

- Set up the tests for the first execution environment. For example, simulation.
- Use “run (cgv.CGV)” to run the tests for the first execution environment.
- Set up the tests for the second execution environment. For example, top-model PIL.
- Use `cgv.CGV.run` to run the tests for the second execution environment.
- Use “getOutputData (cgv.CGV)” to get the output data for each execution environment.
- Use “getSavedSignals (cgv.CGV)” to display the signal names in the output data. (optional)
- Build a list of signal names for input to other `cgv.CGV` methods. (optional)
- Use “createToleranceFile (cgv.CGV)” to create a file correlating tolerance information with output signal names. (optional)
- Use “compare (cgv.CGV)” to compare the output signals of the first and second execution environments for numerical equivalence.

Verify Numerical Equivalence Between Two Modes of Execution of a Model

The following example describes configuring, executing, and comparing the results of the `rtwdemo_cgv` model in simulation and SIL modes.

This example contains the following tasks:

- “Configure the Model” on page 32-70
- “Execute the Model” on page 32-71
- “Compare All Output Signals” on page 32-72
- “Compare Individual Output Signals” on page 32-74

Configure the Model

The first task for verifying numerical equivalence is to check the configuration of your model.

- 1 Open the `rtwdemo_cgv` model.

```
cgvModel = 'rtwdemo_cgv';  
load_system(cgvModel);
```


- 2 Save the model to a working directory.

```
save_system(cgvModel, fullfile(pwd, cgvModel));
close_system(cgvModel); % avoid original model shadowing saved model
```

- 3 Use the “cgv.Config class” to create a `cgv.Config` object. Specify parameters that check and modify configuration parameter values and save the model for top-model SIL mode of execution.

```
cgvCfg = cgv.Config('rtwdemo_cgv', 'connectivity', 'sil', 'SaveModel', 'on');
```

- 4 Use the “configModel (cgv.Config)” method to review your model configuration and to change the settings to configure your model for SIL. When 'connectivity' is set to 'sil', the system target file is automatically set to 'ert.tlc'. If you specified the parameter/value pair, ('SaveModel', 'on') when you created the `cgvCfg` object, the `cgv.Config.configModel` method saves the model.

Note: CGV runs on models that are open. If you modify a model without saving it, CGV might issue an error.

```
cgvCfg.configModel(); % Evaluate, change, and save your model for SIL
```

- 5 Display a report of the changes that `cgv.Config.configModel` makes to the model.

```
cgvCfg.displayReport(); % In this example, this reports no changes
```

Execute the Model

Use the CGV API to execute the model in two modes. The two modes in this example are normal mode simulation and SIL mode. In each execution of the model, the CGV object for each mode captures the output data and writes the data to a file.

- 1 If you have not already done so, follow the steps described in “Configure the Model” on page 32-70.
- 2 Create a `cgv.CGV` object that specifies the `rtwdemo_cgv` model in normal mode simulation.

```
cgvSim = cgv.CGV(cgvModel, 'connectivity', 'sim');
```

Note: When the top model is set to Normal simulation mode, the CGV API sets referenced models in PIL mode to Accelerator mode.

- 3 Provide the input file to the `cgvSim` object.

```
cgvSim.addInputData(1, [cgvModel '_data']);
```

- 4 Before execution of the model, specify the MATLAB files to execute or MAT-files to load. This step is optional.

```
cgvSim.addPostLoadFiles({[cgvModel '_init.m']});
```

- 5 Specify a location where the object writes all output data and metadata files for execution. This step is optional.

```
cgvSim.setOutputDir('cgv_output');
```

- 6 Execute the model.

```
result1 = cgvSim.run();
```

```
*** handling PostLoad file rtwdemo_cgv_init.m
Start CGV execution of model rtwdemo_cgv, ComponentType topmodel, ...
connectivity sim, InputData rtwdemo_cgv_data.mat
End CGV execution: status completed
```

- 7 Get the output data associated with the input data.

```
outputDataSim = cgvSim.getOutputData(1);
```

- 8 For the next mode of execution, SIL, repeat steps 2–7.

```
cgvSil = cgv.CGV( cgvModel, 'Connectivity', 'sil');
cgvSil.addInputData(1, [cgvModel '_data']);
cgvSil.addPostLoadFiles({[cgvModel '_init.m']});
cgvSil.setOutputDir('cgv_output');
result2 = cgvSil.run();
```

At the MATLAB command line, the result is:

```
*** handling PostLoad file rtwdemo_cgv_init.m
Start CGV execution of model rtwdemo_cgv, ComponentType topmodel, ...
connectivity sil, InputData rtwdemo_cgv_data.mat

### Starting build procedure for model: rtwdemo_cgv
### Successful completion of build procedure for ...
model: rtwdemo_cgv
### Preparing to start SIL simulation ...
### Starting SIL simulation for model: rtwdemo_cgv
### Stopping SIL simulation for model: rtwdemo_cgv
End CGV execution: status completed
```

Compare All Output Signals

After setting up and running the test, compare the outputs by doing the following:

- 1 If you have not already done so, configure and test the model, as described in “Configure the Model” on page 32-70 and “Execute the Model” on page 32-71.
- 2 Test that the execution result of the model:

```

if ~result1 || ~result2
    error('Execution of model failed.');
```

- 3 Use the “getOutputData (cgv.CGV)” method to get the output data from the cgv.CGV objects.

```

simData = cgvSim.getOutputData(1);
silData = cgvSil.getOutputData(1);
```

- 4 Display a list of signals by name using the “getSavedSignals (cgv.CGV)” method.

```

cgvSim.getSavedSignals(simData);
```

At the MATLAB command line, the result is:

```

simData.hi0.Data(:,1)
simData.hi0.Data(:,2)
simData.Vector.Data(:,1)
simData.Vector.Data(:,2)
simData.Vector.Data(:,3)
simData.Vector.Data(:,4)
simData.BusOutputs.hi0.Data(:,1)
simData.BusOutputs.hi0.Data(:,2)
simData.BusOutputs.hi1.mid0.lo0.Data(1,1,:)
simData.BusOutputs.hi1.mid0.lo0.Data(1,2,:)
simData.BusOutputs.hi1.mid0.lo0.Data(2,1,:)
simData.BusOutputs.hi1.mid0.lo0.Data(2,2,:)
simData.BusOutputs.hi1.mid0.lo1.Data
simData.BusOutputs.hi1.mid0.lo2.Data
simData.BusOutputs.hi1.mid1.Data(:,1)
simData.BusOutputs.hi1.mid1.Data(:,2)
simData.ErrorsInjected.Data
```

- 5 Using the list of signals, build a list of signals in a cell array of strings. The signal list can contain a number of signals.

```

signalList = {'simData.ErrorsInjected.Data'};
```

- 6 Use the “createToleranceFile (cgv.CGV)” method to create a file, in this example, 'localtol', correlating tolerance information with output signal names.

```

toleranceList = {'absolute', 0.5}};
cgv.CGV.createToleranceFile('localtol', signalList, toleranceList);
```

- 7 Compare the output data signals. By default, the “compare (cgv.CGV)” method looks at all signals which have a common name between both executions. If a tolerance file is present, cgv.CGV.compare uses the associated tolerance for a specific signal during comparison; otherwise the tolerance is zero. In this example, the 'Plot' parameter is set to 'mismatch'. Therefore, only mismatched signals produce a plot.

```

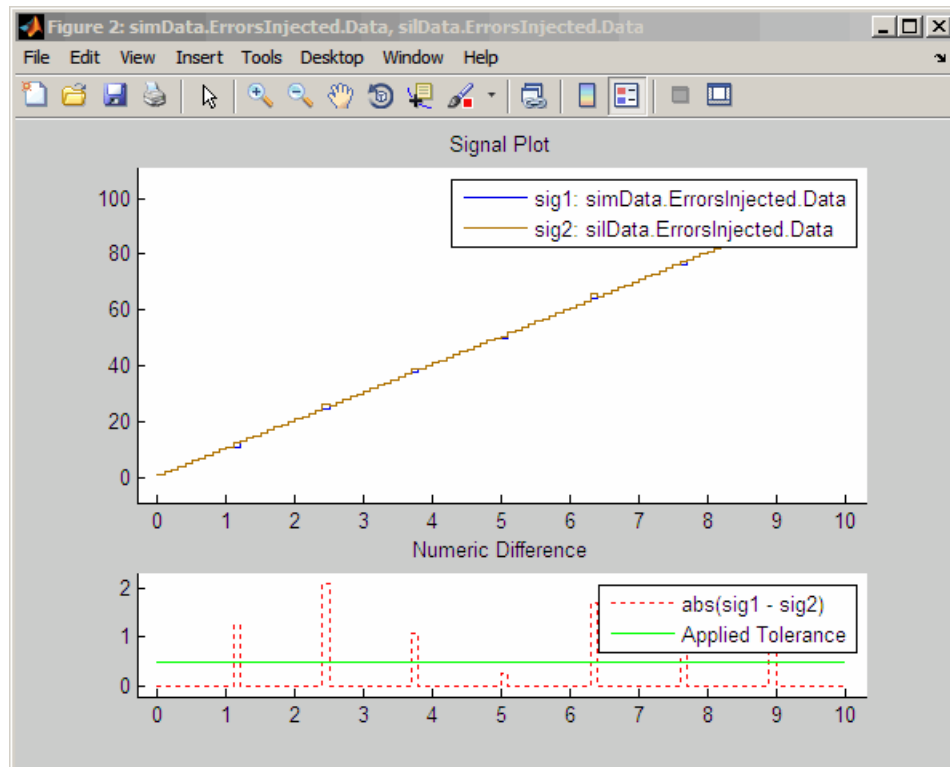
[matchNames, ~, mismatchNames, ~] = ...
    cgv.CGV.compare(simData, silData, 'Plot', 'mismatch', ...
        'Tolerancefile', 'localtol');
```

```
fprintf( '%d Signals match, %d Signals mismatch\n', ...
        length(matchNames), length(mismatchNames));
disp('Mismatched Signal Names:');
disp(mismatchNames);
```

At the MATLAB command line, the result is:

```
14 Signals match, 1 Signals mismatch
Mismatched Signal Names:
    'simData.ErrorsInjected.Data'
```

A plot results from the mismatch on signal `simData.ErrorsInjected.Data`.



The lower plot displays the numeric difference between the results.

Compare Individual Output Signals

After setting up and running the test, compare the outputs of individual signals by doing the following:

- 1 If you have not already done so, configure and test the model, as described in “Configure the Model” on page 32-70 and “Execute the Model” on page 32-71.
- 2 Use the “getOutputData (cgv.CGV)” method to get the output data from the `cgv.CGV` objects.

```
simData = cgvSim.getOutputData(1);
silData = cgvSil.getOutputData(1);
```

- 3 Use the “getSavedSignals (cgv.CGV)” method to display the output data signal names. Build a list of specific signal names in a cell array of strings. The signal list can contain number of signals.

```
cgv.CGV.getSavedSignals(simData);
signalList = {'simData.BusOutputs.hi1.mid0.lo1.Data', ...
'simData.BusOutputs.hi1.mid0.lo2.Data', 'simData.Vector.Data(:,3)'};
```

- 4 Use the specified signals as input to the “compare (cgv.CGV)” method to compare the signals from separate runs.

```
[matchNames, ~, mismatchNames, ~] = ...
    cgv.CGV.compare(simData, silData, 'Plot', 'mismatch', ...
    'signals', signalList);
fprintf( '%d Signals match, %d Signals mismatch\n', ...
    length(matchNames), length(mismatchNames));
if ~isempty(mismatchNames)
    disp( 'Mismatched Signal Names:');
    disp(mismatchNames);
end
```

At the MATLAB command line, the result is:

```
3 Signals match, 0 Signals mismatch
```

Plot Output Signals

After setting up and running the test, use the “plot (cgv.CGV)” method to plot output signals.

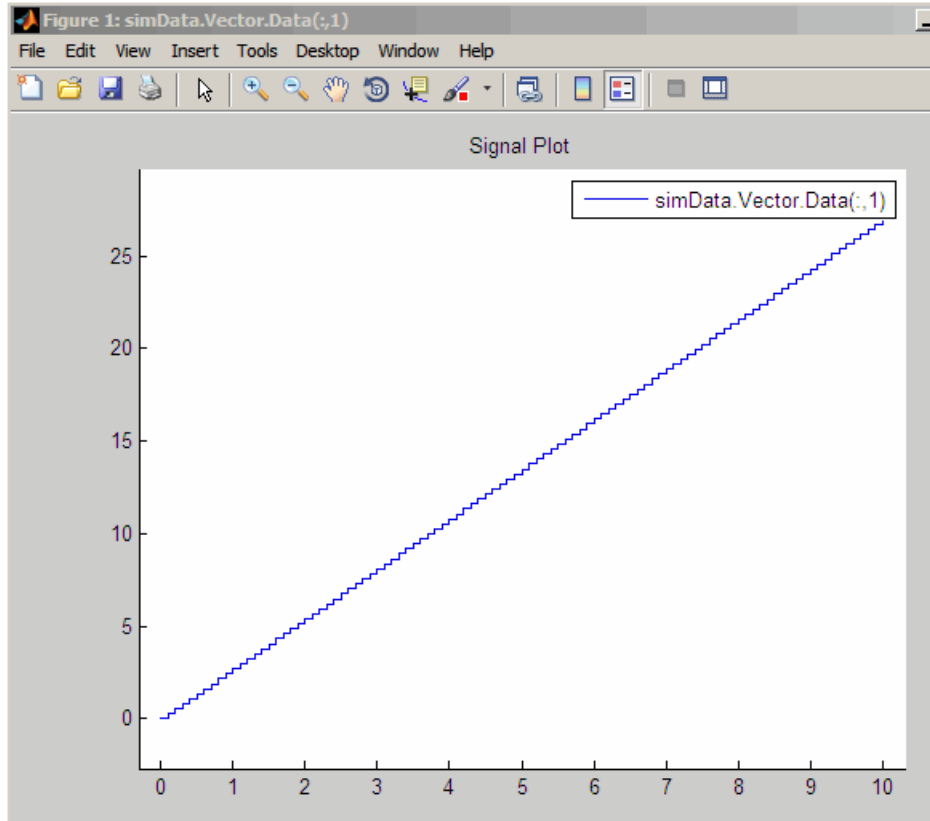
- 1 If you have not already done so, configure and test the model, as described in “Configure the Model” on page 32-70 and “Execute the Model” on page 32-71.
- 2 Use the “getOutputData (cgv.CGV)” method to get the output data from the `cgv.CGV` objects.
- 3 Use the “getSavedSignals (cgv.CGV)” method to display the output data signal names. Build a list of specific signal names in a cell array of strings. The signal list can contain number of signals.

```
cgv.CGV.getSavedSignals(simData);
```

```
signalList = {'simData.Vector.Data(:,1)'};
```

- 4 Use the specified signal list as input to the “plot (cgv.CGV)” method to compare the signals from separate runs.

```
[signalNames, signalFigures] = cgv.CGV.plot(simData, ...  
    'Signals', signalList);
```



Software-in-the-Loop Execution for MATLAB Coder

- “Code Verification Through Software-in-the-Loop and Processor-in-the-Loop Execution” on page 33-2
- “Software-in-the-Loop Execution Through Project Interface” on page 33-4
- “Software-in-the-Loop Execution From Command Line” on page 33-6
- “Code Debugging During SIL Execution” on page 33-9
- “PIL Customization for Target Environment” on page 33-12
- “Create PIL Target Connectivity Configuration” on page 33-15
- “Processor-in-the-Loop Execution Through Project Interface” on page 33-22
- “Processor-in-the-Loop Execution From Command Line” on page 33-24
- “SIL/PIL Execution Support and Limitations” on page 33-29

Code Verification Through Software-in-the-Loop and Processor-in-the-Loop Execution

MATLAB Coder supports software-in-the-loop (SIL) and processor-in-the-loop (PIL) execution, which enables you to verify production-ready source code and compiled object code. With these execution modes, you can reuse test vectors developed for your MATLAB functions to verify the numerical behavior of library code.

In SIL execution, through a MATLAB SIL interface, the software compiles and runs library code on your host computer. In PIL execution, through a MATLAB PIL interface, the software cross-compile and runs production object code on a target processor or an equivalent instruction set simulator. Before you run a PIL execution, you must set up a PIL connectivity configuration for your target.

The workflow for generating and verifying code is:

- 1 Set up MATLAB Coder.
- 2 Fix errors detected at design time.
- 3 Generate MEX function.
- 4 Test MEX function.
- 5 Generate C/C++ library code.
- 6 Verify generated C/C++ code through SIL or PIL execution — requires Embedded Coder license.

In step 4, you verify code that is generated for execution within MATLAB. However, this code is different from the standalone code generated for libraries. In step 6, with an Embedded Coder license, you use SIL or PIL execution to verify the standalone code.

For more information, use the following table.

Feature	See
SIL execution	<ul style="list-style-type: none"> • “Software-in-the-Loop Execution Through Project Interface” on page 33-4 • “Software-in-the-Loop Execution From Command Line” on page 33-6

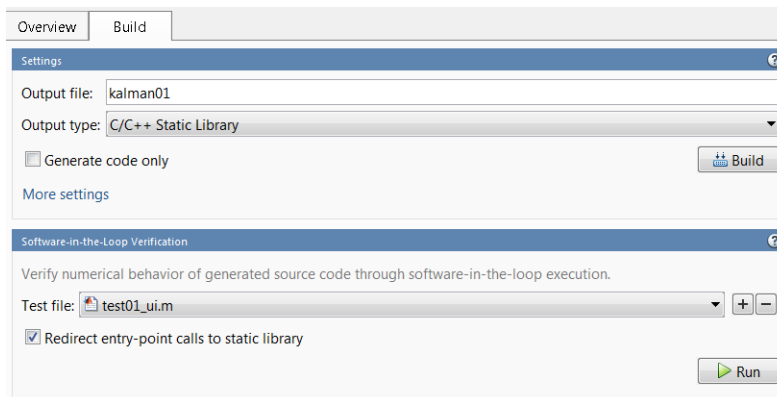
Feature	See
PIL target connectivity configuration	<ul style="list-style-type: none">• “PIL Customization for Target Environment” on page 33-12• “Create PIL Target Connectivity Configuration” on page 33-15• “Processor-in-the-Loop Execution From Command Line” on page 33-24
PIL execution	<ul style="list-style-type: none">• “Processor-in-the-Loop Execution Through Project Interface” on page 33-22• “Processor-in-the-Loop Execution From Command Line” on page 33-24
Code generation, MEX functions, and libraries	<ul style="list-style-type: none">• “MATLAB Code Analysis”• “Generating Code”• “Deployment”

Software-in-the-Loop Execution Through Project Interface

Use software-in-the-loop (SIL) execution to verify the numerical behavior of the generated C/C++ code with reference to your original MATLAB functions.

To set up and run a SIL execution for a project through the MATLAB Coder project interface:

- 1 From the MATLAB apps gallery, open your MATLAB Coder project. For example, `kalman_filter.prj`. For information about creating a project, see “C Code Generation Using the Project Interface”.
- 2 Click the **Build** tab.



- 3 Under **Settings**:
 - a In the **Output file** field, use the default value. For example, `kalman01`.
 - b From the **Output type** drop-down list, select `C/C++ Static Library` or `C/C++ Dynamic Library`.
 - c Clear the **Generate code only** check box.
 - d Click **More Settings**.
- 4 In the Project Settings dialog box:
 - a Select the **Hardware > Test hardware is the same as production hardware** check box.
 - b From the **Hardware > Device Type** drop-down list, select `MATLAB Host Computer`.

- c** If required, enable your debugger for SIL execution. From the **All Settings > Advanced > Enable source-level debugging for SIL** drop-down list, select **Yes**.
 - d** Click **Close**.
- 5** Under **Software-in-the-Loop Verification**:
- a** In the **Test file** field, specify the test file that calls the original MATLAB functions, for example, `test01_ui.m`.
 - b** By default, the **Redirect entry-point calls to generated library** check box is selected. Otherwise, select this check box.
- 6** To start the SIL execution, click **Run**.

The software:

- Generates a standalone library, for example, `kalman01`.
- Runs the test file, replacing calls to the MATLAB function by calls to the generated code in the library.

You can halt the execution at any time by clicking **Stop**.

Verify that the output of the SIL execution matches the output from your original MATLAB function.

- 7** To terminate the SIL execution process, click **Terminate verification**.

Note: On a Windows operating system, the Windows Firewall might block the SIL execution. To allow the SIL execution, use the Windows Security Alert dialog box. For example, in Windows 7, click **Allow access**.

Software-in-the-Loop Execution From Command Line

Use software-in-the-loop (SIL) execution to verify the numerical behavior of the generated C/C++ code with reference to your original MATLAB functions.

To set up and start a SIL execution from the command line:

- 1 Create a `coder.EmbeddedCodeConfig` object.
- 2 Configure the object for SIL.
- 3 Use the `codegen` function to generate library code for your MATLAB function and the SIL interface.
- 4 Use the `coder.runTest` function to run the test file for your original MATLAB function.

To terminate the SIL execution, use the `clear function_sil` or `clear mex` command.

The following example shows how you can set up and run a SIL execution from the command line.

SIL Execution of Code Generated for a Kalman Estimator

1 Copy MATLAB code for Kalman estimator

From `docroot\toolbox\coder\examples\kalman`, copy the following files to your working folder:

- `kalman01.m` — MATLAB function for the Kalman estimator
- `test01_ui.m` — MATLAB file to test `kalman01.m`
- `plot_trajectory.m` — File that plots actual target trajectory and Kalman estimator output
- `position.mat` — Input data

```
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'), '.')
```

For a description of the Kalman estimator in this example, see “C Code Generation at the Command Line”.

2 Configure SIL execution

- a** From your working folder, create a `coder.EmbeddedCodeConfig` object.

```
config = coder.config('lib');
config.GenerateReport = true; % Optional, documents code in HTML report
```

- b** Configure the object for SIL.

```
config.VerificationMode = 'SIL';

% Check that production hardware setting is the default
% i.e. 'Generic->MATLAB Host Computer'
disp(config.HardwareImplementation.ProdHWDeviceType);
```

- c** If required, enable the Microsoft Visual Studio debugger for SIL execution:

```
config.SILDebugging = true;
```

3 Generate code and run SIL execution

- a** Generate library code for the `kalman01` MATLAB function and the SIL interface.

```
codegen('-config', config, '-args', {zeros(2,1)}, 'kalman01');
```

The software creates the following output folders:

- `codegen\lib\kalman01` — Standalone code for `kalman01`.
- `codegen\lib\kalman01\sil` — SIL interface code for `kalman01`.

- b** Run the MATLAB test file `test01_ui` with `kalman01_sil`. `kalman01_sil` is the SIL interface for `kalman01`.

```
coder.runTest('test01_ui', ['kalman01_sil.' mexext]);
```

Verify that the output of this run matches the output from the original `kalman01.m` function.

Note: On a Windows operating system, the Windows Firewall might block the SIL execution. To allow the SIL execution, use the Windows Security Alert dialog box. For example, in Windows 7, click **Allow access**.

4 Debug code during SIL execution

If you enable the Microsoft Visual Studio debugger, then running the test file opens the Microsoft Visual Studio IDE with debugger breakpoints at the start of the `kalman01_initialize` and `kalman01` functions.

You can use the debugger features to observe code behavior. For example, you can step through code and examine variables.

To end the debugging session:

- a Remove all breakpoints.
- b Click the **Continue** button (**F5**).

The SIL execution runs to completion.

5 Terminate SIL execution

Terminate the SIL execution process.

```
clear kalman01_sil;
```

You can also use the command `clear mex`, which clears MEX functions from memory.

Code Debugging During SIL Execution

If you notice differences between the outputs of your original MATLAB functions and the generated code from a SIL execution, you can rerun the SIL execution with a debugger enabled. By inserting breakpoints, you can observe the behavior of code sections, which might help you to understand the cause of the differences in results.

The software supports the following debuggers:

- On Windows, Microsoft Visual Studio debugger.
- On Linux, GNU Data Display Debugger (DDD).

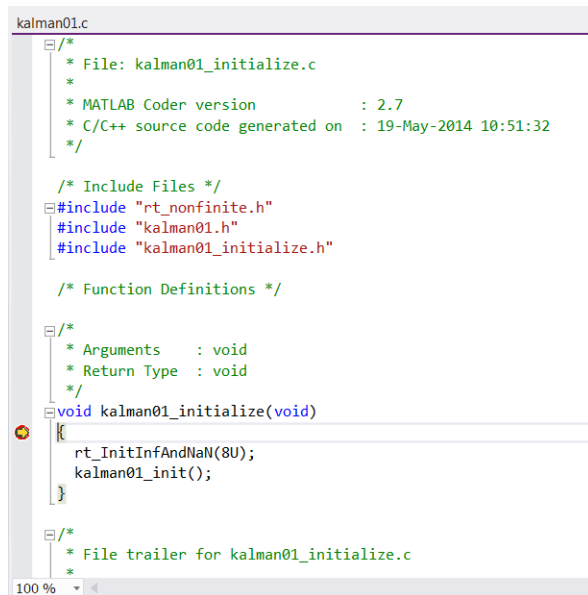
Note: You can perform SIL debugging only if your Microsoft Visual C++ or GNU GCC compiler is supported by the MATLAB product family. For more information, see supported compilers.

To enable your debugger for SIL execution:

- 1 In MATLAB Coder Project Interface, click the **Build** tab.
- 2 Under **Settings**, click **More settings**.
- 3 In the Project Settings dialog box, from the **All Settings > Advanced > Enable source-level debugging for SIL** drop-down list, select **Yes**.
- 4 Click **Close**.

For information about enabling the debugger from the command line, see “SIL Execution of Code Generated for a Kalman Estimator” on page 33-6.

When you run the SIL execution, for example on a Windows computer, your *user_fn.c* or *user_fn.cpp* file opens in the Microsoft Visual Studio IDE with debugger breakpoints at the start of the *user_fn_initialize* and *user_fn* functions.



```
kalman01.c
/*
 * File: kalman01_initialize.c
 *
 * MATLAB Coder version      : 2.7
 * C/C++ source code generated on : 19-May-2014 10:51:32
 */

/* Include Files */
#include "rt_nonfinite.h"
#include "kalman01.h"
#include "kalman01_initialize.h"

/* Function Definitions */

/*
 * Arguments      : void
 * Return Type    : void
 */
void kalman01_initialize(void)
{
    rt_InitInfAndNaN(8U);
    kalman01_init();
}

/*
 * File trailer for kalman01_initialize.c
 */
```

You can now use the debugger features to observe code behavior. For example, you can step through code and examine variables.

To end the debugging session:

- 1 Remove all breakpoints.
- 2 Click the **Continue** button (**F5**).

The SIL execution runs to completion.

- 3 To terminate the SIL execution process, in the MATLAB Command Window, click the `clear kalman01_sil` link.

The Microsoft Visual Studio IDE closes automatically.

Note: If you select **Debug > Stop Debugging**, the SIL execution times out with the following error message:

Communications error: failed to send data to the target. There might be multiple reasons for this failure.

...
...

PIL Customization for Target Environment

In this section...

“Target Connectivity Configurations for PIL” on page 33-12

“Target Connectivity PIL API Components” on page 33-12

“Communications `rtiostream` API” on page 33-13

Target Connectivity Configurations for PIL

Use target connectivity configurations and the target connectivity API to customize processor-in-the-loop (PIL) verification for target environments.

Through a target connectivity configuration, you specify:

- A target connectivity configuration name for a target connectivity API implementation.
- Settings that define the MATLAB code that the configuration is compatible with, for example, the code that is generated for a particular hardware implementation.

You must associate a connectivity configuration name with a connectivity API implementation. You can have many different connectivity configurations, each configuration being available for PIL simulation. Register a connectivity configuration with MATLAB by creating an `rtwTargetInfo.m` file and placing it on the MATLAB search path.

To run a PIL verification, the software must first determine which of the available connectivity configurations to use. The software looks for a connectivity configuration that is compatible with the code under test. If the software finds multiple or no compatible connectivity configurations, the software generates an error message with information about resolving the problem.

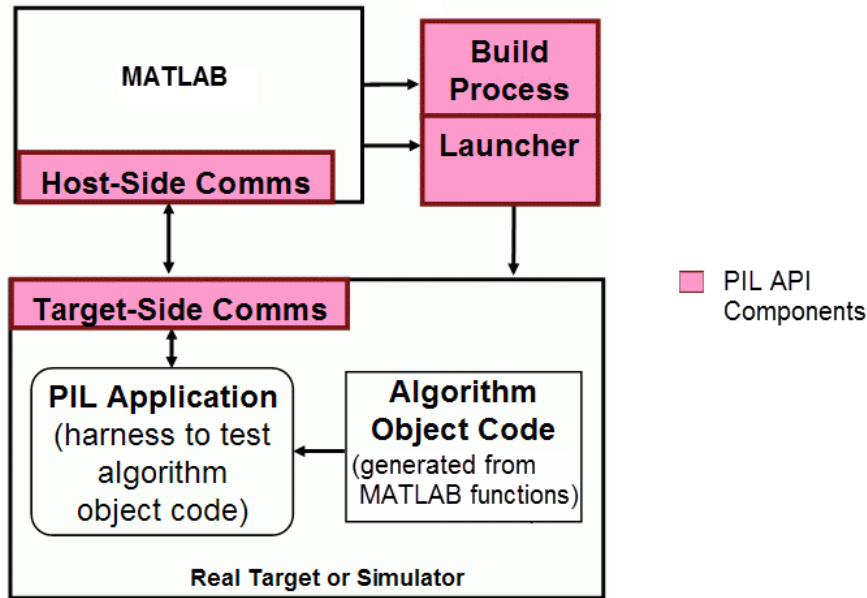
For more information, see:

- “Target Connectivity PIL API Components”
- “Create PIL Target Connectivity Configuration”

Target Connectivity PIL API Components

Use the target connectivity PIL API to integrate third-party tools for:

- Building the PIL application, an executable for the target hardware
- Downloading, starting, and stopping the application on the target
- Communicating between MATLAB and the target



Communications `rtiostream` API

The `rtiostream` API supports communications for the target connectivity API. Use the `rtiostream` API to implement a communication channel that enables data exchange between different processes.

PIL verification requires a host-target communications channel. This communications channel comprises driver code that runs on the host and target. The `rtiostream` API defines the signature of target-side and host-side functions that must be implemented by this driver code.

The API is independent of the physical layer that sends the data. Possible physical layers include RS232, Ethernet, or Controller Area Network (CAN).

A full `rtiostream` implementation requires both host-side and target-side drivers. Code generation software includes host-side drivers for the default TCP/IP implementation as well as a version for serial communications. To use:

- The TCP/IP `rtiostream` communications channel, you must provide, or obtain from a third party, target-specific TCP/IP device drivers.
- The serial communications channel, you must provide, or obtain from a third party, target-specific serial device drivers.

For other communication channels and platforms, the code generation software does not provide default implementations. You must provide both the host-side and the target-side drivers.

The `rtiostream` API comprises the following functions:

- `rtIOStreamOpen`
- `rtIOStreamSend`
- `rtIOStreamRecv`
- `rtIOStreamClose`

For information about:

- Using `rtiostream` functions in a connectivity implementation, see “Create a Connectivity API Implementation” on page 33-15.
- Testing the `rtiostream` shared library methods from MATLAB code, see `rtiostream_wrapper`.
- Debugging and verifying the behavior of custom `rtiostream` interface implementations, see “Test an `rtiostream` Driver” on page 33-16.

Create PIL Target Connectivity Configuration

In this section...

“Create a Connectivity API Implementation” on page 33-15

“Test an `rtiostream` Driver” on page 33-16

“Synchronize Host and Target” on page 33-38

“Specify Hardware Timer” on page 33-19

“Register a Connectivity API Implementation” on page 33-41

Create a Connectivity API Implementation

To create a target connectivity API implementation, you must create a subclass of `rtw.connectivity.Config`.

- You must instantiate `rtw.connectivity.MakefileBuilder`. This class configures the build process.
- You must create a subclass of `rtw.connectivity.Launcher`. This class downloads and executes the application using a third-party tool.
- Configure your `rtiostream` communications implementation:
 - On the target-side, integrate the driver code implementing `rtiostream` functions directly into the build process by creating a subclass of `rtw.pil.RtIOStreamApplicationFramework`.
 - On the host-side, compile the driver code into a shared library. You load and initialize this shared library by instantiating (or optionally, customizing) `rtw.connectivity.RtIOStreamHostCommunicator`.
- For code execution profiling, you must create a timer object that provides details of the hardware-specific timer and associated source files. See “Specify Hardware Timer”.

For information about creating a subclass, see “Creating Subclasses — Syntax and Techniques” in MATLAB documentation.

For a target connectivity API implementation example, see “Processor-in-the-Loop Execution From Command Line” on page 33-24.

Test an `rtiostream` Driver

Use a test suite to debug and verify the behavior of custom `rtiostream` interface implementations.

The test suite has the following advantages:

- Reduces time for integrating custom hardware that does not have built-in `rtiostream` support.
- Reduces time for testing custom `rtiostream` drivers.
- Helps analyze the performance of custom `rtiostream` drivers.

The test suite has two parts. One part of the test suite runs on the target.

Note: After building the target application, download it to the target and run it.

To launch this part, compile and link the following files, which are in `matlabroot/toolbox/coder/rtiostream/src/rtiostreamtest`.

- `rtiostreamtest.c`
- `rtiostreamtest.h`
- `rtiostream.h` (located at `matlabroot/rtw/c/src/`)
- `rtiostream` implementation under investigation (for example, `rtiostream_tcpip.c`)
- `main.c`

To run the MATLAB part of the test suite, invoke `rtiostreamtest`. The syntax is as follows:

```
rtiostreamtest(connection,param1,param2)
```

- `connection` is a string indicating the communication method. It can have values `'tcp'` or `'serial'`.
- `param1` and `param2` have different values depending on the value of `connection`.
 - If `connection` is `'tcp'`, then `param1` and `param2` are hostname and port, respectively. For example, `rtiostreamtest('tcp', 'localhost', 2345)`.

- If connection is 'serial', then param1 and param2 are COM port and baud rate, respectively. For example, `rtiostreamtest('serial', 'COM1', 9600)`.

You can run the MATLAB part of the test suite as follows:

```
rtiostreamtest('tcp', 'localhost', '2345')
```

An output in the following format appears in the MATLAB window:

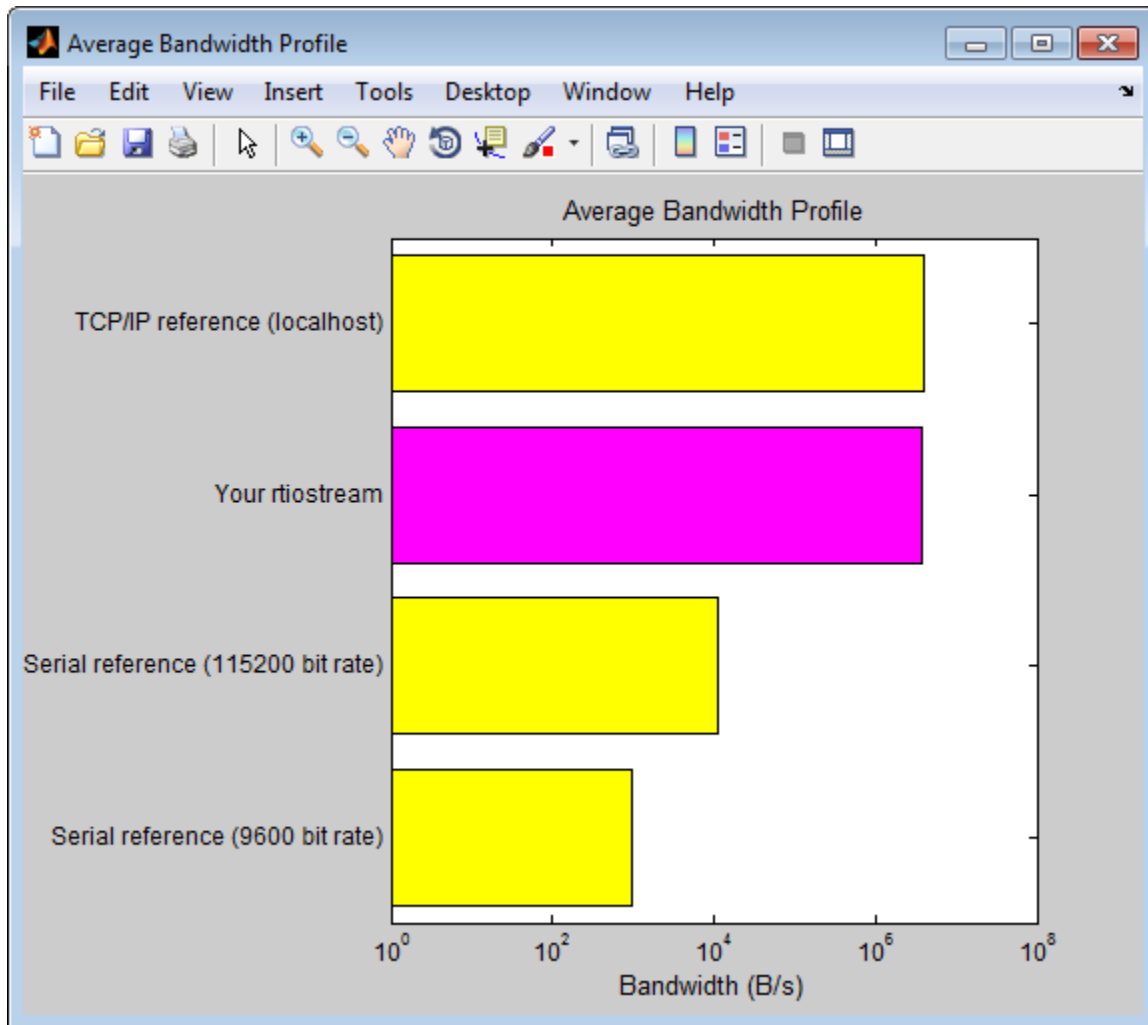
```
### Test suite for rtiostream ###
Initializing connection with target...

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

### rtiostream characteristics discovered
Round trip time : 0.96689 ms
rtIOStreamRecv behavior : non-blocking

### Test results
Test 1 (fixed size data exchange): ..... PASS
Test 2 (varying size data exchange): ..... PASS

### Test suite for rtiostream finished successfully ###
Furthermore, the following profile appears.
```



Synchronize Host and Target

If you use the `rtostream` API to implement the communications channel, the host and target must be synchronized, which prevents MATLAB from transmitting and receiving data before the target application is fully initialized.

To synchronize the host and target for TCP/IP `rtiostream` implementations, use the `setInitCommsTimeout` method from `rtw.connectivity.RtIOStreamHostCommunicator`. This approach works well for connection-oriented TCP/IP `rtiostream` implementations because MATLAB automatically waits until the target server is running.

With other `rtiostream` implementations, for example, serial, the MATLAB side of the `rtiostream` connection will open without waiting for the target to be fully initialized. In this case, you must make your `Launcher` implementation wait until the target application is fully initialized. Use one of the following approaches to synchronize your host and target:

- Add a pause at the end of the `Launcher` implementation that makes the `Launcher` wait until target initialization is complete.
- In the `Launcher` implementation, use third-party downloader or debugger APIs that wait until target initialization is complete.
- Implement a handshaking mechanism in the `Launcher / rtiostream` implementation to confirm that target initialization is complete.

Specify Hardware Timer

For code execution profiling, you must create a timer object that provides details of the hardware-specific timer and associated source files. You can use the Code Replacement Tool or the code replacement library API to specify this hardware-specific timer.

To specify the timer with the Code Replacement Tool:

- 1 Open the Code Replacement Tool. In the Command Window, enter `crtool`.
- 2 Create a new code replacement table. Select **File > New table**.
- 3 Create a new function entry. Under **Tables List**, right-click the new table. Then, from the context-menu, select **New entry > Function**.
- 4 In the middle view, select the new unnamed function.
- 5 On the **Mapping Information** pane:
 - a From the **Function** drop-down list, select `code_profile_read_timer`.
 - b Specify the count direction for your timer. For example, from the **Count direction** drop-down list, select **Up**.
 - c In the **Ticks per second** field, specify the number of ticks per second for your timer, for example, `1e+09`.

The default value is 0. In this case, the software reports time measurements in terms of ticks, not seconds.

- d In the **Name** field, specify a replacement function name, for example, `MyTimer`.
- e Click **Apply**.

- f To validate the function entry, click **Validate entry**.
- 6 On the **Build Information** pane, specify the required build information. See “Specify Build Information for Replacement Code”.
 - 7 Save the table (**Ctrl+S**). When you save the table for the first time, use the Save As dialog box to specify the file name and location.

You must save the table in a location that is on the MATLAB search path. For example, you can save this file in the folder for your subclass of `rtw.connectivity.Config`.

The software stores your timer information as a code replacement library table.

- 8** Assuming you save the table as *MyCr1Table.m*, in your subclass of `rtw.connectivity.Config`, add the following line:

```
setTimer(this, MyCr1Table)
```

For more information, see “What Is Code Replacement?” and “What Is Code Replacement Customization?”.

Register a Connectivity API Implementation

Register the new connectivity API implementation with MATLAB as a connectivity configuration, by creating or adding to an `rtwTargetInfo.m` file. Through this action, you also specify the MATLAB code that is compatible with the new connectivity configuration.

For more information, see:

- `rtw.connectivity.ConfigRegistry`
- “Processor-in-the-Loop Execution From Command Line”

.

Processor-in-the-Loop Execution Through Project Interface

Use processor-in-the-loop (PIL) execution to verify the numerical behavior of cross-compiled object code, with reference your original MATLAB functions.

Before you run a PIL execution, you must specify a target connectivity configuration. In “Processor-in-the-Loop Execution From Command Line” on page 33-24, steps 1 and 2 of the example, PIL Execution of Code Generated for a Kalman Estimator, show how you can set up and register a connectivity configuration for host-based PIL.

To set up and run a PIL execution for a project through the MATLAB Coder project interface:

- 1 From the MATLAB apps gallery, open your MATLAB Coder project. For example, `kalman_filter.prj`. For information about creating a project, see “C Code Generation Using the Project Interface”.
- 2 Click the **Build** tab.
- 3 Under **Settings**:
 - a In the **Output file** field, use the default value. For example, `kalman01`.
 - b From the **Output type** drop-down list, select `C/C++ Static Library` or `C/C++ Dynamic Library`.
 - c Clear the **Generate code only** check box.
 - d Click **More Settings**.
- 4 In the Project Settings dialog box:
 - a Click **Hardware**.
 - b Specify your device vendor and type, which must match the target hardware settings specified in the `rtwTargetInfo.m` file of your target connectivity configuration. For example, for host-based PIL:
 - From the **Device Vendor** drop-down list, select `Generic`.
 - From the **Device Type** drop-down list, select `32-bit x86 compatible`.
 - c Click **Close**.
- 5 Under **Processor-in-the-Loop Verification**:
 - a In the **Test file** field, specify the test file that calls the original MATLAB functions, for example, `test01_ui.m`.

- b** By default, the **Redirect entry-point calls to generated library** check box is selected. Otherwise, select this check box.
- 6** To start the PIL execution, click **Run**.

The software:

- Generates a standalone library, for example, `codegen\lib\kalman01`.
- Generates PIL interface code, for example, `codegen\lib\kalman01\pil`
- Runs the test file, replacing calls to the MATLAB function by calls to the generated code in the library.

You can halt the execution at any time by clicking **Stop**.

Verify that the output of the PIL execution matches the output from your original MATLAB function.

- 7** To terminate the PIL execution process, click **Terminate verification**.

Processor-in-the-Loop Execution From Command Line

Use processor-in-the-loop (PIL) execution to verify code that you intend to deploy in production.

To set up and start a PIL execution from the command line:

- 1 Create a connectivity configuration for your target.
- 2 Create a `coder.EmbeddedCodeConfig` object.
- 3 Configure the object for PIL.
- 4 Use the `codegen` function to generate library code for your MATLAB function and the PIL interface.
- 5 Use the `coder.runTest` function to run the test file for your original MATLAB function.

To terminate the PIL execution, use the `clear function_pil` or `clear mex` command.

The following example shows how you can set up and run a host-based PIL execution from the command line.

PIL Execution of Code Generated for a Kalman Estimator

1 Create a target connectivity API implementation

- a 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');
```

- b Make the copied files writable.

```
fileattrib(fullfile('+mypil', '*'), '+w');
```

- c Update the package name to reflect the new location of the files.

```

coder.mypil.Utils.UpdateClassName(...
    './+mypil/ConnectivityConfig.m',...
    'coder.mypil',...
    'mypil');

```

- d** Check that you now have a folder `+mypil` in the current folder, which includes three files, `Launcher.m`, `TargetApplicationFramework.m`, and `ConnectivityConfig.m`.

```
dir './+mypil'
```

- e** Review the code that starts the PIL application. The `mypil.Launcher` class configures a tool for starting the PIL executable. Open this class in the editor.

```
edit(which('mypil.Launcher'))
```

Review the content of this file. For example, consider the `setArgString` method. This method allows additional command line parameters to be supplied to the application. These parameters can include a TCP/IP port number. For an embedded processor implementation, you might have to hard code these settings.

- f** The class `mypil.ConnectivityConfig` configures target connectivity.

```
edit(which('mypil.ConnectivityConfig'))
```

Review the content of this file. For example:

- The creation of an instance of `rtw.connectivity.RtIOStreamHostCommunicator` that configures the host side of the TCP/IP communications channel.
- A call to the `setArgString` method of `Launcher` that configures the target side of the TCP/IP communications channel.
- A call to `setTimer` that configures a timer for execution time measurement. To define your own target-specific timer for execution time profiling, you must use the Code Replacement Library to specify a replacement for the function `code_profile_read_timer`.

- g** Review the target-side communication drivers.

```
rtiostreamtcpip_dir=fullfile(matlabroot,'rtw','c','src','rtiostream',...
    'rtiostreamtcpip');
```

```
edit(fullfile(rtiostreamtcpip_dir,'rtiostream_tcpip.c'))
```

Scroll down to the end of this file. The file contains a TCP/IP implementation of the functions `rtIOStreamOpen`, `rtIOStreamSend`, and `rtIOStreamRecv`.

These functions are required for target communication with the host. For each of these functions, you must provide an implementation that is specific to your target hardware and communication channel.

The `myPil.TargetApplicationFramework` class adds target-side communication drivers to the connectivity configuration.

```
edit(which('myPil.TargetApplicationFramework'))
```

The file specifies additional files to include in the build.

2 Register a target connectivity configuration

Use an `rtwTargetInfo.m` file to:

- Create a target connectivity configuration object.
- Invoke `registerTargetInfo`, which registers the target connectivity configuration.

The target connectivity configuration object specifies, for example:

- The configuration name and associated API implementation. See `rtw.connectivity.ConfigRegistry`
 - A toolchain for your target hardware. This example assumes that the target hardware is your host computer, and uses the toolchain supplied for host-based PIL verification. For information about toolchains, see “Custom Toolchain Registration”.
- Insert the following code into your `rtwTargetInfo.m` file, and save the file in the current working folder or in a folder that is on the MATLAB search path:

```
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 TargetHWDeviceType property, define compatible code for the
% target connectivity configuration
config.TargetHWDeviceType = {'Generic->32-bit x86 compatible' ...
                             'Generic->Custom' ...
                             'ARM Compatible->ARM Cortex'};

```

- b** Refresh the MATLAB Coder library registration information.

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

3 Copy MATLAB code for Kalman estimator

Copy the 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'), '.')

```

For a description of the Kalman estimator in this example, see “C Code Generation at the Command Line”.

4 Configure the PIL execution

- a** Create a `coder.EmbeddedCodeConfig` object.

```
config = coder.config('lib');
```

- b** Configure the object for PIL.

```
config.VerificationMode = 'PIL';
```

- c** Specify production hardware, which must match one of the test hardware settings in `rtwTargetInfo.m`.

```

config.HardwareImplementation.ProdHWDeviceType = ...
    'Generic->32-bit x86 compatible';

```

5 Generate code and run PIL execution

- a** Generate library code for the `kalman01` MATLAB function and the PIL interface.

```
codegen('-config', config, '-args', {zeros(2,1)}, 'kalman01');
```

The software creates the following output folders:

- `codegen\lib\kalman01` — Standalone code for `kalman01`.
- `codegen\lib\kalman01\pil` — PIL interface code for `kalman01`.

- b** Run the MATLAB test file `test01_ui` with `kalman01_pil`. `kalman01_pil` is the PIL interface for `kalman01`.

```
coder.runTest('test01_ui', ['kalman01_pil.' mexext]);
```

Verify that the output of this run matches the output from the original `kalman01.m` function.

6 Terminate PIL execution

Terminate the PIL execution process.

```
clear kalman01_pil;
```

SIL/PIL Execution Support and Limitations

Feature		Supported
Output types	Static library	Yes
	Dynamic library	Yes
	Executable	No
Languages	C	Yes
	C++	Yes
Interface types	Inputs	Yes
	Outputs	Yes
	Constant inputs	Yes
	Global data	No
	Constant global data	Yes
	Reentrant code	Yes
	Multiple entry points	Yes
Data types	Basic types	Yes
	Enumerated types	Yes
	Structures	Yes
	Complex data	Yes
	Fixed-point data	Yes
	Multiword fixed-point data	SIL only
	char arrays	Yes
	Empty values	Yes
Size	Scalars	Yes
	Fixed-size arrays	Yes
	Static variable-size arrays	Yes

Feature		Supported
	Dynamic variable-size arrays	No

Code Coverage

- “Code Coverage in SIL and PIL Simulations” on page 34-2
- “Configure SIL and PIL Code Coverage” on page 34-3
- “View Code Coverage Information at the End of SIL or PIL Simulations” on page 34-5
- “Configure Code Coverage Programmatically” on page 34-8
- “Code Coverage Summary and Annotations” on page 34-10
- “Code Coverage Tool Support” on page 34-15
- “Code Coverage for PIL” on page 34-16
- “Tips and Limitations” on page 34-18

Code Coverage in SIL and PIL Simulations

During a top-model or Model block SIL or PIL simulation, you can collect code coverage metrics for generated code using a third-party tool. Embedded Coder supports the following tools:

- LDRA Testbed[®] from LDRA Software Technology. For information about installing and using this tool, go to www.ldra.com.

The software supports LDRA Testbed code coverage for SIL and PIL. For information about PIL support, see “PIL Support for LDRA Testbed” on page 34-16.

- BullseyeCoverage[™] from Bullseye Testing Technology[™]. For information about installing and using this tool, go to www.bullseye.com/cgi-bin/mwEval.

The software supports BullseyeCoverage code coverage for SIL and, in certain cases, PIL. For information about PIL support, see “PIL Support for BullseyeCoverage” on page 34-17.

For more information, see “Code Coverage Tool Support” on page 34-15.

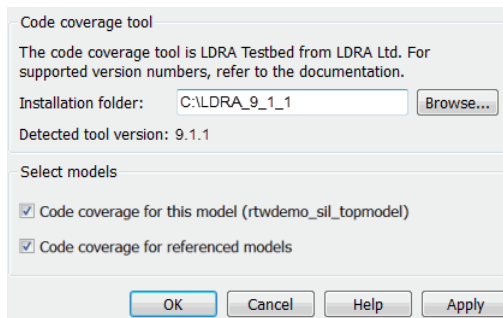
For information about setting up code coverage and viewing results, see:

- “Configure SIL and PIL Code Coverage” on page 34-3
- “Configure Code Coverage Programmatically” on page 34-8
- “View Code Coverage Information at the End of SIL or PIL Simulations” on page 34-5

Configure SIL and PIL Code Coverage

To configure a code coverage tool for a “top-model or Model block SIL or PIL simulation”:

- 1 Select **Simulation > Model Configuration Parameters > Code Generation > Verification**.
- 2 From the **Code coverage tool** drop-down list, select a tool, for example, **BullseyeCoverage** or **LDRA Testbed**.
- 3 Click **Configure Coverage** to open the Code Coverage Settings dialog box.
- 4 In the **Installation folder** field, specify the location where your coverage tool is installed. If you click **Browse**, the Select Installation Folder dialog box opens, which allows you to navigate to the folder where your coverage tool is installed. The software detects and displays the tool version.



By default, the software selects the following check boxes:

- **Code coverage for this model** — Generate coverage data for the current (top) model.
- **Code coverage for referenced models** — Generate data for models referenced by the current (top) model.

If your top model has Model blocks where the **Code under test** block parameter is set to **Top model**, then the top model and referenced models must have the same settings for these parameters. Otherwise, the software produces an error.

- 5 Click **OK**. You return to the **Verification** pane.
- 6 To view cumulative code coverage results within a code generation report, in the **Configuration Parameters > Code Generation > Report** pane, select the following check boxes:

- **Create code generation report**
 - **Launch report automatically**
- 7** Click **OK**. You return to the model window.

With LDRA Testbed:

- The evaluation of cumulative code coverage begins from the point when you last added a new file to the existing set of source files. For example, existing code coverage results are deleted when you:
 - Run a simulation with a new model using the existing code generation folder.
 - Run a simulation that results in additional source code files being instrumented.
- If you switch between SIL and PIL simulations of a model, the software generates separate cumulative code coverage results for the SIL and PIL simulations.

For a model in a reference hierarchy, the software does not support simultaneous function execution time measurement and code coverage.

For an example of code coverage metrics collection, see `rtwdemo_code_coverage_script`.

View Code Coverage Information at the End of SIL or PIL Simulations

If you specify code coverage for a SIL or PIL simulation, when the simulation is complete, the code generation report opens automatically and you see hyperlinks in the Command Window.

If you specified the LDRA Testbed, you see three links in the Command Window:

```
### Starting SIL simulation for component: rtwdemo_sil_topmodel
### Stopping SIL simulation for component: rtwdemo_sil_topmodel
### Starting analysis of coverage data
### Use the following links to view code coverage results:
    LDRA Testbed GUI
    LDRA Testbed Code Coverage Overview Report
    HTML code generation report with code coverage annotations
### Completed code coverage analysis
>>
```

To:

- Go to the LDRA Testbed GUI, click the first link.
- Open the LDRA Testbed Report with your Web browser, click the second link.

LDRA Testbed® Dynamic Overview Report

Set : work3_3afef0b64dc51060

Report Production	Report Configuration
<ul style="list-style-type: none"> • C/C++ LDRA Testbed Version: 8.5.1 	<ul style="list-style-type: none"> • DO-178B Level: 'a' • Report Format: Procedure Listing • Procedure Sort Method: Source File order • Reporting Scope: Source file and associated header

Contents

Combined Coverage for Selected Metrics

- Statement
- Branch/Decision
- Modified Condition/ Decision

[Table of Coverage Metric Pass Levels](#)

[Key to Terms](#)

For information about using this report, refer to the LDRA Testbed documentation.

- View summary data and code annotations with coverage information in the code generation report, click the third link. See “Code Coverage Summary and Annotations” on page 34-10.

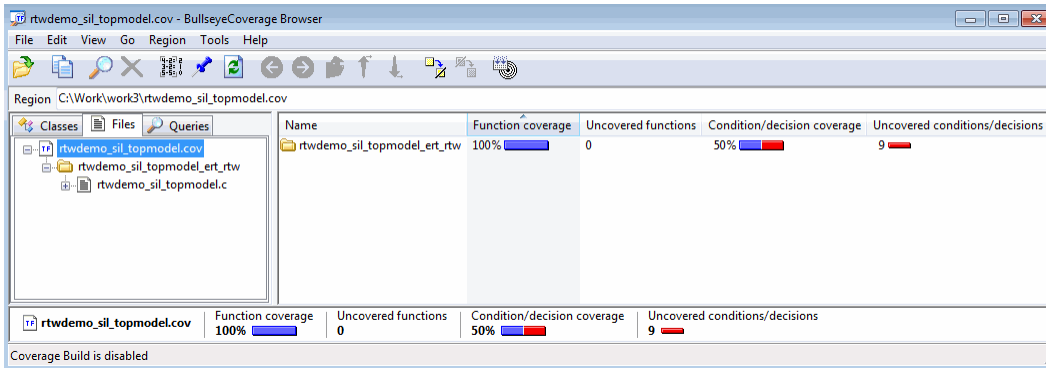
If you specified the BullseyeCoverage tool, you see two links in the Command Window:

```
### Starting SIL simulation for component: rtwdemo_sil_topmodel
### Stopping SIL simulation for component: rtwdemo_sil_topmodel
### Processing code coverage data
### Use the following links to view code coverage results:
    BullseyeCoverage browser (coverage for last run)
    HTML code generation report (cumulative coverage)
```

```
### Completed code coverage analysis
>>
```

To:

- View the coverage report using the BullseyeCoverage Browser, click the first link.



The BullseyeCoverage Browser shows coverage data for instrumented files associated with your latest top-model simulation. The coverage data shown in the browser is not cumulative and pertains only to the most recent simulation. For information about the BullseyeCoverage Browser, go to www.bullseye.com.

- View summary data and code annotations with coverage information in the code generation report, click the second link. See “Code Coverage Summary and Annotations” on page 34-10.

Configure Code Coverage Programmatically

You can configure code coverage for your model using command-line APIs. A typical workflow with BullseyeCoverage is:

- 1 Using `get_param`, retrieve the object containing coverage settings for the current model, for example, `gcs`.

```
>> covSettings = get_param(gcs, 'CodeCoverageSettings')

covSettings =

  cov.CodeCoverageSettings handle
  Package: cov

  Properties:
    TopModelCoverage: 'on'
    ReferencedModelCoverage: 'off'
    CoverageTool: 'BullseyeCoverage'

  Methods, Events, Superclasses
```

The property `TopModelCoverage` determines whether the software generates code coverage data for just the top model, while `ReferencedModelCoverage` determines whether the software generates coverage data for models referenced by the top model. If neither property is 'on', then no code coverage data is generated during a SIL simulation.

If LDRA Testbed is the specified code coverage tool, then the property `CoverageTool` is 'LDRA Testbed'.

When you save your model, the properties `TopModelCoverage`, `ReferencedModelCoverage`, and `CoverageTool` are also saved.

- 2 Check the class of `covSettings`.

```
>> class(covSettings)

ans =

  cov.CodeCoverageSettings
```

- 3 Turn on coverage for referenced models.

```
>> covSettings.ReferencedModelCoverage='on';
```

- 4 Using `set_param`, apply the new coverage settings to the model.

```
>>set_param(gcs, 'CodeCoverageSettings', covSettings);
```

- 5** Assuming you have installed the BullseyeCoverage tool, specify the installation path.

```
>> cov.BullseyeCoverage.setPath('C:\Program Files\BullseyeCoverage')
```

For LDRA Testbed, use `cov.LDRA.setPath('C:\...')`.

- 6** Check that the path is saved as a preference.

```
>> cov.BullseyeCoverage.getPath
```

For LDRA Testbed, use `cov.LDRA.getPath`.

Code Coverage Summary and Annotations

If you specify a code coverage tool for a SIL or PIL simulation (see “Configure SIL and PIL Code Coverage” on page 34-3), the software produces a code generation report that provides summary data and code annotations with coverage information. Each code annotation is associated with a code feature and indicates the nature of the feature coverage during code execution. For more information, see:

- “LDRA Testbed Coverage” on page 34-10
- “BullseyeCoverage Information” on page 34-12

The code generation report also allows you to navigate easily between blocks in your model and the corresponding sections in the source code. For more information, see “Trace Model Objects to Generated Code” and “Trace Code to Model Objects Using Hyperlinks”.

LDRA Testbed Coverage

The cumulative coverage data in a code generation report is derived from instrumented files associated with your latest top-model simulation **and** coverage data collected from simulations with other top models that share referenced models with your current top model.

The screenshot displays a code generation report for the file `rtwdemo_sil_topmodel.c`. The report includes a navigation pane on the left with sections for Contents, Generated Files, and Interface files. The main content area shows the LDRA Testbed code coverage enabled status and a summary of coverage metrics:

Metric	Value
Function exit points	100%
Statement	87%
Branch/condition	37%
Branch/decision	57%
MC/DC	0%

The source code snippet shows the following annotations:

```

1  /*
2  * File: rtwdemo_sil_topmodel.c
3  *
4  * Code generated for Simulink model 'rtwdemo_sil_topmodel'.
5  *
6  * Model version           : 1.206
7  * Simulink Coder version  : 8.2 (R2012a) 07-Oct-2011
8  * TLC version            : 8.2 (Oct 7 2011)
9  * C/C++ source code generated on : Wed Oct 19 12:29:12 2011
10 *
11 * Target selection: ert.tlc
12 * Embedded hardware selection: Specified
13 * Code generation objectives: Unspecified
14 * Validation result: Not run
15 */
16
17 #include "rtwdemo_sil_topmodel.h"
18
19 /* Block signals and states (auto storage) */
20 #ifndef rtWorK
21

```

The software provides LDRA Testbed annotations in the code generation report to help you to review code coverage.

Note: Do not use the code generation report alone to verify that you have achieved your coverage goals. You must refer to the LDRA Testbed Report. See “View Code Coverage Information at the End of SIL or PIL Simulations” on page 34-5.

This example shows three kinds of annotations. On lines 134, 139, 140, and 141, the annotation **•** indicates that statement coverage for each of these lines of code is not complete.

```

• 134   if (rtWork.bitsForTID0.LogicalOperator1) {
=>b   .....
135   /* Switch: 'cS2>/Switch1' incorporates:
136   * Constant: 'cS2>/C1'
137   * Import: 'cRoot>/reset'
138   */
• 139   if (xll.reset) {
=>   .....
140   rtWork.PreviousOutput_DSTATE = 00;
• 141   }

```

Placing the cursor over the annotation =>b produces a tooltip.

```

• 134   if (rtWork.bitsForTID0.LogicalOperator1) {
=>b   .....
      |-----|
      | Branch destinations |
      | covered              |
      | line<<column>:134:3  |
      | branch destinations |
      | not                  |
      | covered:134c45     |
      |-----|
      | [L.reset] (

```

This tooltip indicates that only one branch destination is covered. The code within the curly brackets, which starts at column 45 of line 134, is not executed. As the `if` statement on line 139 lies within this code, the corresponding annotation `=>` states that the branch is not covered.

The following table describes the LDRA Testbed code annotations that you might see in a code generation report produced by a SIL and PIL simulations.

Code feature	Annotation symbol	What happened during simulation
Function	Fcn	Function <i>name</i> returned through this exit point.
	=>	Function <i>name</i> never returned through this exit point.
Branch/condition	=>	Condition not encountered.
	=>t	Condition evaluated true only.
	=>f	Condition evaluated false only.

Code feature	Annotation symbol	What happened during simulation
	tf	Condition evaluated both true and false.
Branch/decision	=>	Branch never encountered.
	=>b	Branch to at least one destination covered and branch to at least one other destination not covered.
	b	Branch fully exercised.
Modified Condition/ Decision Coverage (MC/DC)	=>mc	Condition did not independently affect outcome of decision.
	mc	Condition independently affected outcome of decision.
Statement	▪	Statements associated with line covered.
	•	Not all statements associated with line covered.
Code that is reformatted by LDRA Testbed and does not match the original source code. For example, source code with <code>#include</code> statements to include other files, and source code with <code>#define</code> statements for macros. For detailed coverage information, refer to the LDRA Testbed report.	=>Σ	Zero coverage — probes within source code line or files included by source code line not exercised.
	=>Σ	Coverage probes within source code line or any included file partially exercised.
	Σ	Coverage probes within source code line or included files fully exercised.

BullseyeCoverage Information

The cumulative coverage data in a code generation report is derived from instrumented files associated with your latest top-model simulation **and** coverage data collected from

simulations with other top models that share referenced models with your current top model.

```

File: rtwdemo_sil_topmodel.c
BullseyeCoverage code coverage enabled
Function: 100% Condition/decision: 50%
1 /*
2  * File: rtwdemo_sil_topmodel.c
3  *
4  * Code generated for Simulink model 'rtwdemo_sil_topmodel'.
5  *
6  * Model version      : 1.208
7  * Simulink Coder version : 8.2 (R2012a)
8  * TLC version        : 8.2 (Oct 7 2011)
9  * C/C++ source code generated on : Tue Oct 18 11:33:01 2011
10 *
11 * Target selection: ert.tlc
12 * Embedded hardware selection: Specified
13 * Code generation objectives: Unspecified
14 * Validation result: Not run
15 */
16
17 #include "rtwdemo_sil_topmodel.h"
18
19 /* Block signals and states (auto storage) */
20 P_Work rtDWork;
21
22 /* External inputs (root inport signals with auto storage) *

```

The software provides BullseyeCoverage annotations in the code generation report to help you to review code coverage.

This example shows two kinds of annotations. At line 41, TF indicates that the `if` decision had both true and false outcomes during the simulation. At line 52, `=>F` indicates that the `if` decision was false only during the simulation.

```

TF 41  if (rtU.reset) {
42      rtDWork.PreviousOutput_DSTATE = 20U;
43  }
44
45  /* Switch: '<Root>/Switch' incorporates:
46   * Constant: '<Root>/C1'
47   * Constant: '<Root>/C5'
48   * Inport: '<Root>/ ticks_to_count'
49   * RelationalOperator: '<Root>/upper_GE input'
50   * Sum: '<Root>/Add'
51   */
=>F 52  if ((uint8_T) ((uint32_T)rtU.ticks_to_count + (uint32
53      rtDWork.PreviousOutput_DSTATE) == 40)

```

The following table describes the BullseyeCoverage code annotations that you might see in a code generation report produced by a SIL simulation.

Code feature	Annotation symbol	What happened during simulation
Decision	=>	Decision not executed.
	TF	Decision evaluated both true and false.

Code feature	Annotation symbol	What happened during simulation
	=>T	Decision evaluated true only.
	=>F	Decision evaluated false only.
Function	=>	Function not called.
	Fcn	Function called.
Switch label	=>	Switch command not used.
	Sw	Switch command used.
Constant	k	Decision or condition was constant, which did not allow any variation in coverage.
Condition	=>	Condition not encountered.
	tf	Condition evaluated both true and false.
	=>t	Condition evaluated true only.
	=>f	Condition evaluated false only.
Try	=>	Try block never completed.
	Try	Try block covered.
Catch	=>	Catch block not covered.
	Cat	Catch block covered.

Code Coverage Tool Support

Embedded Coder is tested with the following versions of the BullseyeCoverage and LDRA Testbed tools.

Operating system	BullseyeCoverage	LDRA Testbed
Windows	8.4.23 ^a	9.3.0
Linux	8.4.19	Not supported
Mac	Not supported	Not supported

- a. BullseyeCoverage is not supported with LCC compiler.

Code Coverage for PIL

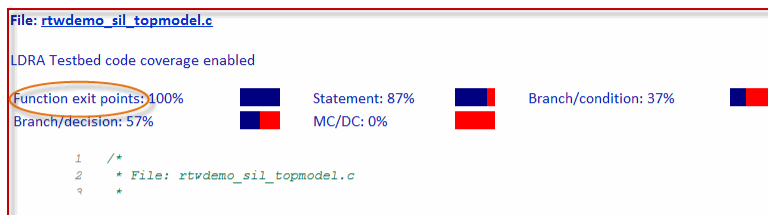
PIL Support for LDRA Testbed

The target connectivity API supports code coverage with LDRA Testbed for top-model and Model block PIL.

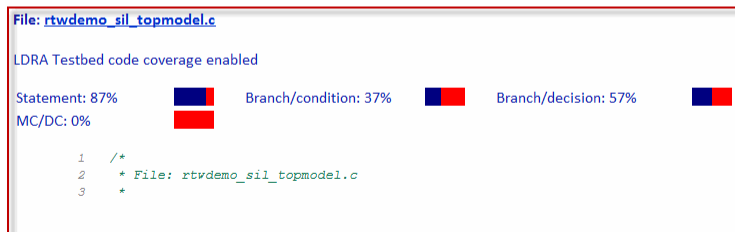
For LDRA Testbed version 9.1.1, MathWorks instrumentation files are located in the LDRA Testbed installation folder, for example:

- C:\LDRA_Toolsuite\Compiler_spec\MathWorks\MathWorks_Cinstr.DAT
- C:\LDRA_Toolsuite\Compiler_spec\MathWorks\MathWorks_CPPINSTR.DAT

There are minor differences in the code coverage information collected during SIL and PIL simulations. In particular, with PIL, the software does not explicitly show function exit point coverage. However, you can infer the coverage of function exit points by examining statement coverage.



SIL



PIL

PIL Support for BullseyeCoverage

Code coverage with BullseyeCoverage is available for top-model and Model block PIL provided your PIL application can write directly to the host file system. Your target for the PIL application must provide `fopen` and `fread` access to the host file system.

If code coverage is not available when you run the PIL application on your target hardware, you might be able to collect code coverage measurements by running the PIL application on an instruction set simulator that supports direct file I/O with the host file system.

Tips and Limitations

Compiler and Platform Support for SIL

For SIL code coverage, the software supports the following compilers and platforms:

- On a Linux platform, gcc
- On a Windows platform, Microsoft Visual C++ (MSVC)

Note: For both SIL and PIL, the Watcom compiler is not supported. If you specify this compiler, an error appears when you build your model.

For information on how to specify a compiler, see “Compiler or IDE Selection and Configuration”.

Right-Click Subsystem Build Unsupported for Code Coverage

The software does not support right-click builds for subsystems if a code coverage tool is specified.

BullseyeCoverage License Wait

When you build your model, you might have to wait for a BullseyeCoverage license. If you want to see information about the wait, before you build your model, select **Code Generation > Debug > Verbose build**.

Current Working Folder Cannot be UNC Path

If your MATLAB current working folder is a Universal Naming Convention (UNC) path, code coverage fails.

Characters in matlabroot and File Path

If `matlabroot` or the path to your generated files contains a space or the . (period) character, code coverage might fail.

Header Files with Identical Names

Consider a model that is configured for LDRA Testbed code coverage. During the build process, if the software detects two header files with the same name in the folder for generated code, the software generates an error.

Code Coverage for Source Files in Shared Utility Folders

The software supports code coverage for source files generated in shared utility folders. If you configure code coverage for a model that uses shared utility code generation, when you build the model, you also build all source files in the shared utilities folder with code coverage enabled.

Whenever you build a model, the code coverage settings of the model must be consistent with source files that you previously built in the shared utilities folder. Otherwise, the software reports that code in the shared utilities folder is inconsistent with the current model configuration and must be rebuilt. For example, if you run a SIL simulation for a model with code coverage enabled and then run a SIL simulation for another model with code coverage disabled, the software must rebuild all source files in the shared utilities folder.

BullseyeCoverage Behavior with Inline Macros

The BullseyeCoverage tool, by default, does not provide code coverage data for inline macros.

For example, if a model generates a file `slprj/ert/_sharedutils/rt_SATURATE.h` that contains the macro

```
#define rt_SATURATE(sig,ll,ul) (((sig) >= (ul)) ? (ul) : (((sig) <= (ll)) ? (ll) : (sig)) )
```

and the macro is in `sat_ert_rtw/sat.c`, then the coverage report provides a measurement for `sat.c`, but no coverage data for the conditions within the macro `rt_SATURATE`.

To configure the BullseyeCoverage tool to provide code coverage data for inline macros:

- 1 Open the BullseyeCoverage Browser.
- 2 Select **Tools > Options** to open the Options dialog box.
- 3 On the **Build** tab, select the **Instrument macro expansions** check box.

- 4 Click **OK**.
- 5 Rerun your simulation.

Alternatively, you can add the text `-macro` to the `BullseyeCoverage` configuration file. For more information, go to www.bullseye.com/help/ref_covc.html.

SIL and PIL Simulations with Open LDRA Testbed

If you enable code coverage with the LDRA Testbed tool, you must verify that the LDRA Testbed GUI is not open when you run your SIL or PIL simulation. If the set name in the LDRA Testbed GUI differs from the set name used by the SIL or PIL simulation, the SIL or PIL simulation fails.

PIL Zero Coverage LDRA Testbed Annotations

For a PIL simulation with LDRA Testbed code coverage specified, there might be some source files where the recorded coverage is zero. In this case, the software provides summary information indicating that:

- There is coverage to measure.
- The coverage is zero.

You do not see information for individual probes on each line. The displayed summary information has an associated annotation tooltip:

```
0 out of N coverage probes were exercised (detailed breakdown unavailable)
```

Modify Legacy Code

If you modify legacy code and rerun a SIL or PIL simulation, the legacy code is recompiled. However, the code from the model may be up-to-date. In this case, the code generation report is not updated and does not show the modified legacy code. Instead, the code coverage information for the modified legacy code is displayed with reference to the original legacy code. You must regenerate the report. For more information, see “Limitation”.

IDE Link Does Not Support LDRA Testbed

When you generate code for IDE Link, you cannot use LDRA Testbed for SIL or PIL code coverage. Specifically, this limitation applies when you use the following settings together:

- **Configuration Parameters > Code Generation > System target file:**
idelink_ert.tlc
- **Configuration Parameters > Code Generation > Verification > Code coverage tool:** LDRA Testbed.

Embedded IDEs and Embedded Targets

Getting Started with Embedded Targets

Embedded Coder Supported Hardware

As of this release, Embedded Coder supports the following hardware.

Support Package	Vendor	Platforms	Earliest Release Available	Last Release Available
“Altera SoC”	Altera®	Windows	R2014b	Current
“Analog Devices DSPs ”	Analog Devices™	Windows	R2013a	Current
“ARM Cortex-A Processors”	ARM®	Windows	R2014a	Current
“ARM Cortex-M Processors”	ARM	Windows	R2013b	Current
“BeagleBone Black Hardware”	BeagleBoard	Windows	R2014b	Current
“Freescale FRDM-KL25Z Board”	Freescale™	Windows	R2014b	Current
Green Hills MULTI	Green Hills® Software	Windows, Linux	R2012b	R2014a
“STMicroelectronics STM32F4-Discovery Board”	STMicroelectronics	Windows	R2013b	Current
“Texas Instruments C2000 Processors”	Texas Instruments	Windows	R2013b	Current
“Texas Instruments C2000 F28M3x Concerto Processors”	Texas Instruments	Windows	R2014b	Current
“Texas Instruments C6000 DSPs”	Texas Instruments	Windows	R2014a	Current
“Wind River VxWorks RTOS”	Wind River	Windows, Linux	R2013b	Current
“Xilinx Zynq-7000 Platform”	Xilinx®	Windows	R2013a	Current

For a complete list of supported hardware, see Hardware Support.

Project and Build Configurations for Embedded Targets

- “Model Setup” on page 36-2
- “IDE Projects” on page 36-13
- “Makefiles for Software Build Tool Chains” on page 36-15

Model Setup

In this section...

“Block Selection” on page 36-2

“Configure Target Hardware Resources” on page 36-3

“Configuration Parameters” on page 36-5

“Model Reference” on page 36-12

Block Selection

You can create models for targeting the same way you create other Simulink models—by combining standard blocks and C-MEX S-functions.

You can use blocks from the following sources:

- The Embedded Coder Support Packages.
- The Embedded Targets library (`embeddedtargetslib`) in the Embedded Coder product.
- Blocks from the System Toolboxes products
- Custom blocks

Avoid using blocks that do not generate code, including the following blocks.

Block Name/Category	Library	Description
Scope	Simulink, DSP System Toolbox software	Provides oscilloscope view of your output. Do not use the Save data to workspace option on the Data history pane in the Scope Parameters dialog.
To Workspace	Simulink	Return data to your MATLAB workspace.
From Workspace	Simulink	Send data to your model from your MATLAB workspace.
Spectrum Scope	DSP System Toolbox	Compute and display the short-time FFT of a signal. It has internal

Block Name/Category	Library	Description
		buffering that can slow your process without adding value.
To File	Simulink	Send data to a file on your host machine.
From File	Simulink	Get data from a file on your host machine.
Triggered to Workspace	DSP System Toolbox	Send data to your MATLAB workspace.
Signal To Workspace	DSP System Toolbox	Send a signal to your MATLAB workspace.
Signal From Workspace	DSP System Toolbox	Get a signal from your MATLAB workspace.
Triggered Signal From Workspace	DSP System Toolbox	Get a signal from your MATLAB workspace.
To Wave device	DSP System Toolbox	Send data to a .wav device.
From Wave device	DSP System Toolbox	Get data from a .wav device.

Configure Target Hardware Resources

This topic contains the following subtopics:

- “About Supported IDEs” on page 36-3
- “Configure Parameters Under the Target Hardware Resources Tab” on page 36-4

About Supported IDEs

This “Configure Target Hardware Resources” on page 36-3 section applies to the following IDEs:

- Analog Devices VisualDSP++®
- Texas Instruments Code Composer Studio™ 3.3
- Texas Instruments Code Composer Studio 4 (makefile generation only)
- Texas Instruments Code Composer Studio 5 (makefile generation only)
- Wind River Diab/GCC (makefile generation only)


Configure Parameters Under the Target Hardware Resources Tab

Configure the parameters under the Target Hardware Resources tab of your Simulink model for a specific tool chain and target hardware. Doing so updates other parameters in the Configuration Parameters dialog to the default values for the software build tool chain and target hardware you are using.

Note: The Target Preferences (Removed) block has been removed from the Simulink block libraries for the Embedded Coder and Simulink Coder products.

Parameters in the Target Preferences block have been moved to the Target Hardware Resources tab.

To configure your Simulink model for a specific tool chain and target hardware:

- 1 In a Simulink model, open the model Configuration Parameters by:
 - Clicking the gear icon,

 - Pressing **Ctrl+E** on your keyboard
 - Selecting the **Simulation > Model Configuration Parameters** menu items
- 2 In the Configuration Parameters dialog, click **Code Generation**, and then click “+” next to Code Generation. This action displays the sub-panes under Code Generation.
- 3 On the Code Generation pane, change **System target file** to `idmlink_ert.tlc` or `idmlink_grt.tlc`.

The dialog displays a **Coder Target** pane under the Code Generation pane.

- 4 Select the **Coder Target** pane.
- 5 Select the **Target Hardware Resources** tab.
- 6 Set the following parameters to match the tool chain and target hardware you are using:
 - **IDE/Tool Chain**
 - **Board**
 - **Processor**

- 7 Review the other parameters under the **Target Hardware Resources** tab.
- 8 Click **Apply**, and save the changes to your model.

Configuration Parameters

- “What are Configuration Parameters?” on page 36-5
- “Setting Model Configuration Parameters” on page 36-5

What are Configuration Parameters?

To see the model Configuration Parameters, open the **Configuration Parameters** dialog. You can do this in the model editor by selecting **Simulation > Model Configuration Parameters**, or by pressing **Ctrl+E** on your keyboard.

The **Configuration Parameters** dialog specifies the values for a model's active *configuration set*. These parameters determine the type of solver used, the import and export settings, and other values that determine how the model runs.

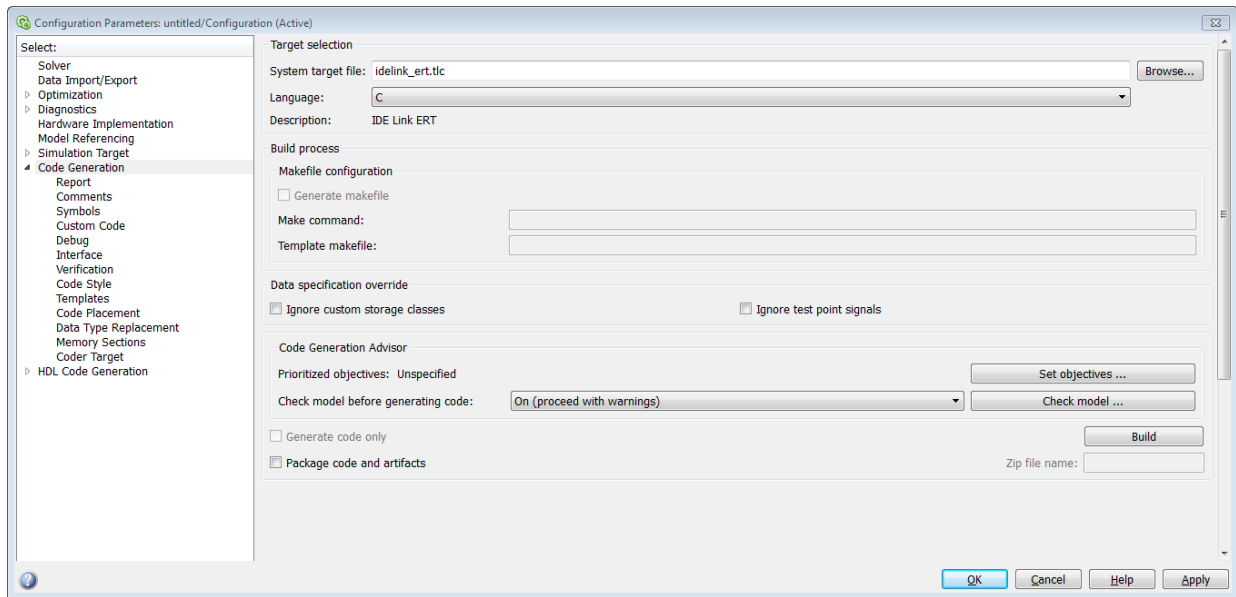
Setting Model Configuration Parameters

To set the Configuration Parameters to the right values for you to generate code from your model, see “Configure Parameters Under the Target Hardware Resources Tab” on page 36-4. This action initializes the model Configuration Parameters to the right default values for you to generate code. You can then use the Configuration Parameters dialog to make further modifications to the values. You can generate buildable code using these default values.

The following subsections provide a quick overview of the panes and parameters with which you are most likely to interact.

Code Generation Pane

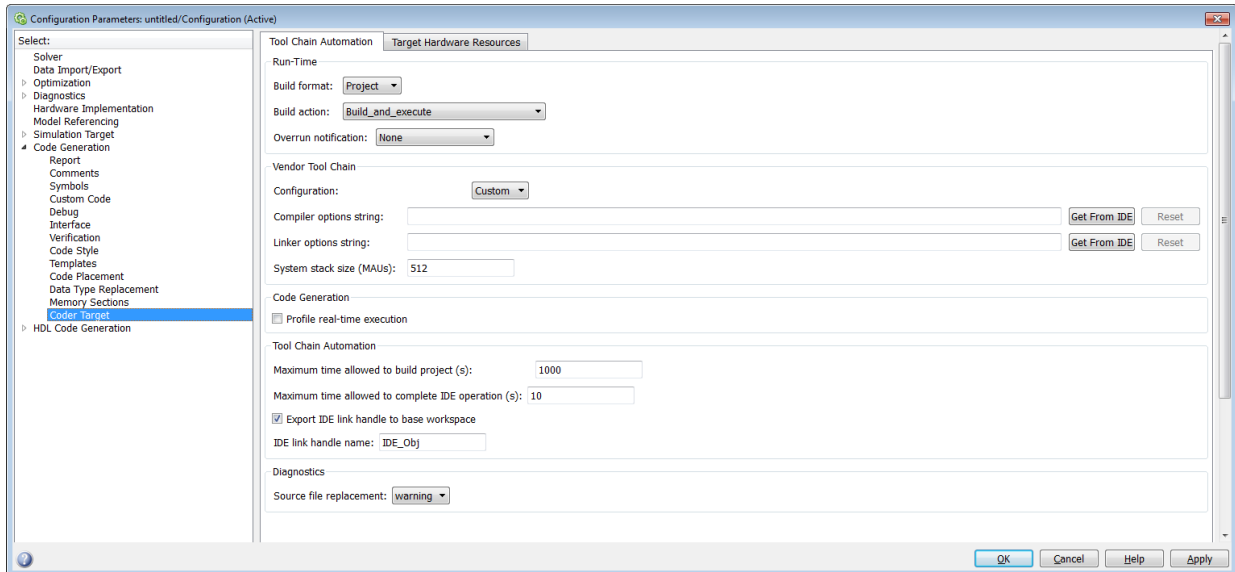
When you set **System target file** to `idmlink_ert.tlc` or `idmlink_grt.tlc`, the dialog adds an **Coder Target** pane to the list of panes under **Code Generation**.



Leave **Language** set to **C**. The `idelink_ert.tlc` and `idelink_grt.tlc` system target files do not support C++ code generation.

For more information, see “Code Generation Pane: General”

Coder Target Pane Parameters



The Coder Target entry provides options in these areas:

- **Run-Time** — Set options for run-time operations, like the build action
- **Vendor Tool Chain** — Set compiler, linker, and system stack size options
- **Code Generation** — Configure your code generation requirements
- **Link Automation** — Export an IDE link handle object, such as `IDE_Obj`, to your MATLAB workspace
- **Diagnostics** — Determine how the code generation process responds when you use source code replacement in the **Custom Code** pane.

For more information, see “Code Generation Pane: Coder Target”.

Build format

Select **Project** to create an IDE project, or select **Makefile** to create a makefile build script.

Build action

Your selection for **Build action** determines what happens when you click **Build** or press **Ctrl+B**. Your selection tells Simulink Coder software when to stop the code generation and build process.

To run your model on the processor, select **Build_and_execute**. This selection is the default build action.

The actions are cumulative—each action performs an additional step relative to the preceding action on the list.

If you set **Build format** to **Project**, select one of the following options:

- **Create_project** — Directs Simulink Coder software to start the IDE and populate a new project with the files from the build process. This option offers a convenient way to build projects in the IDE.
- **Archive_library** — Directs Simulink Coder software to create an archive library for this model. Use this option when you plan to use the model in a model reference application. Model reference requires that you archive your the IDE projects for models that you use in model referencing.
- **Build** — Builds the executable file, but does not download the file to the target hardware.
- **Build_and_execute** — Directs Simulink Coder software to build, download, and run your generated code as an executable on your target hardware.
- **Create_processor_in_the_loop_project** — Directs code generation process to create PIL algorithm object code as part of the project build. This option requires an Embedded Coder license.

If you set **Build format** to **Makefile**, select one of the following options:

- **Create_makefile** — Creates a makefile.
- **Archive_library** — Creates a makefile and the generated output will be an archive library.
- **Build** — Creates a makefile and an executable.
- **Build_and_execute** — Creates a makefile and an executable. Then it evaluates the execute instruction in the current configuration.

Overrun notification

To enable the overrun indicator, choose one of three ways for the target to respond to an overrun condition in your model:

- **None** — Ignore overruns encountered while running the model.
- **Print_message** — When the target encounters an overrun condition, it prints a message to the standard output device, `stdout`.
- **Call_custom_function** — Respond to overrun conditions by calling the custom function you identify in **Function name**.

Function name

When you select `Call_custom_function` from the **Overrun notification** list, you enable this option. Enter the name of the function the target should use to notify you that an overrun condition occurred. The function must exist in your code on the target hardware.

Configuration

The **Configuration** parameter defines sets of build options that apply to the files generated from your model.

The **Release** and **Debug** option apply build settings that are defined by your compiler. For more information, refer to your compiler documentation.

Custom has the same default values as **Release**, but:

- Leaves **Compiler options string** empty.

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 coder product does not set optimization flags.

With Texas Instruments Code Composer Studio 3.3 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**.

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 coder product does not set linker options.

With Texas Instruments Code Composer Studio 3.3 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 clear the linker options, click **Reset**.

System stack size (MAUs)

Enter the amount of memory that is available for allocating stack data, measured in minimum addressable units (MAU). Block output buffers are placed on the stack until the stack memory is fully allocated. After that, the output buffers go in global memory. An MAU is typically 1 byte, but its size can vary by target hardware.

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, this value specifies the stack space allocated per thread.

This parameter also applies to the “Maximum stack size (bytes)” parameter, located in the Optimization > Signals and Parameters pane.

System heap size (MAUs)

Set the default heap size that the target hardware reserves for dynamic memory allocation.

The target hardware 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)

Profile real-time execution

To enable the real-time execution profile capability, select **Profile real-time execution**. With this selected, the build process instruments your code to provide performance

profiling at the task level or for atomic subsystems. When you run your code, the executed code reports the profiling information in an HTML report.

Link Automation

When you build a model for a target, the coder product automatically creates or uses an existing *IDE link handle object* (named `IDE_Obj`, by default) to connect to your IDE.

Although `IDE_Obj` is a handle for a specific instance of the IDE, it also contains information about the IDE instance to which it refers, such as the target the IDE accesses. In this pane, the **Export IDE link handle to base workspace** option lets you instruct the coder product to export the object to your MATLAB workspace, giving it the name you assign in **IDE link handle name**.

You can also use the IDE link handle object to interact with the IDE using IDE Automation Interface commands.

Maximum time allowed to build project (s)

Specifies how long the software waits for the IDE to build the software.

Maximum time allowed to complete IDE operation (s)

Specifies how long the software waits for IDE functions, such as `read` or `write`, to return completion messages. If you do not specify a timeout, the default value is 10 seconds.

Export IDE link handle to base workspace

Directs the software to export the `IDE_Obj` object to your MATLAB workspace.

IDE link handle name

Specifies the name of the `IDE_Obj` object that the build process creates.

Source file replacement

Selects the diagnostic action to take if the software detects conflicts when you replace source code with custom code. The diagnostic message responds to both source file replacement in the Configuration Parameters under Code Generation > Coder Target parameters and under Code Generation > Custom Code.

The following settings define the messages you see and how the code generation process responds:

- `none` — Does not generate warnings or errors when it finds conflicts.
- `warning` — Displays a warning. `warn` is the default value.
- `error` — Terminates the build process and displays an error message that identifies which file has the problem and suggests how to resolve it.

The build operation continues if you select `warning` and the software detects custom code replacement problems. 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. Use `none` when the replacement process works and you do not want to see multiple messages during your build.

Model Reference

The `idelink_ert.tlc` and `idelink_grt.tlc` system target files provide support for generating code from models that use Model Reference. A referenced model will generate an archive library.

To enable Model Reference builds:

- 1 Open your referenced model.
- 2 Select Simulation > Model Configuration Parameters from the model menus.
- 3 From the list of panes under **Code Generation**, choose **Coder Target**.
- 4 In the right pane, under Run-Time, select `Archive_library` from the **Build action** list.

If your top-model uses a reference model that does not have the **Build action** set to `Archive_library`, the build process automatically changes the **Build action** to `Archive_library` and issues a warning about the change.

Configuration Parameters in Reference Models

Use the same Coder Target pane settings in Configuration Parameters for the models in the model hierarchy.

IDE Projects

In this section...

“Support for Third Party Products” on page 36-13
“Code Generation and Build” on page 36-13

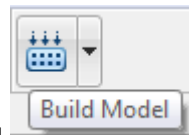
Support for Third Party Products

For more information about support for third-party IDEs and targets, see:

- Supported and Compatible Compilers
- Hardware Support

Code Generation and Build

Building Your Model



In your model, click **Build Model**. The software performs the actions you selected for **Build action** in the model Configuration Parameters, under Code Generation > Coder Target.

IDE Project Generator Features

The *IDE Project Generator* component provides or supports the following features for developing IDE projects and generating code:

- Automatically create IDE projects for your generated code during the code generation process.
- Customize code generation using options in the model Configuration Parameters.
- Configure the automatic project build process.
- Automatically download and run your generated projects on your target hardware.

IDE Link Handle Objects

IDE Project Generator automatically creates and uses an IDE link handle object to communicate with your IDE and target hardware.

To create the IDE link handle object, IDE Project Generator uses one of the following constructor functions:

- `adivdsp` for Analog Devices VisualDSP++
- `ticcs` for Texas Instruments Code Composer Studio

For a command line example of how to use a constructor function, see the corresponding reference page for each function.

Makefiles for Software Build Tool Chains

In this section...

- “What is the XMakefile Feature” on page 36-15
- “Using Makefiles to Generate and Build Software” on page 36-18
- “Making an XMakefile Configuration Operational” on page 36-21
- “Creating a New XMakefile Configuration” on page 36-21
- “XMakefile User Configuration dialog” on page 36-27

What is the XMakefile Feature

- “Overview” on page 36-15
- “Supported Tool Chains in Embedded Coder” on page 36-16
- “Available XMakefile Configurations” on page 36-16
- “Feature Support” on page 36-18

Overview

You can use makefiles instead of IDE projects during the automated software build process. This approach is described in “Using Makefiles to Generate and Build Software” on page 36-18.

The XMakefile feature lets you choose the *configuration* of a specific software build tool chain to use during the automated build process. The configuration contains paths and settings for your make utility, compiler, linker, archiver, pre-build, post-build, and execute tools.

You can choose one built-in configuration described in “Supported Tool Chains in Embedded Coder” on page 36-16 and “Available XMakefile Configurations” on page 36-16.

You can also create a new configuration for a new tool chain, as described in “Creating a New XMakefile Configuration” on page 36-21.

Your requirements for specific features may determine whether you choose makefiles or IDE projects. See “Feature Support” on page 36-18.

Supported Tool Chains in Embedded Coder

Embedded Coder includes support for the following IDEs and tool chains.

Tool Chain	Processor Family/Target Operating System	Host Operating System
Analog Devices VisualDSP++	Blackfin [®] , SHARC [®] , and TigerSHARC [®]	Windows
GNU development tools	Linux	Linux
GNU development tools	ARM	MontaVista Linux
Texas Instruments Code Composer Studio 3.3, 4, and 5.1	Texas Instruments C2000 [™] , C5500 [™] , and C6000 [™]	Windows
Wind River Diab/GCC	ARM9, Host Simulator, VxWorks, RTP and RTP_SO	Windows
Xilinx ISE Design Suite	ARM Cortex-A9 on Xilinx Zynq [®] -7000 running Linux	32-bit and 64-bit Windows

Simulink Coder includes support for other IDEs and tool chains. See .

Available XMakefile Configurations

The following list describes the configurations in the XMakefile dialog that this product supports:

- `adivdsp_blackfin`: Analog Devices VisualDSP++ & Analog Devices Blackfin
- `adivdsp_sharc`: Analog Devices VisualDSP++ & Analog Devices SHARC
- `adivdsp_tigersharc`: Analog Devices VisualDSP++ & Analog Devices TigerSHARC
- `gcc_target`: GNU Compiler Collection & Host Operating System or Embedded Operating System
- `ticcs_c2000_ccsv3`: Texas Instruments Code Composer Studio 3 & Texas Instruments C2000
- `ticcs_c2000_ccsv4`: Texas Instruments Code Composer Studio 4 & Texas Instruments C2000
- `ticcs_c2000_ccsv5`: Texas Instruments Code Composer Studio 5.1 & Texas Instruments C2000

- `ticcs_c5500_ccsv3`: Texas Instruments Code Composer Studio 3 & Texas Instruments C5500
- `ticcs_c5500_ccsv4`: Texas Instruments Code Composer Studio 4 & Texas Instruments C5500
- `ticcs_c5500_ccsv5`: Texas Instruments Code Composer Studio 5.1 & Texas Instruments C5500
- `ticcs_c6000_ccsv3`: Texas Instruments Code Composer Studio 3 & Texas Instruments C6000
- `ticcs_c6000_ccsv4`: Texas Instruments Code Composer Studio 4 & Texas Instruments C6000
- `ticcs_c6000_ccsv5`: Texas Instruments Code Composer Studio 5.1 & Texas Instruments C6000
- `ticcs_c6000_dsplibios_ccsv3`: Texas Instruments Code Composer Studio 3 & Texas Instruments DSP/BIOS on C6000
- `ticcs_c6000_dsplibios_ccsv4`: Texas Instruments Code Composer Studio 4 & Texas Instruments DSP/BIOS on C6000
- `ticcs_c6000_dsplibios_ccsv5`: Texas Instruments Code Composer Studio 5.1 & Texas Instruments DSP/BIOS on C6000
- `wrsdiab_arm9_vxworks67_rtp`: Wind River Systems DIAB Compiler & ARM 9 & VxWorks 6.7 & real-time process applications
- `wrsdiab_arm9_vxworks67_rtp_so`: Wind River Systems DIAB Compiler & ARM 9 & VxWorks 6.7 & real-time process applications with shared object
- `wrsdiab_hostsim_vxworks67_rtp`: Wind River Systems DIAB Compiler & VxWorks Host Simulator & VxWorks 6.7 & real-time process applications
- `wrsdiab_hostsim_vxworks67_rtp_so`: Wind River Systems DIAB Compiler & VxWorks Host Simulator & VxWorks 6.7 & real-time process applications with shared object
- `wrsdiab_hostsim_vxworks68_rtp`: Wind River Systems DIAB Compiler & VxWorks Host Simulator & VxWorks 6.8 & real-time process applications
- `wrsdiab_hostsim_vxworks68_rtp_so`: Wind River Systems DIAB Compiler & VxWorks Host Simulator & VxWorks 6.8 & real-time process applications with shared object
- `wrsgnu_arm9_vxworks67_rtp`: Wind River Systems GNU Compiler & VxWorks Host Simulator & VxWorks 6.7 & real-time process applications

- `wrsgnu_hostsim_vxworks67_rtp`: Wind River Systems GNU Compiler & VxWorks Host Simulator & VxWorks 6.7 & real-time process applications with shared object
- `wrsgnu_hostsim_vxworks68_rtp`: Wind River Systems GNU Compiler & VxWorks Host Simulator & VxWorks 6.8 & real-time process applications with shared object
- `xilinx_ise_14_x`: Xilinx ISE Design Suite & ARM Cortex-A9 running Linux on Xilinx Zynq-7000 platform

For more information about supported versions of third-party software, see “Support for Third Party Products” on page 36-13

Feature Support

With makefiles, you cannot use features that rely on direct communications between your MathWorks software and third-party IDEs.

You cannot use the following features with makefiles:

- IDE Project Generation
- IDE Automation Interface
- IDE debugger communications during Processor-in-the-loop (PIL) simulation

Using Makefiles to Generate and Build Software

In addition to this chapter, see the Makefile Generator Tutorial example for more information about using makefiles to generate code.

Configuring Your Model to Use Makefiles

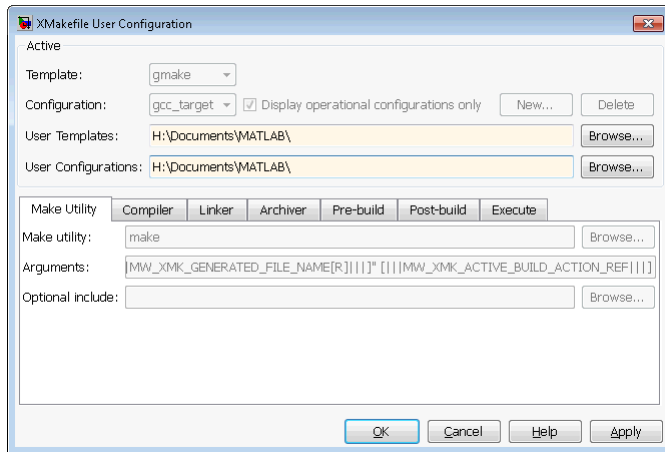
Update your model Configuration Parameters to use a makefile instead of an IDE when you build software from the model:

- 1 Configure your model for your IDE, tool chain, and target hardware, as described in “Configure Target Hardware Resources” on page 36-3.
- 2 In the Configuration Parameters dialog, under the **Code Generation** tab, select **Coder Target**.
- 3 Set **Build format** to **Makefile**. For more information, see “Build format” on page 36-7.
- 4 Set **Build action** to **Build_and_execute**. For more information, see “Build action” on page 36-8.

Choosing an XMakefile Configuration

Configure how to generate makefiles:

- 1 Enter `xmakefilessetup` on the MATLAB Command Window. The software opens an XMakefile User Configuration dialog.



- 2 Set the **Template** parameter to the option that matches the **Configuration** parameter.

Note: In most cases, the only option for **Template** is `gmake`. However, if you have installed a Support Package, **Template** can have multiple options.

- 3 For **Configuration**, select the option that describes your software build toolchain and target platform. Click **Apply**.

Note: Changing some elements of the XMakefile dialog disables other elements until you apply the changes. Click **Apply** or **OK** after changing:

- **Template**
- **Configurations**
- **User Templates**
- **User Configurations**

- **Tool Directories**

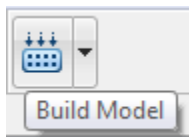
Note: With the XMakefile User Configuration dialog, if you have an Embedded Coder license and do not have a Simulink Coder license, the **Configuration** list includes two unsupported options: `gcc_target` or `msvs_host`. Disregard those two configurations. Choose one of the other configurations.

Things to consider while setting **Configuration**:

- Selecting **Display operational configurations only** hides configurations that contain incomplete or invalid information. For a configuration to be operational, the vendor tool chain must be installed, and the configuration must have the valid paths for each component of the vendor tool chain. For more information, see “Making an XMakefile Configuration Operational” on page 36-21.
- To display the configurations, including non-operational configurations, clear **Display operational configurations only**.
- The list of configurations can include non-editable configurations defined in the software and editable configurations defined by you.
- To create a new editable configuration, use the **New** button.
- For more information, see “XMakefile User Configuration dialog” on page 36-27.

Building Your Model

In your model, click **Build Model**.



This action creates a makefile and performs the other actions you specified in **Build action**.

By default, this process outputs files in the `<builddir>/<buildconfiguration>` folder. For example, in `model_name/CustomMW`.

Making an XMakefile Configuration Operational

When the XMakefile utility starts, it checks each configuration file to verify that the specified paths for the vendor tool chain are valid. If the paths are not valid, the configuration is non-operational. Typically, the cause of this problem is a difference between the path in the configuration and the actual path of the vendor toolchain.

To make a configuration operational:

- 1 Clear **Display operational configurations only** to display non-operational configurations.
- 2 Select the non-operational configuration from the **Configuration** options.
- 3 When you click **Apply**, a new dialog prompts you for the folder path of the missing resources the configuration requires.

Use mapped network drives instead of UNC paths to specify directory locations. Using UNC paths with compilers that do not support them causes build errors.

Creating a New XMakefile Configuration

- “Overview” on page 36-21
- “Create a Configuration” on page 36-21
- “Modify the Configuration” on page 36-23
- “Test the Configuration” on page 36-25

Overview

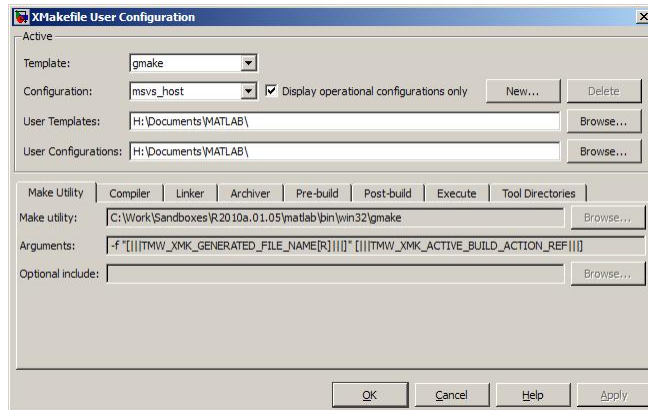
This example shows you how to add support for a software development toolchain to the XMakefile utility. This example uses the Intel Compiler and an IDE.

Note: To specify directory locations, use mapped network drives instead of UNC paths. UNC paths cause build errors with compilers that do not support them.

Create a Configuration

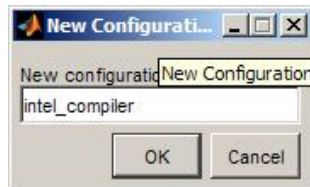
When you click **New**, the new configuration inherits values and behavior from the current configuration. To create a configuration for the Intel Compiler, clone a configuration from one of these configurations: `montavista_arm` and `gcc_target`.

Open the XMakefile User Configuration UI by typing `xmakefilesetup` at the MATLAB prompt. This action displays the following dialog.

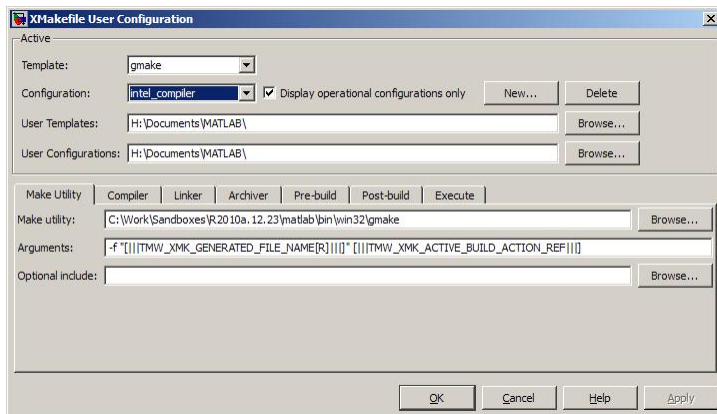


Select an existing configuration, such as `montavista_arm` or `gcc_target`. Click the **New** button.

A pop-up dialog prompts you for the name of the new configuration. Enter `intel_compiler` and click **OK**.



The dialog displays a new configuration called `intel_compiler`, based on the previous configuration.

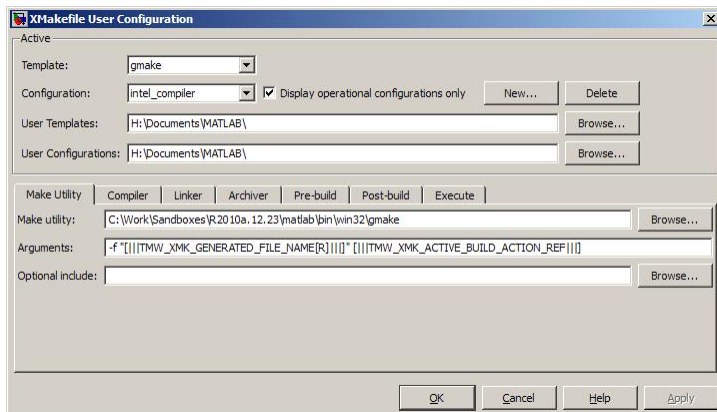


Modify the Configuration

Adjust the compiler, linker, and archiver settings of the newly created configuration. This example assumes the location of the Intel compiler is `C:\Program Files\Intel\Compiler\`.

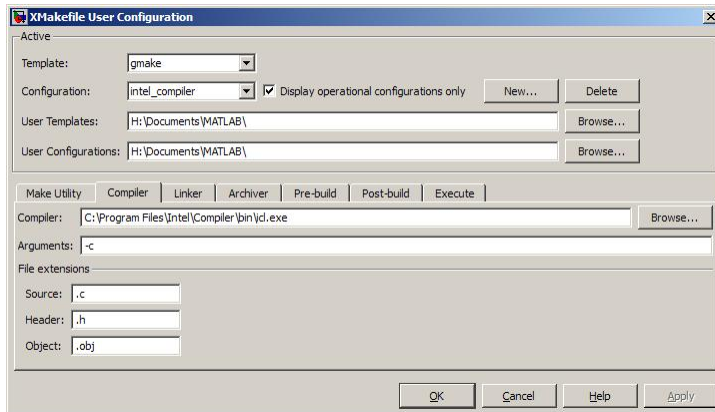
Make Utility

You do not need to make changes. This configuration uses the `gmake` tool that ships with MATLAB.



Compiler

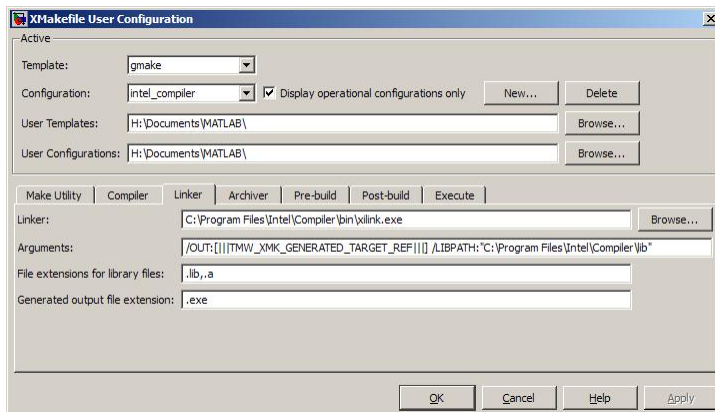
For **Compiler**, enter the location of `icl.exe` in the Intel installation.



Linker

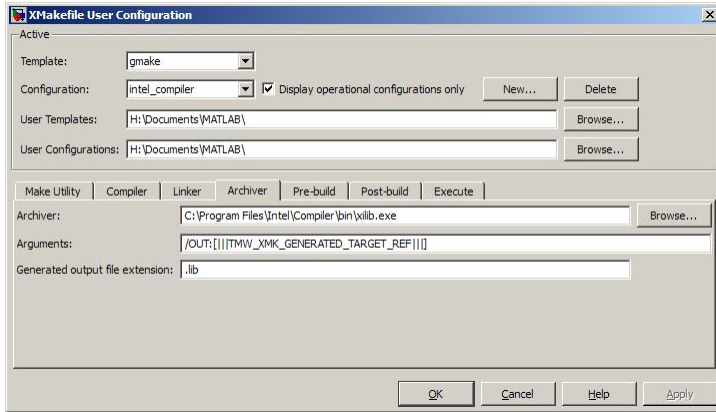
For **Linker**, enter the location of the linker executable, xilink.exe.

For **Arguments**, add the /LIBPATH path to the Intel libraries.



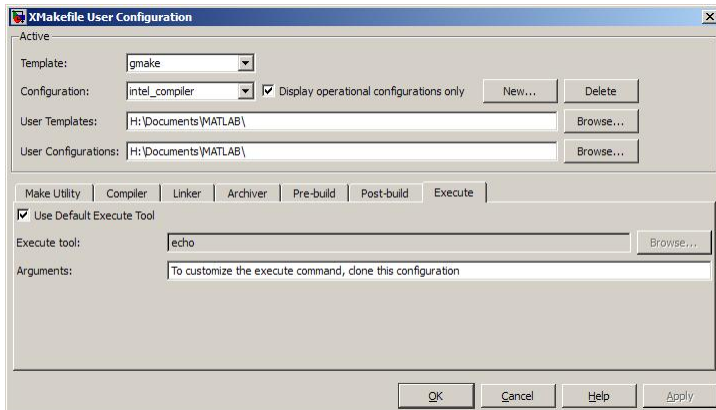
Archiver

For **Archiver**, enter the location of the archiver, xilib.exe. Confirm that **File extensions for library files** includes .lib.



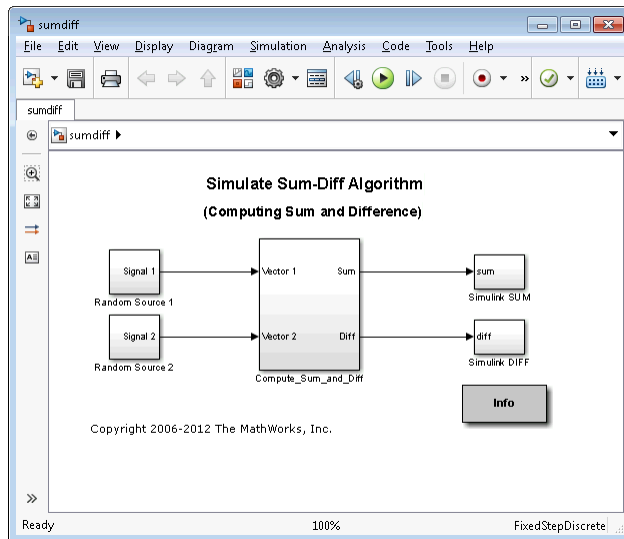
Other tabs

For this example, ignore the remaining tabs. In other circumstances, you can use them to configure additional build actions. In a later step of this example, you will configure the software to automatically build and run the generated code.



Test the Configuration

Open the “sumdiff” model by entering `sumdiff` on the MATLAB prompt.



Configure the `sumdiff` model for use with an IDE. Follow the steps in “Configure Target Hardware Resources” on page 36-3, set the **IDE/Tool Chain** parameter, set **Board** to **Custom**, and **Processor** to **Intel x86/Pentium**.

On the Tool Chain Automation page, set **Operating System** to **None** or select **Windows**. Click **OK**.

Open the Configuration Parameters for the `sumdiff` model by pressing **Ctrl+E**. Set **Build format** to **Makefile** and **Build action** to **Build_and_execute**.

Save the model to a temporary location, such as `C:\Temp\IntelTest\`.

Set that location as a Current Folder by typing `cd C:\temp\IntelTest\` at the MATLAB prompt.

Build the model by pressing **Ctrl+B**. The MATLAB Command Window displays something like:

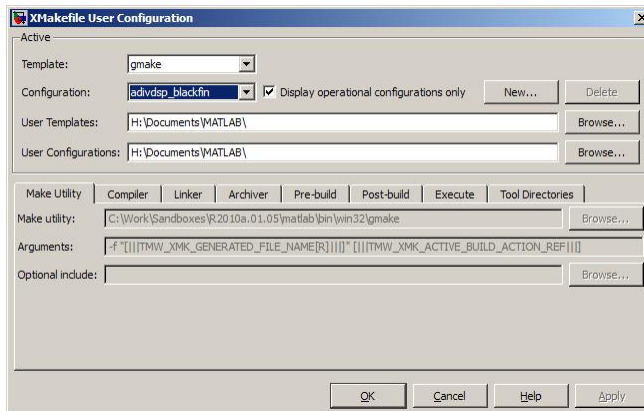
```
### TLC code generation complete.
### Creating HTML report file sumdiff_codegen_rpt.html
### Creating project: c:\temp\IntelTest\sumdiff_idenameide\sumdiff.mk
### Project creation done.
### Building project...
### Build done.
### Downloading program: c:\temp\IntelTest\sumdiff_idenameide\sumdiff
### Download done.
```


A command window comes up showing the running model. Terminate the generated executable by pressing **Ctrl+C**.

XMakefile User Configuration dialog

- “Active” on page 36-27
- “Make Utility” on page 36-29
- “Compiler” on page 36-29
- “Linker” on page 36-30
- “Archiver” on page 36-31
- “Pre-build” on page 36-31
- “Post-build” on page 36-32
- “Execute” on page 36-32
- “Tool Directories” on page 36-33

Active



Template

Set the **Template** parameter to the option that matches the **Configuration** parameter.

Note: In most cases, the only option for **Template** is **gmake**. However, if you have installed a Support Package, **Template** can have multiple options.

The template defines the syntax rules for writing the contents of the makefile or buildfile. The default template is `gmake`, which works with the GNU make utility.

To add templates to this parameter, save them as `.mkt` files to the location specified by the **User Templates** parameter. For more information, see “User Templates” on page 36-29.

Configuration

Select the configuration that best describes your toolchain and target hardware.

You cannot edit or delete the configurations provided by MathWorks. You can, however, edit and delete the configurations that you create.

Use the **New** button to create an editable copy of the currently selected configuration.

Use the **Delete** button to delete a configuration you created.

Note: You cannot edit or delete the configurations provided by MathWorks.

Note: Use mapped network drives instead of UNC paths to specify directory locations. Using UNC paths with compilers that do not support them causes build errors.

Display operational configurations only

When you open the XMakefile User Configuration dialog, the software verifies that each configuration provided by MathWorks contains valid paths to the executable files it uses. If the paths are valid, the configuration is operational. If the paths are not valid, the configuration is not operational.

This setting only applies to configurations provided by MathWorks, not configurations you create.

To display valid configurations, select **Display operational configurations only**.

To display the configurations, including non-operational configurations, clear **Display operational configurations only**.

For more information, see “Making an XMakefile Configuration Operational” on page 36-21.

User Templates

Set the path of the folder to which you can add template files. Saving templates files with the .mkt extension to this folder adds them to the **Templates** options.

User Configurations

Set the location of configuration files you create with the **New** button.

Make Utility

Make utility

Set the path and filename of the make utility executable.

Arguments

Define the command-line arguments to pass to the make utility. For more information, consult the third-party documentation for your make utility.

Optional include

Set the path and file name of an optional makefile to include.

Compiler

Compiler

Set the path and file name of the compiler executable.

Arguments

Define the command-line arguments to pass to the compiler. For more information, consult the third-party documentation for your compiler.

Source

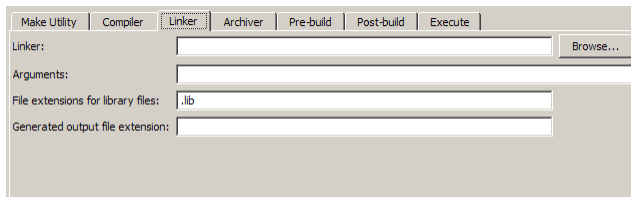
Define the file name extension for the source files. Use commas to separate multiple file extensions.

Header

Define the file name extension for the header files. Use commas to separate multiple file extensions.

Object

Define the file name extension for the object files.

Linker**Linker**

Set the path and file name of the linker executable.

Arguments

Define the command-line arguments to pass to the linker. For more information, consult the third-party documentation for your linker.

File extensions for library files

Define the file name extension for the file library files. Use commas to separate multiple file extensions.

Generated output file extension

Define the file name extension for the generated libraries or executables.

Archiver

Make Utility | Compiler | Linker | **Archiver** | Pre-build | Post-build | Execute

Archiver: Browse...

Arguments:

Generated output file extension:

Archiver

Set the path and file name of the archiver executable.

Arguments

Define the command-line arguments to pass to the archiver. For more information, consult the third-party documentation for your archiver.

Generated output file extension

Define the file name extension for the generated libraries.

Pre-build

Make Utility | Compiler | Linker | Archiver | **Pre-build** | Post-build | Execute

Enable Pre-build Step

Pre-build tool: Browse...

Arguments:

Enable Prebuild Step

Select this check box to define a prebuild tool that runs before the compiler.

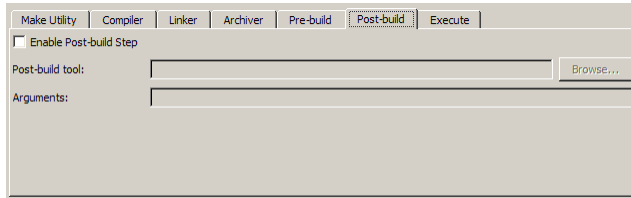
Prebuild tool

Set the path and file name of the prebuild tool executable.

Arguments

Define the command-line arguments to pass to the prebuild tool. For more information, consult the third-party documentation for your prebuild tool.

Post-build



Enable Postbuild Step

Select this check box to define a postbuild tool that runs after the compiler or linker.

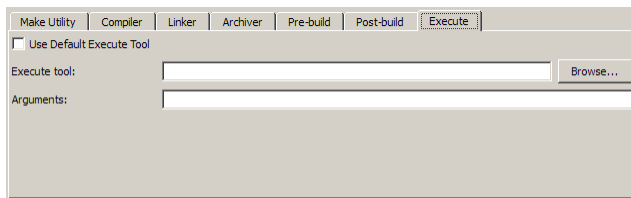
Postbuild tool

Set the path and file name of the postbuild tool executable.

Arguments

Define the command-line arguments to pass to the postbuild tool. For more information, consult the third-party documentation for your postbuild tool.

Execute



Use Default Execute Tool

Select this check box to use the generated derivative as the execute tool when the build process is complete. Uncheck it to specify a different tool. The default value, `echo`, simply displays a message that the build process is complete.

Note: On the Linux operating system, multirate multitasking executables require root privileges to schedule POSIX threads with real-time priority. If you are using makefiles to build multirate multitasking executables on your Linux development system, you

cannot use **Execute tool** to run the executable. Instead, use the Linux command, `sudo`, to run the executable.

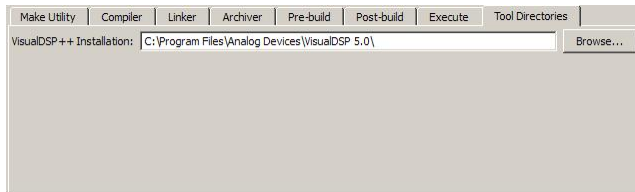
Execute tool

Set the path and file name of the execute tool executable or built-in command.

Arguments

Define the command-line arguments to pass to the execute tool. For more information, consult the third-party documentation for your execute tool.

Tool Directories



Installation

Use the Tool Directories tab to change the toolchain path of an operational configuration.

For example, if you installed two versions of a vendor build tool in separate folders, you can use the **Installation** path to change which one the configuration uses.

Verification and Profiling Generated Code

- “PIL Simulation for IDE and Toolchain Targets” on page 37-2
- “Code Execution Profiling for IDE and Toolchain Targets” on page 37-19
- “Perform Execution Time Profiling for IDE and Toolchain Targets” on page 37-22
- “Perform Stack Profiling with IDE and Toolchain Targets” on page 37-27

PIL Simulation for IDE and Toolchain Targets

In this section...

“Overview” on page 37-2

“PIL Approaches” on page 37-3

“Communications” on page 37-7

“Running Your PIL Application to Perform Simulation and Verification” on page 37-13

“Performing a Model Block PIL Simulation via SCI Using Makefiles” on page 37-13

“Definitions” on page 37-16

“PIL Issues and Limitations” on page 37-17

Overview

Verification consists broadly of running generated code on a processor and verifying that the code does what you intend. Embedded Coder provides processor-in-the-loop (PIL) simulation to meet this need. PIL compares the numeric output of your model under simulation with the numeric output of your model running as an executable on a target hardware.

With PIL, you run your generated code on a target hardware or instruction set simulator. To verify your generated code, you compare the output of model simulation modes, such as Normal or Accelerator, with the output of the generated code running on the processor. You can switch between simulation and PIL modes. This flexibility allows you to verify the generated code by executing the model as compiled code in the target environment. You can model and test your embedded software component in Simulink and then reuse your regression test suites across simulation and compiled object code. This process avoids the time-consuming process of leaving the Simulink software environment to run tests again on object code compiled for the production hardware.

Embedded Coder supports the following PIL approaches:

- Model block PIL
- Top-model PIL
- PIL block

When you use makefiles with PIL, use the “model block PIL” approach. With makefiles, the other two approaches, “top-model PIL” and “PIL block”, are not supported.

PIL Approaches

- “Model Block PIL” on page 37-3
- “Top-Model PIL” on page 37-4
- “PIL Block” on page 37-5

Model Block PIL

Use model block PIL to:

- Verify code generated for referenced models (model reference code interface).
- Provide a test harness model (or a system model) to generate test vector or stimulus inputs.
- Switch a model block between normal, SIL, or PIL simulation modes.

To perform a model block PIL simulation, start with a top-model that contains a model block. The top-model serves as a test harness, providing inputs and outputs for the model block. The model block references the model you plan to run on target hardware. During PIL simulation, the referenced model runs on the target hardware.

For more information about using the model block, see “Model Variants” and “Model Reference”.

By default, your MathWorks software uses the IDE debugger for PIL communications with the target hardware. To achieve faster communications, consider using one of the alternatives presented in “Communications” on page 37-7.

To use model block PIL:

- 1 Create and share a configuration reference between the top model and the referenced model, as described in “Share a Configuration for Multiple Models”.
- 2 Right-click the Model block, and select **ModelReference Parameters**.
- 3 When the software displays the **Function Block Parameters: Model** dialog box, set **Simulation mode** to **Processor-in-the-loop (PIL)** and click **OK**.
- 4 Open the model block.
- 5 In the referenced model (model block) Configuration Parameters (**Ctrl+E**), under **Code Generation > Coder Target**, set **Build action** set to **Archive_library**. This action avoids a warning when you start the simulation.
- 6 Save the changes to both models.

- 7 In the top-model menu bar, select **Simulation > Run**. This action builds the referenced model in the model block, downloads it to your target hardware, and runs the PIL simulation.

Note: In the top-model Configuration Parameters (**Ctrl+E**), under **Code Generation > Coder Target**, leave **Build action** set to **Build_and_execute**. Do not change **Build action** to **Create_Processor_In_the_Loop_Project**.

Top-Model PIL

Use top-model PIL to:

- Verify code generated for a top-model (standalone code interface).
- Load test vectors or stimulus inputs from the MATLAB workspace.
- Switch the entire model between normal and SIL or PIL simulation modes.

Setting Model Configuration Parameters to Generate the PIL Application

Configure your model to generate the PIL executable from your model:

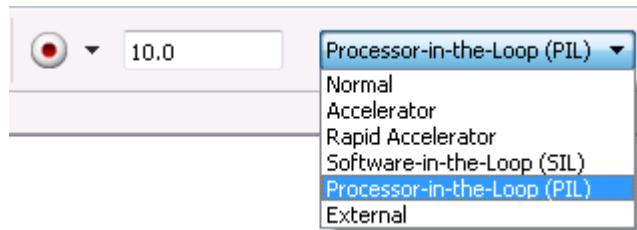
- 1 Configure your model to run on target hardware, as described in “Configure Target Hardware Resources” on page 36-3.
- 2 From the model toolstrip, select **Simulation > Model Configuration Parameters**.
- 3 In Configuration Parameters, select **Code Generation**.
- 4 Set **System Target File** to `idmlink_ert.tlc`.
- 5 From the list of panes under **Code Generation**, choose **Coder Target**.
- 6 Set **Build format** to **Project**.
- 7 Set **Build action** to `Create_processor_in_the_loop_project`.
- 8 Click **OK** to close the Configuration Parameters dialog box.

For more information, see “Code Generation: Coder Target Pane”.

Running the Top-Model PIL Application

To create a PIL block, perform the following steps:

- 1 In the model toolstrip, set the Simulation mode to **Processor-in-the-loop**.



- 2 In the model toolstrip, click Run.



A new Simulink Editor opens with the new PIL model block in it. The third-party IDE compiles and links the PIL executable file. Follow the progress of the build process in the MATLAB Command Window.

PIL Block

Use the PIL block to:

- Verify code generated for a top-model (standalone code interface) or subsystem (right-click build standalone code interface).
- Represent a component running in SIL or PIL mode. The test harness model or a system model provides test vector or stimulus inputs.

Preparing Your Model to Generate a PIL Block

Start with a model that contains the algorithm blocks you want to verify on the processor as compiled object code. To create a PIL application and PIL block from your algorithm subsystem, follow these steps:

- 1 Identify the algorithm blocks to cosimulate.
- 2 Convert those blocks into an unmasked subsystem in your model.

For information about how to convert your process to a subsystem, refer to “Creating Subsystems” in *Using Simulink* or in the online Help system.

- 3 Open the newly created subsystem.
- 4 Configure your subsystem to run on target hardware, as described in “Configure Target Hardware Resources” on page 36-3.

Setting Model Configuration Parameters to Generate the PIL Application

After you create your subsystem, set the Configuration Parameters for your model to enable the model to generate a PIL block.

Configure your model to enable it to generate PIL algorithm code and a PIL block from your subsystem:

- 1 From the model menu bar, select **Simulation > Model Configuration Parameters**. This action opens the Configuration Parameters dialog box.
- 2 In the Configuration Parameters dialog box, select **Code Generation**.
- 3 Set **System Target File** to `idmlink_ert.tlc`.
- 4 From the list of panes under **Code Generation**, choose **Coder Target**.
- 5 Set **Build format** to `Project`.
- 6 Set **Build action** to `Create_processor_in_the_loop_project`.
- 7 Click **OK** to close the Configuration Parameters dialog box.

For more information, see “Code Generation: Coder Target Pane”.

Creating the PIL Block from a Subsystem

To create a PIL block, perform the following steps:

- 1 Right-click the masked subsystem in your model and select **C/C++ Code > Build This Subsystem** from the context menu.

A new Simulink Editor opens and the new PIL block appears in it. The third-party IDE compiles and links the PIL executable file.

This step builds the PIL algorithm object code and a PIL block that corresponds to the subsystem, with the same inputs and outputs. Follow the progress of the build process in the MATLAB Command Window.

- 2 Copy the new PIL block from the new model to your model. To simulate the subsystem processes concurrently, place it parallel to your masked subsystem. Otherwise, replace the subsystem with the PIL block.

To see a PIL block in a parallel masked subsystem, search the product help for *Getting Started with Application Development* and select the example that matches your IDE.

Note: Models can have multiple PIL blocks for different subsystems. They cannot have more than one PIL block for the same subsystem. Including multiple PIL blocks for the same subsystem causes errors and inaccurate results.

Communications

- “TCP/IP” on page 37-8
- “Additional Steps for TI C6000 Processors” on page 37-9
- “Serial Communication Interface (SCI) for Texas Instruments C2000” on page 37-9
- “IDE Debugger” on page 37-11

Choose one of the following communication methods for transferring code and data during PIL simulations:

Method	Speed	Comments
IDE Debugger	Slow	<ul style="list-style-type: none"> • Supports PIL communications with an executable running on embedded target hardware. • Supports the largest number of targets. • Requires a physical connection between host and target hardware. • Only works with builds from IDE projects. Does not work with builds from makefiles.
TCP/IP	Fast	<ul style="list-style-type: none"> • Supports PIL communications with an executable running on a Linux or Windows host. • Supports embedded targets running Linux, TI DSP/BIOS, and Wind River VxWorks. • Requires network connection between host and target hardware. • Works with builds from IDE projects and from makefiles.
Serial Communication Interface (SCI)	Fast	<ul style="list-style-type: none"> • Supports PIL communications with an executable running on embedded target hardware.

Method	Speed	Comments
		<ul style="list-style-type: none"> • Supports TI C2802x, C2803x, 2806x, c280x, C281x, C2834x, C28x3x microcontrollers. • Requires an SCI connection between host and target hardware. • Works with builds from IDE projects and from makefiles.

TCP/IP

You can use TCP/IP for PIL communications with target hardware running:

- Linux
- Texas Instruments DSP/BIOS
- Wind River VxWorks

Using TCP/IP for PIL communications is typically faster than using a debugger, particularly for large data sets, such as with video and audio applications.

It also works well when you build an application on a remote Linux target using the `remoteBuild` function.

You can use TCP/IP with the following PIL approaches:

- Top-model PIL
- Model block PIL

TCP/IP does not work with the Subsystem PIL approach.

To enable and configure TCP/IP with PIL:

- 1 Set up a PIL simulation according to the PIL approach you have chosen.
- 2 In the MATLAB Command Window, use `setpref` to specify the IP address of the PIL server (`servername`).

If you are running the PIL server on a remote target, specify the IP address of the target hardware. For example:

```
setpref('MathWorks_Embedded_IDE_Link_PIL_Preferences', 'servername', '144.212.109.114');
```

If you are running PIL server locally, on your host Windows or Linux system, enter `'localhost'` instead of an IP address:


```
setpref('MathWorks_Embedded_IDE_Link_PIL_Preferences','servername','localhost');
```

- 3 Specify the TCP/IP port number to use for PIL data communication. Use one of the free ports in your system. For example:

```
setpref('MathWorks_Embedded_IDE_Link_PIL_Preferences','portnum', 17025);
```

- 4 Enable PIL communications over TCP/IP:

```
setpref('MathWorks_Embedded_IDE_Link_PIL_Preferences','enabletcpip', true);
```

To disable PIL communications over TCP/IP, change the value to `false`. This action automatically enables PIL communications over an IDE debugger, if an IDE is available.

- 5 Open the Configuration Parameters in your model. On the Coder Target pane, set the **Operating System** parameter to the operating system your target hardware is running.

Note: You cannot use TCP/IP for PIL when the value of **Operating System** is **None**.

- 6 Regenerate the code or PIL block.

To disable PIL communications over TCP/IP, enter:

```
setpref('MathWorks_Embedded_IDE_Link_PIL_Preferences','enabletcpip', false);
```

Additional Steps for TI C6000 Processors

Add an IP Config block to the following location in your model:

- For top-model PIL, add it at the top level of your model.
- For model block PIL, add it to the referenced model to which you are pointing.
- For Subsystem PIL, place it in the subsystem.

Configure the IP Config block settings as described in C6000 IP Config.

To determine the IP address assigned to the PIL server on the C6000 target:

- 1 Enter an arbitrary IP address the first time you specify the IP address.
- 2 Build and run the code for your model.
- 3 In the CCS command window, observe the actual IP address assigned to the C6000 processor by the DHCP server.
- 4 Enter the actual IP address the second time you specify the IP address.

Serial Communication Interface (SCI) for Texas Instruments C2000

You can use SCI for processor-in-the-loop (PIL) simulations with Texas Instruments C2000 processors that support Serial Communications Interface (SCI). For other targets, configure PIL to communicate through TCP/IP or an IDE debugger.

SCI typically provides faster communications than an IDE debugger, particularly for large data sets.

To enable and configure SCI with PIL:

- 1 Set up a PIL simulation according to the PIL approach you have chosen. For more information, see “PIL Approaches” on page 37-3.
- 2 In the MATLAB Command Window, use `setpref` to specify the Configuration Parameters:
 - a Select the SCI port on your host computer for communicating with the target hardware. For example, to use COM1, enter the following command:

```
setpref('MathWorks_Embedded_IDE_Link_PIL_Preferences', 'COMPort', 'COM1');
```
 - b Set the baud rate of the SCI port. For example, if both the host computer and the target support 11,5200 baud, enter:

```
setpref('MathWorks_Embedded_IDE_Link_PIL_Preferences', 'BaudRate', 115200);
```
 - c Enable PIL communications over SCI:

```
setpref('MathWorks_Embedded_IDE_Link_PIL_Preferences', 'enableserial', true);
```
- 3 Configure the serial communications settings on your host computer to match the preceding values. For example, in Windows 7:
 - a Open the Windows Device Manager. (Press the Windows key on your keyboard and search for “Device Manager”.)
 - b Expand **Ports (COM & LPT1)**.
 - c Right-click the communications port you previously specified in MATLAB, such as **Communications Port (COM1)**, and select **Properties**.
 - d Go to the **Port Settings** tab, and match the value of **Bits per second** with the baud rate you previously specified in MATLAB. This value should match the baud rate you set in MATLAB. For example, `'BaudRate', 115200`.
- 4 Regenerate the code or PIL block.

Note: In serial PIL simulation, the changes that you make to the `BaudRate` or `COMPort` parameters in `MathWorks_Embedded_IDE_Link_PIL_Preferences` are not detected, if the following conditions are met:

- You have already built your PIL application.
- You have set the **Configuration Parameters >Model Referencing >Rebuild** option to a value other than **Always**.
- You have not made changes to the model.

To apply your COM port or baudrate changes, either change the value in the **Configuration Parameters >Model Referencing >Rebuild** option to **Always**, or resave the model to force a new build of the PIL application.

To disable PIL communications over SCI, enter:

```
setpref('MathWorks_Embedded_IDE_Link_PIL_Preferences','enableserial',false);
```

Note: If you change the following parameters while using TCP/IP or serial communication for PIL, the software does not regenerate the PIL and communication code.

- TCP/IP Communication Parameters: **IP Address** and **Port number**
- Serial Communication Parameters: **COM Port** and **BaudRate**

To work around this issue, remove the `slprj` folder, generated code, and the generated MEX file. Then, regenerate the PIL code.

See “Performing a Model Block PIL Simulation via SCI Using Makefiles” on page 37-13

IDE Debugger

To enable PIL communications over an IDE debugger, disable PIL communications over TCP/IP and SCI by entering the following commands:

```
setpref('MathWorks_Embedded_IDE_Link_PIL_Preferences','enabletcpip',false);  
setpref('MathWorks_Embedded_IDE_Link_PIL_Preferences','enableserial',false);
```

Then regenerate the code or PIL block.

Using IDE debugger for PIL communication only works when you build your code from IDE projects. Using IDE debugger for PIL communication does not work with builds from makefiles.

Configuring Breakpoints

You can use the `setStartApplicationPause` API to set breakpoints in the PIL application on the *first* PIL block simulation. If you do not use the `setStartApplicationPause` API, you can configure breakpoints after the initial run. The breakpoints remain active for subsequent runs.

You can enter the following static API method to pause after loading the application and manually configure breakpoints:

```
rtw.connectivity.Launcher.setStartApplicationPause(pauseAmount)
```

About this method:

- This method tells the MATLAB session to pause immediately after the PIL launcher starts the PIL application.
- `pauseAmount` is a pause time in seconds. To disable the pause, enter 0.

When you do not specify a pause, the software displays the following message:

```
### To pause during PIL application start, run: >> rtw.connectivity.Launcher.  
setStartApplicationPause(120)
```

The default pause is 120 sec. You can change this value.

When you specify a pause, a Start PIL Application Pause message box appears and displays following message:

```
Pausing during PIL application start for 120s (click OK to continue).  
To disable this pause, see the hyperlink in the MATLAB command window.
```

- The MATLAB Command Window shows the following text:

```
### To remove the pause during PIL application start,  
run: >> rtw.connectivity.Launcher. setStartApplicationPause(0)  
where rtw.connectivity.Launcher. setStartApplicationPause(0) is a  
hyperlink.
```

- The pause times out, or you can clear it early by closing the message box.
- During the pause, you cannot access MATLAB and thus cannot configure breakpoints programmatically via the IDE Automation Interface API.
- For the PIL block, the debugger stays open between simulation runs. When you perform an initial simulation run, you can automatically configure breakpoints via the IDE Automation Interface API before starting the second simulation.

Running Your PIL Application to Perform Simulation and Verification

After you add PIL block to your model, add the required pause in seconds, using the following command in the MATLAB command prompt:

```
rtw.connectivity.Launcher.setStartApplicationPause(120)
```

Then click **Simulation > Run** or press **Ctrl+T** to run the PIL simulation and view the results.

Note The pause command is to make sure that the automatic download of PIL completes, before the model starts executing.

Performing a Model Block PIL Simulation via SCI Using Makefiles

This example shows you the complete workflow for performing a processor-in-the loop (PIL) simulation that uses Serial Communications Interface (SCI) for communications.

Prerequisites

Follow the board vendor's instructions for setting up a Texas Instruments C2000-based board that supports SCI. Connect the board to your host computer using a serial cable.

Configure Your Model for Target Hardware

- 1 Enter `fuelsys_pil` in MATLAB. This action opens the `fuelsys_pil` model with the title, "Verifying the Fixed-Point Fuel Control System".
- 2 Configure `fuelsys_pil`. Follow the steps in "Configure Target Hardware Resources" on page 36-3 setting:
 - **IDE/Tool Chain** to the version of CCS you are using.
 - **Board** to a board that supports using SCI, such as SD F28335 eZdsp.
- 3 Click **Yes**.

If you are working with CCSv3, configure `fuelsys_pil` to use makefiles:

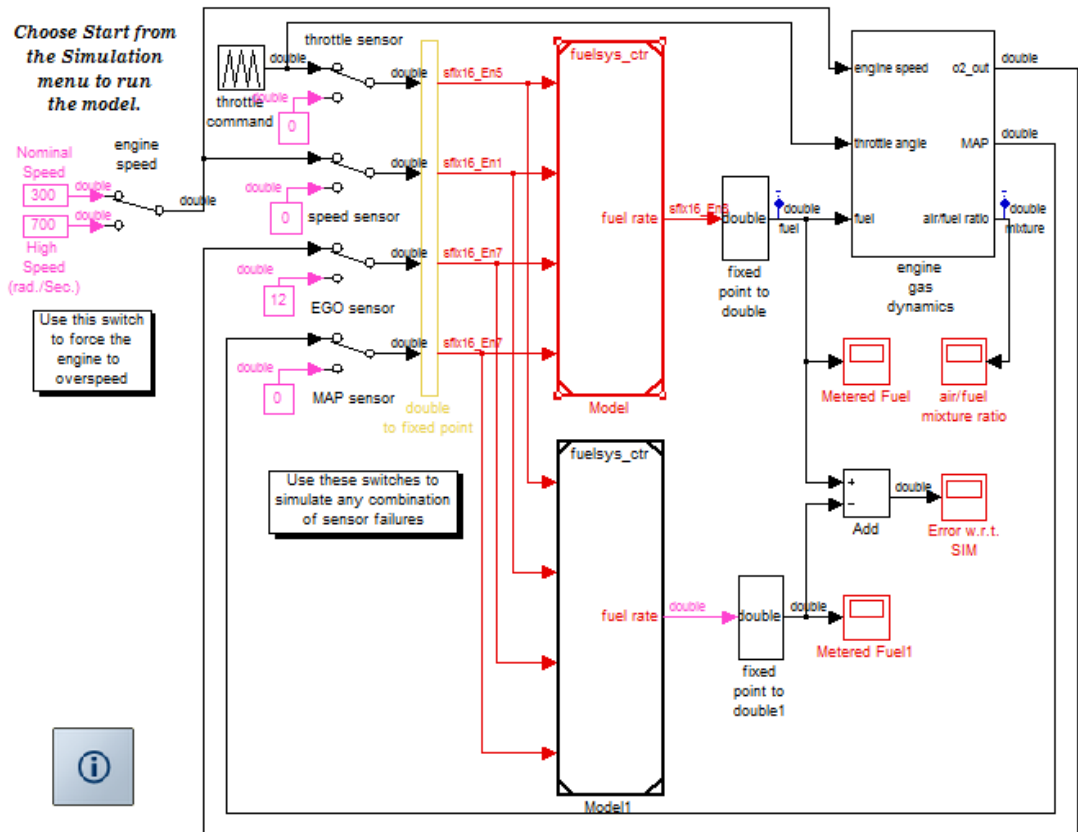
- 1 Select **Simulation > Model Configuration Parameters**.
- 2 In the Configuration Parameters dialog box, expand **Code Generation** and select **Coder Target**.
- 3 On the Coder Target pane, set **Build format** to **Makefile**.

If you are working with CCSv4/5, you do not need to configure the model to use makefiles. Initializing the configuration parameters for CCSv4/5 automatically sets **Build format** to Makefile.

Configure Your Model for the Model Block PIL Approach

- 1 In the `fuelsys_pil`, copy the `fuelsys_ctr` model and paste it into the vacant space below. Connect it to the input/output signals provided.

Verifying the Fixed-Point Fuel Control System



Copyright 2005-2011 The MathWorks, Inc.

- 2 Right-click the upper `fuelsys_ctr` model, labeled “Model”, and select **ModelReference Parameters**.
- 3 In the **Function Block Parameters: Model** dialog box, set the **Simulation mode** parameter to `Processor-in-the-loop (PIL)`. Click the **OK** button.
- 4 Open the upper `fuelsys_ctr` model, labeled “Model”.
- 5 From the menu in the open `fuelsys_ctr` model, select **Simulation > Model Configuration Parameters** (or press **Ctrl+E**).
- 6 In the Configuration Parameters dialog box, in the **Solver** pane, set the **Type** parameter to `Fixed-step`, and set **Solver** to `ode3 (Bogacki-Shampine)`.
- 7 In the MATLAB Command Window, enter:


```
set_param('fuelsys_ctr', 'ModelReferenceSymbolNameMessage', 'none')
```
- 8 In the **Code Generation > Interface** pane, clear the Software environment **absolute time** check box.
- 9 In the **Code Generation > Coder Target** pane, set the Run time **Build action** parameter to `Archive library`.
- 10 Save the changes to your model, and leave the model open.
- 11 Open the top model, `fuelsys_pil`. Open the Configuration Parameters dialog box, in the **Solver** pane, verify that the **Type** parameter is set to `Fixed-step`, and reset **Solver** to `ode3 (Bogacki-Shampine)`.

Note: For information other PIL approaches, see “PIL Approaches” on page 37-3.

Enable and configure SCI

- 1 Use `setpref` to specify the Configuration Parameters in MATLAB :


```
setpref('MathWorks_Embedded_IDE_Link_PIL_Preferences', 'COMPort', 'COM1');
setpref('MathWorks_Embedded_IDE_Link_PIL_Preferences', 'BaudRate', 115200);
setpref('MathWorks_Embedded_IDE_Link_PIL_Preferences', 'enableserial', true);
```
- 2 Configure the serial communications settings on your host computer to match the preceding values. For example, in Windows 7:
 - a Open the Windows Device Manager. (Press the Windows key on your keyboard and search for “Device Manager”.)
 - b Expand **Ports (COM & LPT1)**.

- c Right-click the communications port you previously specified in MATLAB, such as **Communications Port (COM1)**, and select **Properties**.
- d Go to the **Port Settings** tab, and match the value of **Bits per second** with the baud rate you previously specified in MATLAB. This value should match the baud rate you set in MATLAB. For example, 'BaudRate' , 115200.

Configure the Software to Use Makefiles

Set up the xmakefile for CCSv4/5 as described in the section “Using Makefiles with Code Composer Studio 4/5”.

Run the PIL Simulation

- 1 Make sure the SD F28335 eZdsp board is connected to your host computer via serial and USB cables and powered up.
- 2 Add the required pause in seconds, using the following command in the MATLAB command prompt:

```
run: >> rtw.connectivity.Launcher.setStartApplicationPause(120)
```

- 3 Simulate the fuelsys_pil model (press **Ctrl+T**).

Definitions

PIL Algorithm

The algorithmic code, which corresponds to a subsystem or portion of a model, to test during the PIL simulation. The PIL algorithm is in compiled object form to enable verification at the object level.

PIL Application

The executable application that runs on the processor platform. Your coder product creates a PIL application by augmenting your algorithmic code with the PIL execution framework. The PIL execution framework code compiles as part of your embedded application.

The PIL execution framework code includes the string.h header file so that the PIL application can use the `memcpy` function. The PIL application uses `memcpy` to exchange data between the Simulink model and the simulation processor.

PIL Block

When you build a subsystem from a model for PIL, the process creates a PIL block optimized for PIL simulation. When you run the simulation, the PIL block acts as the interface between the model and the PIL application running on the processor. The PIL block inherits the signal names and shape from the source subsystem in your model. Inheritance is convenient for copying the PIL block into the model to replace the original subsystem for simulation.

PIL Issues and Limitations

Consider the following issues when you work with PIL blocks.

Constraints

When using PIL in your models, keep in mind the following constraints:

- Models can have multiple PIL blocks for different subsystems. They cannot have more than one PIL block for the same subsystem. Including multiple PIL blocks for the same subsystem causes errors and inaccurate results.
- A model can contain a single model block running PIL mode.
- A model can contain a subsystem PIL block or a model block in PIL mode, but not both.

Generic PIL Issues

Refer to “PIL Feature Support and Limitations” for general information about using the PIL block with embedded link products.

With Texas Instruments CCS, PIL with DSP/BIOS Enabled Does Not Support System Stack Profiling

Enabling DSP/BIOS for Texas Instruments processors disables the stack profiling option. To use stack profiling with PIL, open the **Target Hardware Resources** pane in model Configuration Parameters, and set the **Operating System** parameter to **None**. For help opening the Target Hardware Resources pane, see “Configure Target Hardware Resources” on page 36-3.

Simulink Coder grt.tlc-Based Targets Not Supported

PIL does not support `grt.tlc` system target files.

To use PIL, set **System target file** in the Configuration Parameters > Code Generation pane to `idelink_ert.tlc`.

Code Execution Profiling for IDE and Toolchain Targets

In this section...

“Execution Time Profiling” on page 37-19

“Stack Profiling” on page 37-19

Execution Time Profiling

You can measure the execution times during a standalone execution or processor-in-the-loop (PIL) simulation. You can generate execution time profiles for synchronous tasks, asynchronous tasks, and atomic subsystems. Use this feature to check whether your code runs in real time on your target hardware. For details, see “Perform Execution Time Profiling for IDE and Toolchain Targets” on page 37-22.

You can use this profiling for generated code in the following cases:

- **Code Generation > System target file** is `ert.tlc` and **Code Generation > Target hardware** is not None, for example, ARM Cortex-A9 (QEMU) or ARM Cortex-M3 (QEMU).
- **Code Generation > System target file** is `idelink_ert.tlc`

The following table provides execution time profiling support information.

Mode	Coder Target > Tool Chain Automation > Build format parameter value	Products
Standalone execution or PIL simulation	Project	Supports the following IDEs: <ul style="list-style-type: none"> • Analog Devices VisualDSP++ • Texas Instruments Code Composer Studio 3.3
PIL simulation	Makefile	See “Supported Tool Chains in Embedded Coder”

Stack Profiling

With stack profiling, you can determine how generated code uses the processor system stack. Using the `profile` method, you can initialize and test the size and usage of the

stack. See “Perform Stack Profiling with IDE and Toolchain Targets” on page 37-27. This information can help you optimize both the size of the stack and how your code uses the stack.

You can use this profiling for generated code in the following cases:

- **Code Generation > System target file** is `ert.tlc` and **Code Generation > Target hardware** is not None, for example, ARM Cortex-A9 (QEMU) or ARM Cortex-M3 (QEMU).
- **Code Generation > System target file** is `idelink_ert.tlc`

Note: Stack profiling is not supported on embedded targets that run an operating system or RTOS.

To provide stack profiling, `profile` writes a known pattern to the addresses in the stack. After you run your application for a while, and then stop your application, `profile` examines the contents of the stack addresses. `profile` counts each address that does not contain the known pattern. The total number of addresses that have been used, compared to the total number of addresses that you allocated, becomes the stack usage profile. This profile process does not determine how often your application changes an address.

You can profile the stack with the manually written code in a project and the code that you generate from a model.

When you use `profile` to initialize and test the stack operation, the software returns a report that contains information about stack size, usage, addresses, and direction. With this information, you can modify your code to use the stack efficiently. The following program listing shows the stack usage results from running an application on a simulator.

```
profile(IDE_Obj, 'stack', 'report')
```

```
Maximum stack usage:
```

```
System Stack: 532/1024 (51.95%) MAUs used.
```

```
        name: System Stack  
startAddress: [512    0]
```

```

    endAddress: [1535    0]
    stackSize: 1024 MAUs
    growthDirection: ascending

```

The following table describes the entries in the report.

Report Entry	Units	Description
System Stack	Minimum Addressable Unit (MAU)	Maximum number of MAUs used and the total MAUs allocated for the stack.
name	String for the stack name	Lists the name assigned to the stack.
startAddress	Decimal address and page	Lists the address of the stack start and the memory page.
endAddress	Decimal address and page	Lists the address of the end of the stack and the memory page.
stackSize	Addresses	Reports number of address locations, in MAUs, allocated for the stack.
growthDirection	Not applicable	Reports whether the stack grows from the lower address to the higher address (ascending) or from higher to lower (descending).

Perform Execution Time Profiling for IDE and Toolchain Targets

In this section...

“Execution Profiling During Standalone Execution” on page 37-22

“Execution Profiling During PIL Simulation” on page 37-25

Execution Profiling During Standalone Execution

During standalone execution, instrumentation in the generated code collects execution time samples, which are stored in target hardware memory. After halting target hardware execution, you can use the `profile` function to transfer the execution data from target hardware memory to the MATLAB workspace for viewing and analysis.

You can perform profiling by task or subsystem. A profiling sample represents a task or subsystem execution instance. Each sample requires two memory locations, one for the start time and one for the end time. Therefore, you must specify a buffer size that is twice the number of required profiling samples. Sample collection begins with the start of code execution and ends when the buffer is full.

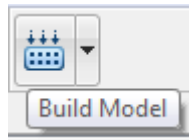
Task Profiling

To configure a model for task execution profiling:

- 1 In your model, select **Simulation > Model Configuration Parameters**.
- 2 Select the **Code Generation > Coder Target** pane.
- 3 Set **Build format** to `Project` and set **Build action** to `Build_and_execute`.
- 4 Select **Profile real-time execution**.
- 5 In the **Profile by** list, select **Tasks**.
- 6 Specify **Number of profiling samples to collect**, the size of the buffer that stores execution data. Enter a value that is twice the number of profiling samples you require.
- 7 Click **OK**.

To view the execution profile for your model:

1



Click **Build Model** on the model toolstrip. This action builds, loads, and runs your code on the processor.

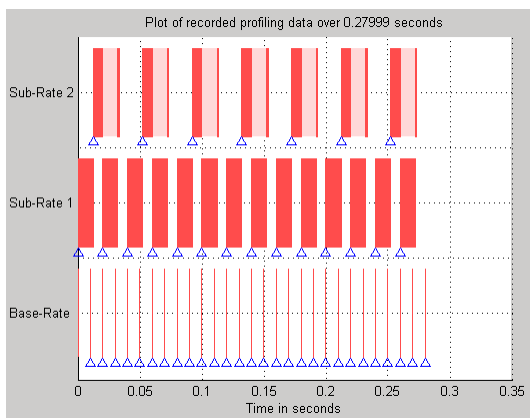
- 2 To stop the running program, select **Debug > Halt** in the IDE or use `IDE_obj.halt` from the MATLAB command line. Gathering profiling data from a running program can yield inaccurate results.
- 3 At the MATLAB command prompt, enter

```
profile(IDE_Obj, 'execution', 'report')
```

to view the MATLAB software graphic of the execution report and the HTML execution report.

For more information about other reporting options, see the product help for the `profile` function.

The following profiling plot is from an application that runs with three rates — the base rate and two slower rates. Gaps in the **Sub-Rate 2** task bars indicate preempted operations.



Subsystem Profiling

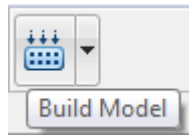
To configure a model for subsystem execution profiling:

- 1 Configure your model for your IDE, tool chain, and target hardware, as described in “Configure Target Hardware Resources” on page 36-3.

- 2 On the **Coder Target** pane, set **Build format** to **Project** and set **Build action** to **Build_and_execute**.
- 3 Select **Profile real-time execution**.
- 4 In the **Profile by** list, select **Atomic subsystems**.
- 5 Specify **Number of profiling samples to collect**, the size of the buffer that stores execution data. Enter a value that is twice the number of profiling samples you require.
- 6 Click **OK**.

To view the execution profile for your model:

1



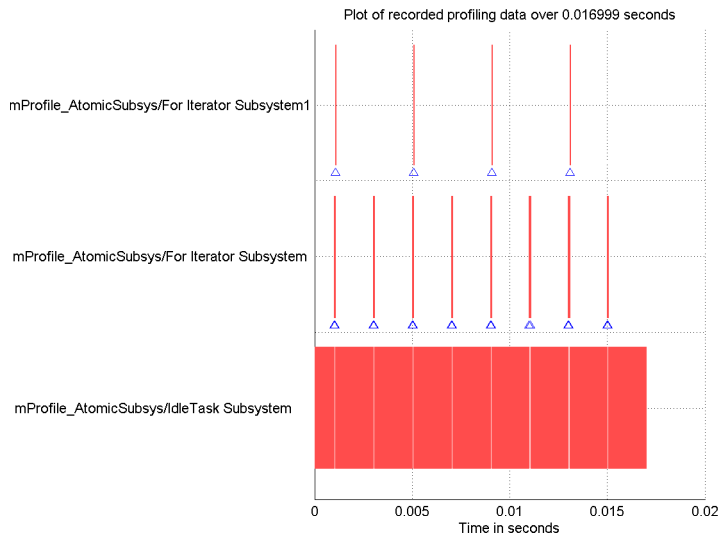
Click **Build Model** on the model toolstrip. This action builds, loads, and runs your code on the processor.

- 2 To stop the running program, select **Debug > Halt** in the IDE or use `IDE_obj.halt` from the MATLAB command line. Gathering profiling data from a running program can yield inaccurate results.
- 3 At the MATLAB command prompt, enter:

```
profile(IDE_Obj, 'execution', 'report')
```

to view the MATLAB software graphic of the execution report and the HTML execution report.

The following profiling plot is from an application with three subsystems — **For Iterator Subsystem**, **For Iterator Subsystem1**, and **Idle Task Subsystem**.



Execution Profiling During PIL Simulation

During a processor-in-the-loop (PIL) simulation, you can profile execution times of synchronous tasks. The software stores the profile data in a `coder.profile.ExecutionTime` object, located in the MATLAB workspace. After halting the PIL simulation, you can view and analyze the data.

Gathering Execution Profile Data

- 1 Configure a model for PIL simulation, as described in “PIL Simulation for IDE and Toolchain Targets” on page 37-2.
- 2 In your model, select **Simulation > Model Configuration Parameters**.
- 3 In the Configuration Parameters dialog box, select **Code Generation**, and then **Verification**.
- 4 Select the **Measure task execution time** check box.
- 5 Provide a valid MATLAB variable name in the **Workspace** edit box. The software uses this name when it creates the `coder.profile.ExecutionTime` object.
- 6 Click **OK** to close the Configuration Parameters dialog box.

- 7 Run the PIL simulation, as described in “PIL Simulation for IDE and Toolchain Targets” on page 37-2.

The software creates the `coder.profile.ExecutionTime` object and stores the execution profile data in it.

- 8 Halt the PIL simulation.

Analyzing the Execution Profile Data

After halting the PIL simulation, you can view or analyze the data in the `coder.profile.ExecutionTime` object. For more information, see:

- “View and Compare Code Execution Times”
- “Analyze Code Execution Data”

Depending on the target, the execution profile data is measured in seconds or timer ticks.

Targets	Units
Texas Instruments C2000, C5000, and C6000 processors with Code Composer Studio IDE	Seconds
Texas Instruments C6000 processors running DSP/BIOS with Code Composer Studio IDE	Timer Ticks
Analog Devices Blackfin, SHARC, and TigerSHARC processors with VisualDSP++ IDE	Timer Ticks
ARM processors running Wind River VxWorks	Timer Ticks

The `coder.profile.ExecutionTime` class has property `TimerTicksPerSecond` for getting and setting the data units. You can use this property on the execution profile data object after halting the PIL simulation. When the data unit is timer ticks, using the `TimerTicksPerSecond` property converts the data units to seconds.

Perform Stack Profiling with IDE and Toolchain Targets

To profile the system stack operation:

- 1 Load an application.
- 2 Set up the stack to enable profiling.
- 3 Run your application.
- 4 Request the stack profile information.

Follow these steps to profile the stack as your application interacts with it. This particular example uses Texas Instruments Code Composer Studio 3.3. However, you can generalize from this example to another supported IDE.

- 1 Load the application to profile.
- 2 Use the `profile` method with the **setup** input keyword to initialize the stack to a known state.

```
profile(IDE_Obj, 'stack', 'setup')
```

With the **setup** input argument, `profile` writes a known pattern into the addresses that compose the stack.

- 3 Run your application.
- 4 Stop your running application. Stack use results gathered from an application that is running may be inaccurate.
- 5 Use the `profile` method to capture and view the results of profiling the stack.

```
profile(IDE_Obj, 'stack', 'report')
```

The following example shows how to set up and profile the stack. The IDE link handle object, `IDE_Obj`, must exist in your MATLAB workspace and your application must be loaded on your processor. This example comes from a TI C6713 simulator.

```
profile(IDE_Obj, 'stack', 'setup') % Set up processor stack
%by write A5 to the stack addresses.
```

Maximum stack usage:

System Stack: 0/1024 (0%) MAUs used.

```
      name: System Stack
startAddress: [512  0]
endAddress: [1535  0]
  stackSize: 1024 MAUs
```

growthDirection: ascending

```
run(IDE_Obj)
halt(IDE_Obj)
profile(IDE_Obj, 'stack', 'report') % Request stack use report.
```

Maximum stack usage:

System Stack: 356/1024 (34.77%) MAUs used.

```
      name: System Stack
startAddress: [512  0]
endAddress: [1535  0]
      stackSize: 1024 MAUs
growthDirection: ascending
```

Processor-Specific Optimizations for Embedded Targets

Replace Code for Embedded Targets

In this section...

“Using a Processor-Specific Code Replacement Library to Optimize Code” on page 38-2

“Process of Determining Optimization Effects Using Real-Time Profiling Capability” on page 38-2

Using a Processor-Specific Code Replacement Library to Optimize Code

You can optimize the code the code generator produces for a specific processor by configuring the code generator to use a code replacement library (CRL) during code generation. If you have an Embedded Coder license, you can develop and apply custom code replacement libraries.

For more information about replacing code, using code replacement libraries that MathWorks provides, see “What Is Code Replacement?” and “Replace Code Generated from Simulink Models”. For information about developing code replacement libraries, see “What Is Code Replacement Customization?” and “Develop a Code Replacement Library”.

Process of Determining Optimization Effects Using Real-Time Profiling Capability

You can use the real-time profiling capability to examine the results of applying the processor-specific library functions and operators to your generated code. After you select a processor-specific code replacement library, use the real-time execution profiling capability to examine the change in program execution time.

Use the following process to evaluate the effects of applying a processor-specific code replacement library when you generate code:

- 1 Enable real-time profiling in your model. Refer to “Code Execution Profiling”.
- 2 Generate code for your project without specifying a code replacement library (the default **Code replacement library** setting is **None**).
- 3 Profile the code, and save the report.
- 4 Rebuild your project using a processor-specific code replacement library.
- 5 Profile the code, and save the second report.

- 6 Compare the profile report from running your application with the processor-specific library selected to the profile results in the first report with no code replacement library selected.

For an example of verifying the code optimization, search help for "Optimizing Embedded Code via Code Replacement Library" and select the example that matches your IDE.

Working with Texas Instruments Code Composer Studio 3.3 IDE

- “Set Up” on page 39-2
- “Code Composer Studio” on page 39-3
- “Getting Started” on page 39-5
- “IDE Automation Interface” on page 39-9
- “IDE Project Generator” on page 39-52
- “Exporting Filter Coefficients from FDATool” on page 39-61
- “Using Makefiles with Code Composer Studio 3.x” on page 39-76
- “Reported Limitations and Tips” on page 39-80
- “Setting Up Code Composer Studio (ert.tlc System Target File)” on page 39-87
- “IDE Link Frequently Asked Question: Why do I get an error when I invoke TICCS?” on page 39-89

Set Up

Before you use Embedded Coder with Code Composer Studio (CCS IDE) for the first time, use the `checkEnvSetup` function to check for third-party tools and set environment variables. Run `checkEnvSetup` again whenever you configure CCS IDE to interact with a new board or processor, or upgrade the related third-party tools.

To verify that CCSv3 is installed on your machine and has at least one board configured, enter

```
ccsboardinfo
```

in the MATLAB Command Window. With CCS installed and configured, MATLAB software returns information about the boards that CCS recognizes on your machine, in a form similar to the following listing.

Board Num	Board Name	Proc Num	Processor Name	Processor Type
1	C6xxx Simulator (Texas Instrum	.0	6701	TMS320C6701
0	C6x13 DSK (Texas Instruments)	0	CPU	TMS320C6x1x

If MATLAB software does not return information about the boards, open your CCS installation and use the Setup Utility in CCS to configure at least one board.

As a final test, start CCS to verify that it runs. For Embedded Coder to operate with CCS, the CCS IDE must be able to run on its own.

Code Composer Studio

In this section...

“Using Code Composer Studio with Embedded Coder Software” on page 39-3

“Default Project Configuration” on page 39-3

Using Code Composer Studio with Embedded Coder Software

Texas Instruments (TI) facilitates development of software for TI DSPs by offering Code Composer Studio (CCS) Integrated Development Environment (IDE). Used in combination with Embedded Coder software and Simulink Coder software, CCS provides an integrated environment that, once installed, does not require coding.

Executing code generated from Simulink Coder software on a particular target requires that you tailor the code to the specific hardware target. Target-specific code includes I/O device drivers and interrupt service routines (ISRs). The software must use CCS to compile and link the generated source code in order to load and execute on a TI DSP. To help you to build an executable, Embedded Coder software uses Embedded Coder software to start the code building process within CCS. After you download your executable to your target and run it, the code runs wholly on the target hardware. You can access the running process only from the CCS debugging tools or across a link using Embedded Coder software. A wide range of Texas Instruments DSPs are supported:

- TI's C2000™
- TI's C5000™
- TI's C6000

Default Project Configuration

CCS offers two standard project configurations, **Release** and **Debug**. Project configurations define sets of project build options. When you specify the build options at the project level, the options apply to the files in your project. For more information about the build options, refer to your TI documentation. The models you build with Embedded Coder software use a custom configuration that provides a third combination of build and optimization settings — **CustomMW**.

Default Build Options in the CustomMW Configuration

The default settings for **CustomMW** are the same as the **Release** project configuration in **CCS**, except for the compiler options.

Your **CCS** documentation provides complete details on the compiler build options. You can change the individual settings or the build configuration within **CCS**.

Getting Started

In this section...

“Overview” on page 39-5

“Verifying Your Code Composer Studio Installation” on page 39-8

Overview

- “IDE Automation Interface” on page 39-6
- “IDE Project Generator” on page 39-7
- “Verification” on page 39-7

Embedded Coder software enables you to use MATLAB functions to communicate with Code Composer Studio software and with information stored in memory and registers on a processor. With the `ticcs` objects, you can transfer information to and from Code Composer Studio software and with the embedded objects you get information about data and functions stored in your signal processor memory and registers, as well as information about functions in your project.

Embedded Coder lets you build, test, and verify automatically generated code using MATLAB, Simulink, Simulink Coder, and the Code Composer Studio integrated development environment. You can use Embedded Coder to verify code executing within the Code Composer Studio software environment using a model in Simulink software. This processor-in-the-loop testing environment uses code automatically generated from Simulink models by Embedded Coder software. A range of Texas Instruments targets are supported:

- TI's C2000
- TI's C5000
- TI's C6000

With Embedded Coder, you can use MATLAB software and Simulink software to interactively analyze, profile and debug processor-specific code execution behavior within CCS. In this way, Embedded Coder automates deployment of the complete embedded software application and makes it easy for you to assess possible differences between the model simulation and processor code execution results.

Embedded Coder consists of these components:

- IDE Project Generator—add embedded framework code to the C code generated from Simulink models, and package as a complete IDE project
- IDE Automation Interface—use functions in the MATLAB command window to access and manipulate data and files in the IDE and on the processor
- Verification—verify how your programs run on your processor

With Embedded Coder, you create objects that connect MATLAB software to Code Composer Studio software.

Note Embedded Coder uses objects. You work with them the way you use other MATLAB objects. You can set and get their properties, and use their methods to change them or manipulate them and the IDE to which they refer.

The next sections describe briefly the components of Embedded Coder software.

IDE Automation Interface

The IDE Automation Interface component is a collection of methods that use the Code Composer Studio API to communicate between MATLAB software and Code Composer Studio. With the interface, you can do the following:

- Automate complex tasks in the development environment by writing MATLAB software scripts to communicate with the IDE, or debug and analyze interactively in a live MATLAB software session.
- Automate debugging by executing commands from the powerful Code Composer Studio software command language.
- Exchange data between MATLAB software and the processor running in Code Composer Studio software.
- Set breakpoints, step through code, set parameters and retrieve profiling reports.
- Automate project creation, including adding source files, include paths, and preprocessor defines.
- Configure batch building of projects.
- Debug projects and code.
- Execute API Library commands.

The IDE Automation Interface provides an application program interface (API) between MATLAB software and Code Composer Studio. Using the API, you can create new

projects, open projects, transfer data to and from memory on the processor, add files to projects, and debug your code.

IDE Project Generator

The IDE Project Generator component is a collection of methods that use the Code Composer Studio API to create projects in Code Composer Studio and generate code with Embedded Coder. With the interface, you can do the following:

- Automated project-based build process

Automatically create and build projects for code generated by Embedded Coder.

- Customize code generation

Use Embedded Coder with a Embedded Coder system target file (STF) to generate processor-specific and optimized code.

- Customize the build process
- Automate code download and debugging

Rapidly and effortlessly debug generated code in the Code Composer Studio software debugger, using either the instruction set simulator or real hardware.

- Create and build CCS projects from Simulink software models. IDE Project Generator uses Simulink Coder software or Embedded Coder software to build projects that work with C2000 software, C5000™ software, and C6000 software processors.
- Highly customized code generation with the system target file `idelink_ert.tlc` and `idelink_grt.tlc` that enable you to use the Configuration Parameters in your model to customize your generated code.
- Automate the process of building and downloading your code to the processor, and running the process on your hardware.

Verification

Verifying your processes and algorithms is an essential part of developing applications. The components of Embedded Coder combine to provide the following verification tools for you to apply as you develop your code:

- Processor-in-the-loop simulation (PIL)
- Execution profiling
- Stack profiling

Verifying Your Code Composer Studio Installation

To verify that CCS is installed on your machine and has at least one board configured, enter

```
ccsboardinfo
```

in the MATLAB Command Window. With CCS installed and configured, MATLAB software returns information about the boards that CCS recognizes on your machine, in a form similar to the following listing.

Board Num	Board Name	Proc Num	Processor Name	Processor Type
1	C6xxx Simulator (Texas Instrum	.0	6701	TMS320C6701
0	C6x13 DSK (Texas Instruments)	0	CPU	TMS320C6x1x

If MATLAB software does not return information about boards, open your CCS installation and use the Setup Utility in CCS to configure at least one board.

As a final test, start CCS to verify that it runs. For Embedded Coder to operate with CCS, the CCS IDE must be able to run on its own.

Embedded Coder uses objects to create:

- Connections to the Code Composer Studio Integrated Development Environment (CCS IDE)
- Connections to the RTDX™ (RTDX) interface. This object is a subset of the object that refers to the CCS IDE.

Concepts to know about the objects in this toolbox are covered in these sections:

- “Constructing ticcs Objects” on page 39-42
- “ticcs Properties and Property Values” on page 39-43
- “Overloaded Functions for ticcs Objects” on page 39-43

Refer to MATLAB Classes and Objects in your MATLAB documentation for more details on object-oriented programming in MATLAB software.

Many of the objects use COM server features to create handles for working with the objects. Refer to your MATLAB documentation for more information about COM as used by MATLAB software.

IDE Automation Interface

In this section...

“Getting Started with IDE Automation Interface” on page 39-9
“Getting Started with RTDX” on page 39-25
“Constructing ticcs Objects” on page 39-42
“ticcs Properties and Property Values” on page 39-43
“Overloaded Functions for ticcs Objects” on page 39-43
“ticcs Object Properties” on page 39-44
“Function List” on page 39-50

Getting Started with IDE Automation Interface

- “Introducing the IDE Automation Interface” on page 39-9
- “Selecting Your Processor” on page 39-13
- “Creating and Querying Objects for CCS IDE” on page 39-14
- “Loading Files into CCS” on page 39-16
- “Working with Projects and Data” on page 39-18
- “Closing the Links or Cleaning Up CCS IDE” on page 39-24

Introducing the IDE Automation Interface

To use an interactive example that shows how to automate interaction between IDE Link component and Texas Instruments Code Composer Studio V3.3 using MATLAB® commands, see “Automation Interface Tutorial”.

Embedded Coder provides a connection between MATLAB software and a processor in CCS. You can use objects to control and manipulate a signal processing application using the computational power of MATLAB software. This approach can help you debug and develop your application. Another possible use for automation is creating MATLAB scripts that verify and test algorithms that run in their final implementation on your production processor.

Before using the functions available with the objects, you must select a processor because objects you create are specific to a designated processor and a designated instance of CCS IDE. For multiprocessor boards or multiple board configurations of CCS, select the specific processor.

When you have one board with a single processor, the object defaults to the existing processor. For the objects, the simulator counts as a board; if you have both a board and a simulator that CCS recognizes, you must specify the processor explicitly.

To get you started using objects for CCS software, Embedded Coder includes a tutorial that introduces you to working with data and files. As you work through this tutorial, you perform the following tasks that step you through creating and using objects for CCS IDE:

- 1 Select your processor.
- 2 Create and query objects to CCS IDE.
- 3 Use MATLAB software to load files into CCS IDE.
- 4 Work with your CCS IDE project from MATLAB software.
- 5 Close the connections you opened to CCS IDE.

The tutorial provides a working process (a *workflow*) for using Embedded Coder and your signal processing programs to develop programs for a range of Texas Instruments processors.

During this tutorial, you load and run a digital signal processing application on a processor you select. The tutorial shows both writing to memory and reading from memory in the “Working with Projects and Data” on page 39-18” portion of the tutorial.

You can use the `read` and `write` methods, as described in this tutorial, to read and write data to and from your processor.

The tutorial covers the object methods and functions for Embedded Coder. The functions listed in the first table apply to CCS IDE independent of the objects — you do not need an object to use these functions. The methods listed in the second and third table requires a `ticcs` object that you use in the method syntax:

Functions for Working With Embedded Coder

The following functions do not require a `ticcs` object as an input argument:

Function	Description
<code>ccsboardinfo</code>	Return information about the boards that CCS IDE recognizes as installed on your PC.

Function	Description
<code>ticcs</code>	Construct an object to communicate with CCS IDE. When you construct the object you specify the processor board and processor.

Methods for Working with `ticcs` Objects

The methods in the following table require a `ticcs` object as an input argument:

Method	Description
<code>add</code>	Add files to active project in IDE.
<code>address</code>	Memory address and page value of symbol in IDE.
<code>build</code>	Build or rebuild current project.
<code>“display (IDE Object)”</code>	Display the properties of an object to CCS IDE and RTDX.
<code>“halt”</code>	Terminate execution of a process running on the processor.
<code>“info”</code>	Return information about the processor or information about open RTDX channels.
<code>insert</code>	Insert debug point in file.
<code>isrtdxcapable</code>	Test whether your processor supports RTDX communications.
<code>isvisible</code>	Determine whether IDE appears on desktop.
<code>“isrunning”</code>	Test whether the processor is executing a process.
<code>“list”</code>	Return various information listings from Code Composer Studio software.
<code>load</code>	Load program file onto processor.
<code>“read”</code>	Retrieve data from memory on the processor.
<code>regread</code>	Read values from processor registers.
<code>regwrite</code>	Write data values to registers on processor.

Method	Description
remove	Remove file, project, or breakpoint.
“restart”	Restore the program counter (PC) to the entry point for the current program.
“run”	Execute the program loaded on the processor.
“visible”	Set whether CCS IDE window is visible on the desktop while CCS IDE is running.
“write”	Write data to memory on the processor.

Running Code Composer Studio Software on Your Desktop — Visibility

When you create a `ticcs` object, Embedded Coder starts CCS in the background.

When CCS IDE is running in the background, it does not appear on your desktop, in your task bar, or on the **Applications** page in the Task Manager. It does appear as a process, `cc_app.exe`, on the **Processes** tab in Microsoft Windows Task Manager.

You can make the CCS IDE visible with the function `visible`. The function `invisible` returns the status of the IDE—whether it is visible on your desktop. To close the IDE when it is not visible and MATLAB software is not running, use the **Processes** tab in Microsoft Windows Task Manager and look for `cc_app.exe`.

If a link to CCS IDE exists when you close CCS, the application does not close. Microsoft Windows software moves it to the background (it becomes invisible). Only after you clear links to CCS IDE, or close MATLAB software, does closing CCS IDE unload the application. You can see if CCS IDE is running in the background by checking in the Microsoft Windows Task Manager. When CCS IDE is running, the entry `cc_app.exe` appears in the **Image Name** list on the **Processes** tab.

When you close MATLAB software while CCS IDE is not visible, MATLAB software closes CCS if it started the IDE. This happens because the operating system treats CCS as a subprocess in MATLAB software when CCS is not visible. Having MATLAB software close the invisible IDE when you close MATLAB software prevents CCS from remaining open. You do not need to close it using Microsoft Windows Task Manager.

If CCS IDE is not visible when you open MATLAB software, closing MATLAB software leaves CCS IDE running in an invisible state. MATLAB software leaves CCS IDE in the visibility and operating state in which it finds it.

Interactive Learning

You have the option of running this tutorial from the MATLAB Command Window or entering the functions as described in the following tutorial sections.

To run the tutorial in MATLAB software, click `run ccstutorial`. This command opens the tutorial in an interactive mode where the tutorial program provides prompts and text descriptions to which you respond to move to the next portion of the lesson. The interactive tutorial covers the same information provided by the following tutorial sections. You can view the tutorial file by clicking `ccstutorial.m`.

Selecting Your Processor

Links for CCS IDE provides two tools for selecting a board and processor in multiprocessor configurations. One is a command line tool called “`ccsboardinfo`” which prints a list of the available boards and processors. So that you can use this function in a script, `ccsboardinfo` can return a MATLAB software structure that you use when you want your script to select a board without your help.

Note The board and processor you select is used throughout the tutorial.

- 1 To see a list of the boards and processors installed on your PC, enter the following command at the MATLAB software prompt:

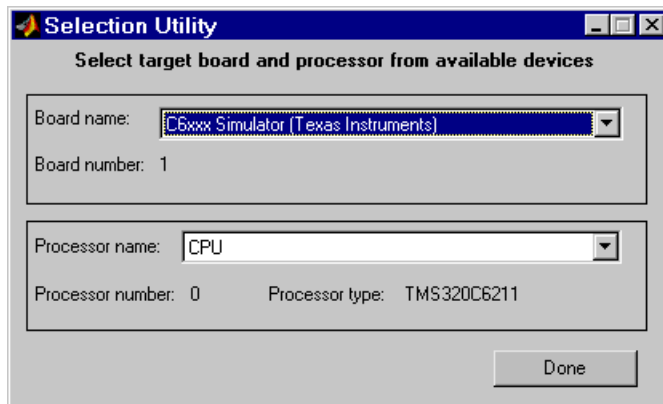
```
ccsboardinfo
```

MATLAB software returns a list that shows you the boards and processors that CCS IDE recognizes as installed on your system.

- 2 To use the Selection Utility, `boardprocse1`, to select a board, enter

```
[boardnum,procnum] = boardprocse1
```

When you use `boardprocse1`, you see a dialog similar to the following. Note that some entries vary depending on your board set.



- 3 Select a board name and processor name from the lists.

You are selecting a board and processor number that identifies your particular processor. When you create the object for CCS IDE in the next section of this tutorial, the selected board and processor become the processor of the object.

- 4 Click **Done** to accept your board and processor selection and close the dialog.

`boardnum` and `procnum` now represent the **Board name** and **Processor name** you selected — `boardnum = 1` and `procnum = 0`

Creating and Querying Objects for CCS IDE

In this tutorial section, you create the connection between MATLAB software and CCS IDE. This connection, or object, is a MATLAB software object that you save as variable `IDE_Obj`.

You use function `ticcs` to create objects. When you create objects, `ticcs` input arguments let you define other object property values, such as the global timeout. Refer to the `ticcs` reference documentation for more information on these input arguments.

Use the generated object `IDE_Obj` to direct actions to your processor. In the following tasks, `IDE_Obj` appears in function syntax that interacts with CCS IDE and the processor:

- 1 Create an object that refers to your selected board and processor. Enter the following command at the prompt.

```
IDE_Obj=ticcs('boardnum',boardnum,'procnum',procnum)
```

If you were to watch closely, and your machine is not too fast, you see Code Composer Studio software appear briefly when you call “ticcs”. If CCS IDE was not running before you created the new object, CCS starts and runs in the background.

- 2 Enter `visible(IDE_Obj,1)` to force CCS IDE to be visible on your desktop.

Usually, you need to interact with Code Composer Studio software while you develop your application. The first function in this tutorial, “visible”, controls the state of CCS on your desktop. `visible` accepts Boolean inputs that make CCS either visible on your desktop (input to `visible = 1`) or invisible on your desktop (input to `visible = 0`). For this tutorial, use `visible` to set the CCS IDE visibility to 1.

- 3 Next, enter `display(IDE_Obj)` at the prompt to see the status information.

```
TICCS Object:
  Processor type   : Cxx
  Processor name  : CPU
  Running?        : No
  Board number    : 0
  Processor number : 0
  Default timeout : 10.00 secs

  RTDX channels   : 0
```

Embedded Coder provides methods to read the status of a board and processor:

- `info` — Return a structure of testable board conditions.
- `display` — Print information about the processor.
- `isrunning` — Return the state (running or halted) of the processor.
- `isrtdxcapable` — Return whether the hardware supports RTDX.

- 4 Type `linkinfo = info(IDE_Obj)`.

The `IDE_Obj` link status information provides information about the hardware as follows:

```
linkinfo =
    procname: 'CPU_1'
    isbigendian: 0
    isrtdxcapable: 0
    family: 320
```

```
    subfamily: 103
    revfamily: 11
    targettype: 'simulator'
    siliconrev: 0
        timeout: 10
    boardname: 'Cxxxx Device Simulator'
```

- 5 Check whether the processor is running by entering

```
runstatus = isrunning(IDE_Obj)
```

MATLAB software responds, indicating that the processor is stopped, as follows:

```
runstatus =
```

```
    0
```

- 6 At last, verify that the processor supports RTDX communications by entering

```
usesrtdx = isrtdxcapable(IDE_Obj)
usesrtdx =
```

```
    1
```

Loading Files into CCS

You have established the connection to a processor and board and have created and queried objects. Next, the processor needs something to do.

In this part of the tutorial, you load the executable code for the processor CPU in CCS IDE. Embedded Coder includes a CCS project file. Through the next tasks in the tutorial, you locate the tutorial project file and load it into CCS IDE. The `open` method directs CCS to load a project file or workspace file.

Note CCS has workspace and workspace files that are different from the MATLAB workspace files and workspace. Remember to monitor both workspaces.

After you have executable code running on your processor, you can exchange data blocks with it. Exchanging data is the purpose of the objects provided by Embedded Coder software.

- 1 To load the project file to your processor, enter the following command at the MATLAB software prompt. `getdemoproject` is a specialized function for loading

Embedded Coder example files. It is not supported as a standard Embedded Coder function.

```
demopjt= getDemoProject(IDE_Obj,'ccstutorial')
demopjt.ProjectFile
ans =
C:\Temp\EmbIDELinkCCDemos_v4.1\ccstutorial\cxx\cxx\ccstut.pjt
demoPjt.DemoDir
ans =
C:\Temp\EmbIDELinkCCDemos_v4.1\ccstutorial\cxx\cxx
```

Your paths may be different if you use a different processor. Note where the software stored the example files on your machine. In general, Embedded Coder software stores the example project files in

EmbIDELinkCCDemos_v#. #

Embedded Coder creates this folder in a location where you have write permission. There are two locations where Embedded Coder software tries to create the example folder, in the following order:

- a** In a temporary folder on your C drive, such as `C:\temp\`.
 - b** If Embedded Coder software cannot use the `temp` folder, you see a dialog that asks you to select a location to store the examples.
- 2** Enter the following command at the MATLAB command prompt to build the processor executable file in CCS IDE.

```
build(IDE_Obj, 'all', 20)
```

You may get an error related to one or more missing `.lib` files. If you installed CCS IDE in a folder other than the default installation folder, browse in your installation folder to find the missing file or files. Refer to the path in the error message as an indicator of where to find the missing files.

- 3** Change your working folder to the example folder and enter `load(IDE_Obj, 'projectname.out')` to load the processor execution file, where *projectname* is the tutorial you chose, such as `ccstut_67x`.

You have a loaded program file and associated symbol table to the IDE and processor.

- 4** To determine the memory address of the global symbol `ddat`, enter the following command at the prompt:

```
ddata = address(IDE_Obj, 'ddat')
ddata =

    1.0e+009 *
    2.1475      0
```

Your values for `ddata` may be different depending on your processor.

Note The symbol table is available after you load the program file into the processor, not after you build a program file.

- 5 To convert `ddata` to a hexadecimal string that contains the memory address and memory page, enter the following command at the prompt:

```
dec2hex(ddata)
```

MATLAB software displays the following response, where the memory page is `0x00000000` and the address is `0x80000010`.

```
ans =

    80000010
    00000000
```

Working with Projects and Data

After you load the processor code, you can use Embedded Coder functions to examine and modify data values in the processor.

When you look at the source file listing in the CCS IDE Project view window, there should be a file named `ccstut.c`. Embedded Coder ships this file with the tutorial and includes it in the project.

`ccstut.c` has two global data arrays — `ddat` and `idat` — that you declare and initialize in the source code. You use the functions `read` and `write` to access these processor memory arrays from MATLAB software.

Embedded Coder provides three functions to control processor execution — `run`, `halt`, and `restart`.

- 1 To see these commands, use the following function to add a breakpoint to line 68 of `ccstut.c`.

```
insert(IDE_Obj, 'ccstut.c', 68)
```

Line 68 is

```
printf("Embedded Coder: Tutorial - Memory Modified by Matlab!\n");
```

For information about adding breakpoints to a file, refer to `insert` in the online Help system. Then proceed with the tutorial.

- 2 To see the new functions, try the following functions.

```
halt(IDE_Obj)           % Halt the processor.
restart(IDE_Obj)        % Reset the PC to start of program.
run(IDE_Obj, 'runthalt', 30); % Wait for program execution to stop at
                           % breakpoint (timeout = 30 seconds).
```

When you switch to viewing CCS IDE, you see that your program stopped at the breakpoint you inserted, and the program printed the following messages in the CCS IDE **Stdout** tab. Nothing prints in the MATLAB command window:

```
Embedded Coder: Tutorial - Initialized Memory
Double Data array = 16.3 -2.13 5.1 11.8
Integer Data array = -1-508-647-7000 (call me anytime!)
```

- 3 Before you restart your program (currently stopped at line 68), change some values in memory. Perform one of the following procedures based on your processor.

C5xxx processor family — Enter the following functions to see the `read` and `write` functions.

- a Enter `ddatv = read(IDE_Obj, address(IDE_Obj, 'ddat'), 'double', 4)`.

MATLAB software responds with

```
ddatv =
```

```
    16.3000   -2.1300    5.1000   11.8000
```

- b Enter `idatv = read(IDE_Obj, address(IDE_Obj, 'idat'), 'int16', 4)`.

Now MATLAB software responds

```
idatv =
```

```
-1 508 647 7000
```

If you used 8-bit integers (`int8`), the returned values would be incorrect.

```
idatv=read(IDE_Obj,address(IDE_Obj,'idat'),'int8',4)
```

```
idatv =
```

```
1 0 -4 1
```

- c** You can change the values stored in `ddat` by entering

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

The `double` argument directs MATLAB software to write the values to the processor as double-precision data.

- d** To change `idat`, enter

```
write(IDE_Obj,address(IDE_Obj,'idat'),int32([1:4]))
```

Here you write the data to the processor as 32-bit integers (convenient for representing phone numbers, for example).

- e** Start the program running again by entering the following command:

```
run(IDE_Obj,'runtohalt',30);
```

The `Stdout` tab in CCS IDE reveals that `ddat` and `idat` contain new values. Next, read those new values back into MATLAB software.

- f** Enter `ddatv = read(IDE_Obj,address(IDE_Obj,'ddat'),'double',4)`.

```
ddatv =
```

```
3.1416 12.3000 0.3679 0.7071
```

`ddatv` contains the values you wrote in step **c**.

- g** Verify that the change to `idatv` occurred by entering the following command at the prompt:

```
idatv = read(IDE_Obj,address(IDE_Obj,'idat'),'int16',4)
```

MATLAB software returns the new values for `idatv`.

```
idatv =
```

```
1 2 3 4
```

- h** Use `restart` to reset the program counter for your program to the beginning. Enter the following command at the prompt:

```
restart(IDE_Obj);
```

C6xxx processor family — Enter the following commands to see the `read` and `write` functions.

- a** Enter `ddatv = read(IDE_Obj,address(IDE_Obj,'ddat'),'double',4)`.

MATLAB software responds with

```
ddatv =
```

```
16.3000 -2.1300 5.1000 11.8000
```

- b** Enter `idatv = read(IDE_Obj,address(IDE_Obj,'idat'),'int16',4)`.

MATLAB software responds

```
idatv =
```

```
-1 508 647 7000
```

If you used 8-bit integers (`int8`), the returned values would be incorrect.

```
idatv=read(IDE_Obj,address(IDE_Obj,'idat'),'int8',4)
```

```
idatv =
```

```
1 0 -4 1
```

- c** Change the values stored in `ddat` by entering `write(IDE_Obj,address(IDE_Obj,'ddat'),double([pi 12.3 exp(-1)... sin(pi/4)]))`

The `double` argument directs MATLAB software to write the values to the processor as double-precision data.

- d** To change `idat`, enter the following command:

```
write(IDE_Obj, address(IDE_Obj, 'idat'), int32([1:4]))
```

In this command, you write the data to the processor as 32-bit integers (convenient for representing phone numbers, for example).

- e** Next, start the program running again by entering the following command:

```
run(IDE_Obj, 'runtohalt', 30);
```

The **Stdout** tab in CCS IDE reveals that `ddat` and `idat` contain new values. Read those new values back into MATLAB software.

- f** Enter `ddatv = read(IDE_Obj, address(IDE_Obj, 'ddat'), 'double', 4)`.

```
ddatv =
```

```
3.1416 12.3000 0.3679 0.7071
```

Verify that `ddatv` contains the values you wrote in step c.

- g** Verify that the change to `idatv` occurred by entering the following command:

```
idatv = read(IDE_Obj, address(IDE_Obj, 'idat'), 'int32', 4)
```

MATLAB software returns the new values for `idatv`.

```
idatv =
```

```
1 2 3 4
```

- h** Use `restart` to reset the program counter for your program to the beginning. Enter the following command at the prompt:

```
restart(IDE_Obj);
```

- 4** Embedded Coder offers more functions for reading and writing data to your processor. These functions let you read and write data to the processor registers: `regread` and `regwrite`. They let you change variable values on the processor in real time. The functions behave slightly differently depending on your processor.

Select one of the following procedures to see `regread` and `regwrite` used with your processor.

C5xxx processor family — Most registers are memory-mapped and available using `read` and `write`. However, the PC register is not memory mapped. To access this register, use the special functions — `regread` and `regwrite`. The following commands show how to use these functions to read and write to the PC register.

- a** To read the value stored in register PC, enter the following command at the prompt to indicate to MATLAB software the data type to read. The input string `binary` indicates that the PC register contains a value stored as an unsigned binary integer.

```
IDE_Obj.regread('PC','binary')
```

MATLAB software displays

```
ans =
```

```
33824
```

- b** To write a new value to the PC register, enter the following command. This time, the `binary` input argument tells MATLAB software to write the value to the processor as an unsigned binary integer. Notice that you used `hex2dec` to convert the hexadecimal string to decimal.

```
IDE_Obj.regwrite('PC',hex2dec('100'),'binary')
```

- c** Verify that the PC register contains the value you wrote.

```
IDE_Obj.regread('PC','binary')
```

C6xxx processor family — `regread` and `regwrite` let you access the processor registers directly. Enter the following commands to get data into and out of the A0 and B2 registers on your processor.

- a** To retrieve the value in register A0 and store it in a variable in your MATLAB workspace. Enter the following command:

```
treg = IDE_Obj.regread('A0','2scomp');
```

`treg` contains the two's complement representation of the value in A0.

- b To retrieve the value in register B2 as an unsigned binary integer, enter the following command:

```
IDE_Obj.regread('B2','binary');
```

- c Next, enter the following command to use `regwrite` to put the value in `treg` into register A2.

```
IDE_Obj.regwrite('A2',treg,'2scomp');
```

CCS IDE reports that A0, B2, and A2 have the values you expect. Select **View > CPU Registers > Core Registers** from the CCS IDE menu bar to list the processor registers.

Closing the Links or Cleaning Up CCS IDE

Objects that you create in Embedded Coder software have COM handles to CCS. Until you delete these handles, the CCS process (`cc_app.exe` in the Microsoft Windows Task Manager) remains in memory. Closing MATLAB software removes these COM handles, but there may be times when you want to delete the handles without closing the application.

Use `clear` to remove objects from your MATLAB workspace and to delete handles they contain. `clear all` deletes everything in your workspace. To retain your MATLAB software data while deleting objects and handles, use `clear objname`. This applies to IDE link handle objects you created with `ticcs`. To remove the objects created during the tutorial, the tutorial program executes the following command at the prompt:

```
clear cvar cfield uintcvar
```

This tutorial also closes the project in CCS with the following command:

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

To delete your link to CCS, enter `clear IDE_Obj` at the prompt.

Your development tutorial using Code Composer Studio IDE is done.

During the tutorial you

- 1 Selected your processor.
- 2 Created and queried links to CCS IDE to get information about the link and the processor.

- 3 Used MATLAB software to load files into CCS IDE, and used MATLAB software to run that file.
- 4 Worked with your CCS IDE project from MATLAB software by reading and writing data to your processor, and changing the data from MATLAB software.
- 5 Created and used the embedded objects to manipulate data in a C-like way.
- 6 Closed the links you opened to CCS IDE.

Getting Started with RTDX

- “Using RTDX” on page 39-26
- “Creating the ticcs Objects” on page 39-30
- “Configuring Communications Channels” on page 39-32
- “Running the Application” on page 39-33
- “Closing the Connections and Channels or Cleaning Up” on page 39-39
- “Listing Functions” on page 39-41

Texas Instruments Real-Time Data Exchange (RTDX) provides “real-time, continuous visibility into the way target applications operate in the real world. RTDX allows system developers to transfer data between target devices and a host without interfering with the target application.”

You can use RTDX with Embedded Coder software and Code Composer Studio to accelerate development and deployment to Texas Instruments C2000 processors. RTDX helps you test and analyze your processing algorithms in your MATLAB workspace. RTDX lets you interact with your process in real time while it's running on the processor. For example, you can:

- Send and retrieve data from memory on the processor
- Change the operating characteristics of the program
- Make changes to algorithms as required without stopping the program or setting breakpoints in the code

Enabling real-time interaction lets you more easily see your process or algorithm in action, the results as they develop, and the way the process runs.

This tutorial assumes you have Texas Instruments Code Composer Studio software and at least one target development board. You can use the hardware simulator in CCS IDE to run this tutorial.

After you complete the tutorial, you can start using RTDX with your applications and hardware.

Note: To use RTDX with the XDS100 USB JTAG Emulator and the C28027 chip, add the following line to the linker command file:

```
_RTDX_interrupt_mask = ~0x00000008;
```

Using RTDX

Digital signal processing development efforts begin with an idea for processing data; an application area, such as audio or wireless communications or multimedia computing; and a platform or hardware to host the signal processing. Usually these processing efforts involve applying strategies like signal filtering, compression, and transformation to change data content; or isolate features in data; or transfer data from one form to another or one place to another.

Developers create algorithms they need to accomplish the desired result. After they have the algorithms, they use models and target hardware development tools to test their algorithms, to determine whether the processing achieves the goal, and whether the processing works on the proposed platform.

Embedded Coder and the links for RTDX and CCS IDE ease the job of taking algorithms from the model realm to the real world of the processor on which the algorithm runs.

RTDX and links for CCS IDE provide a communications pathway to manipulate data and processing programs on your processor. RTDX offers real-time data exchange in two directions between MATLAB software and your processor process. Data you send to the processor do little to alter running processes. Plotting data you retrieve from the processor lets you see how your algorithms are performing in real time.

To introduce the techniques and tools available in Embedded Coder for using RTDX, the following procedures use many of the methods in the link software to configure the processor, open and enable channels, send data to the processor, and clean up after you finish your testing. Among the functions covered are:

Functions From Objects for CCS IDE

Function	Description
ticcs	Create connections to CCS IDE and RTDX.

Function	Description
cd	Change the CCS IDE working folder from MATLAB software.
open	Load program files in CCS IDE.
run	Run processes on the processor.

Functions From the RTDX Class

Function	Description
close	Close the RTDX links between MATLAB software and your processor.
configure	Determine how many channel buffers to use and set the size of each buffer.
disable	Disable the RTDX links before you close them.
display	Return the properties of an object in formatted layout. When you omit the closing semicolon on a function, <code>disp</code> (a built-in function) provides the default display for the results of the operation.
enable	Enable open channels so you can use them to send and retrieve data from your processor.
isenabled	Determine whether channels are enabled for RTDX communications.
isreadable	Determine whether MATLAB software can read the specified memory location.
iswritable	Determine whether MATLAB software can write to the processor.
msgcount	Determine how many messages are waiting in a channel queue.
open	Open channels in RTDX.
readmat	Read data matrices from the processor into MATLAB software as an array.

Function	Description
readmsg	Read one or more messages from a channel.
writemsg	Write messages to the processor over a channel.

This tutorial provides the following workflow to show you how to use many of the functions in the links. By performing the steps provided, you work through many of the operations yourself. The tutorial follows the general task flow for developing digital signal processing programs through testing with the links for RTDX.

Within this set of tasks, numbers 1, 2, and 4 are fundamental to function syntax that interacts development projects. Whenever you work with MATLAB software and objects for RTDX, you perform the functions and tasks outlined and presented in this tutorial. The differences lie in Task 3. Task 3 is the most important for using Embedded Coder to develop your processing system.

- 1 Create an RTDX link to your desired processor and load the program to the processor.

The projects begin this way. Without the links you cannot load your executable to the processor.

- 2 Configure channels to communicate with the processor.

Creating the links in Task 1 did not open communications to the processor. With the links in place, you open as many channels as you need to support the data transfer for your development work. This task includes configuring channel buffers to hold data when the data rate from the processor exceeds the rate at which MATLAB software can capture the data.

- 3 Run your application on the processor. You use MATLAB software to investigate the results of your running process.
- 4 Close the links to the processor and clean up the links and associated debris left over from your work.

Closing channels and cleaning up the memory and links you created prepares CCS IDE, RTDX, and Embedded Coder for the next time you start development on a project.

This tutorial uses an executable program named `rtdxtutorial_6xevm.out` as your application. When you use the RTDX links and CCS IDE to develop your own

applications, replace `rtdxtutorial_6xevm.out` in Task 3 with the filename and path to your digital signal processing application.

You can view the tutorial file used here by clicking `rtdxtutorial`. To run this tutorial in MATLAB software, click run `rtdxtutorial`.

Note To be able to open and enable channels over a link to RTDX, the program loaded on your processor must include functions or code that define the channels.

Your C source code might look something like this to create two channels, one to write and one to read.

```
rtdx_CreateInputChannel(ichan); % processor reads from this.  
rtdx_CreateOutputChannel(ochan); % processor writes to this.
```

These are the entries we use in `int16.c` (the source code that generates `rtdxtutorial_6xevm.out`) to create the read and write channels.

If you are working with a model in Simulink software and using code generation, use the To Rtdx and From Rtdx blocks in your model to add the RTDX communications channels to your model and to the executable code on your processor.

One more note about this tutorial. Throughout the code we use both the dot notation (direct property referencing) to access functions and link properties and the function form.

For example, use the following command to open and configure `ichan` for write mode.

```
IDE_Obj.rtdx.open('ichan','w');
```

You could use an equivalent syntax, the function form, that does not use direct property referencing.

```
open(IDE_Obj.rtdx,'ichan','w');
```

Or, use

```
open(rx,'ichan','w');
```

if you created an alias `rx` to the RTDX portion of `IDE_Obj`, as shown by the following command:

```
rx = IDE_Obj.rtdx;
```

Creating the ticcs Objects

With your processing model converted to an executable for your desired processor, you are ready to use the objects to test and run your model on your processor. Embedded Coder and the objects do not distinguish the source of the executable — whether you used Embedded Coder, CCS IDE, or some other development tool to program and compile your model to an executable does not alter the object connections. So long as your `.out` file is acceptable to the processor you select, Embedded Coder provides the connection to the processor.

Before continuing with this tutorial, you must load a valid GEL file to configure the EMIF registers of your processor and perform required processor initialization steps. Default GEL files provided by CCS are stored in `.. \IDE_Obj \gel` in the folder where you installed CCS software. Select **File > Load_GEL** in CCS IDE to load the default GEL file that matches your processor family, such as `init6x0x.gel` for the Cxxxx processor family, and your configuration.

Note: If you are performing the steps in this tutorial, create `demoPjt` as described in “Loading Files into CCS” on page 39-16 before continuing.

Begin the process of getting your model onto the processor by creating a an object that refers to CCS IDE. Start by clearing function syntax that interacts existing handles and setting echo on so you see functions execute as the program runs:

```
1 clear all; echo on;
```

`clear all` removes debugging breakpoints and resets persistent variables because function breakpoints and persistent variables are cleared whenever the MATLAB file changes or is cleared. Breakpoints within your executable remain after `clear`. Clearing the MATLAB workspace does not alter your executable.

```
2 Now construct the link to your board and processor by entering
```

```
IDE_Obj=ticcs('boardnum',0);
```

`boardnum` defines which board the new link accesses. In this example, `boardnum` is 0. Embedded Coder connects the link to the first, and in this case only, processor on the board. To find the `boardnum` and `procnum` values for the boards and simulators on your system, use “`ccsboardinfo`”. When you enter the following command at the prompt

```
ccsboardinfo
```

- 3 To open and load the processor file, change the path for MATLAB software to be able to find the file.

```
projname = C:\Temp\EmbIDELinkCCDemos_v4.1\rtdxtutorial\cxx\cxxxp\rtdxtut_sim.pjt

outfile = C:\Temp\EmbIDELinkCCDemos_v4.1\rtdxtutorial\cxx\cxxxp\rtdxtut_sim.out

processor_dir = demoPjt.DemoDir

processor_dir = C:\Temp\EmbIDELinkCCDemos_v4.1\rtdxtutorial\cxx\cxxxp

cd(IDE_Obj,processor_dir); % Go to processor directory
cd(IDE_Obj,tgt_dir); % Or IDE_Obj.cd(tgt_dir)
dir(IDE_Obj); % Or IDE_Obj.dir
```

To load the project file to your processor, enter the following commands at the MATLAB software prompt. `getDemoProject` is a specialized function for loading Embedded Coder example files. It is not supported as a standard Embedded Coder function.

```
demoPjt = getDemoProject(IDE_Obj,'ccstutorial');

demoPjt.ProjectFile

ans = C:\Temp\EmbIDELinkCCDemos_v4.1\ccstutorial\cxx\cxxxp\ccstut.pjt

demoPjt.DemoDir

ans = C:\Temp\EmbIDELinkCCDemos_v4.1\ccstutorial\cxx\cxxxp
```

Notice where the example files are stored on your machine. In general, Embedded Coder software stores the example project files in

```
EmbIDELinkCCDemos_v#.#
```

For example, if you are using version 4.1 of Embedded Coder software, the project examples are stored in `EmbIDELinkCCDemos_v4.1\`. Embedded Coder software creates this folder in a location on your machine where you have write permission. Usually, there are two locations where Embedded Coder software tries to create the example folder, in the order shown.

- a In a temporary folder on the C drive, such as `C:\temp\`.
 - b If Embedded Coder software cannot use the `temp` folder, you see a dialog that asks you to select a location to store the examples.
- 4 You have reset the folder path to find the tutorial file. Now open the `.out` file that matches your processor type.

```
IDE_Obj.open('rtdxtutorial_xxx.out')
```

Because `open` is overloaded for the CCS IDE and RTDX links, this may seem a bit strange. In this syntax, `open` loads your executable file onto the processor identified by `IDE_Obj`. Later in this tutorial, you use `open` with a different syntax to open channels in RTDX.

In the next section, you use the new link to open and enable communications between MATLAB software and your processor.

Configuring Communications Channels

Communications channels to the processor do not exist until you open and enable them through Embedded Coder and CCS IDE. Opening channels consists of opening and configuring each channel for reading or writing, and enabling the channels.

In the `open` function, you provide the channel names as strings for the channel name property. The channel name you use is not random. The channel name string must match a channel defined in the executable file. If you specify a string that does not identify an existing channel in the executable, the open operation fails.

In this tutorial, two channels exist on the processor — `ichan` and `ochan`. Although the channels are named `ichan` for input channel and `ochan` for output channel, neither channel is configured for input or output until you configure them from MATLAB software or CCS IDE. You could configure `ichan` as the output channel and `ochan` as the input channel. The links would work just the same. For simplicity, the tutorial configures `ichan` for input and `ochan` for output. One more note—reading and writing are defined as seen by the processor. When you write data from MATLAB software, you write to the channel that the processor reads, `ichan` in this case. Conversely, when you read from the processor, you read from `ochan`, the channel that the processor writes to:

- 1 Configure buffers in RTDX to store the data until MATLAB software can read it into your workspace. Often, MATLAB software cannot read data as quickly as the processor can write it to the channel.

```
IDE_Obj.rtdx.configure(1024,4); % define 4 channels of 1024 bytes each
```

Channel buffers are optional. Adding them provides a measure of insurance that data gets from your processor to MATLAB software without getting lost.

- 2 Define one of the channels as a write channel. Use `'ichan'` for the channel name and `'w'` for the mode. Either `'w'` or `'r'` fits here, for write or read.

```
IDE_Obj.rtdx.open('ichan','w');
```

- 3 Now enable the channel you opened.


```
IDE_Obj.rtdx.enable('ichan');
```

- 4 Repeat steps 2 and 3 to prepare a read channel.

```
IDE_Obj.rtdx.open('ochan', 'r');
IDE_Obj.rtdx.enable('ochan');
```

- 5 To use the new channels, enable RTDX by entering

```
IDE_Obj.rtdx.enable;
```

You could do this step before you configure the channels — the order does not matter.

- 6 Reset the global time-out to 20s to provide a little room for error. `ticcs` applies a default timeout value of 10 s. In some cases this may not be enough.

```
IDE_Obj.rtdx.get('timeout')
ans =
    10
IDE_Obj.rtdx.set('timeout', 20); % Reset timeout = 20 seconds
```

- 7 Check that the `timeout` property value is now 20s and that your object has a valid configuration for the rest of the tutorial.

```
IDE_Obj.rtdx
```

```
RTDX Object:
  API version:      1.0
  Default timeout:  20.00 secs
  Open channels:    2
```

Running the Application

To this point you have been doing common housekeeping functions. You load the processor, configure the communications, and set up other properties you need.

This tutorial shows you some of the Embedded Coder functions you can use to prototype and experiment with your application. To see the RTDX “`readmat`”, `readmsg`, and `writemsg` functions, you write data to your processor. Then, after the data has been processed, you read data from the processor.

- 1 Restart the program you loaded on the processor. `restart` sets the program counter (PC) to the beginning of the executable code on the processor.

```
IDE_Obj.restart
```

Restarting the processor does not start the program executing. You use `run` to start program execution.

2 Type `IDE_Obj.run('run');`

Using 'run' for the run mode tells the processor to continue to execute the loaded program continuously until it receives a halt directive. In this mode, control returns to MATLAB software so you can work in MATLAB software while the program runs. Other options for the mode are

- 'runtohalt' — start to execute the program and wait to return control to MATLAB software until the process reaches a breakpoint or execution terminates.
- 'tohalt' — change the state of a running processor to 'runtohalt' and wait to return until the program halts. Use `tohalt` mode to stop the running processor cleanly.

3 Type the following functions to enable the write channel and verify the change:

```
IDE_Obj.rtdx.enable('ichan');  
IDE_Obj.rtdx.isenabled('ichan')
```

If MATLAB software responds `ans = 0` your channel is not enabled and you cannot proceed with the tutorial. Try to enable the channel again and verify the status.

4 Write some data to the processor. Check that you can write to the processor, then use `writemsg` to send the data. You do not need to enter the if-test code shown.

```
if IDE_Obj.rtdx.iswritable('ichan'), % Used in a script application.  
    disp('writing to processor...') % Optional to display progress.  
    indata=1:10  
    IDE_Obj.rtdx.writemsg('ichan', int16(indata))  
end % Used in scripts for channel testing.
```

The `if` statement simulates writing the data from within a MATLAB software script. The script uses `iswritable` to check that the input channel is functioning. If `iswritable` returns `0` the script would skip the write and exit the program, or respond in some way. When you are writing or reading data to your processor in a script or MATLAB file, checking the status of the channels can help you avoid errors during execution.

As your application runs you may find it helpful to display progress messages. In this case, the program directed MATLAB software to print a message as it reads the data from the processor by adding the function

```
disp('writing to processor...')
```

Function `IDE_Obj.rtdx.writemsg('ichan', int16(indata))` results in 20 messages stored on the processor. Here's how.

When you write `indata` to the processor, the following code running on the processor takes your input data from `ichan`, adds one to the values and copies the data to memory:

```
while ( !RTDX_isInputEnabled(&ichan) )

{ /* wait for channel enable from MATLAB */
RTDX_read( &ichan, recvd, sizeof(recvd) );
puts("\n\n Read Completed ");

for (j=1; j<=20; j++) {
  for (i=0; i<MAX; i++) {
    recvd[i] +=1;
  }
  while ( !RTDX_isOutputEnabled(&ochan) )
    { /* wait for channel enable from MATLAB */ }
  RTDX_write( &ochan, recvd, sizeof(recvd) );
  while ( RTDX_writing != NULL )
    { /* wait for data xfer INTERRUPT DRIVEN for Cxxxx */ }
}
```

Program `int16_rtdx.c` contains this source code. You can find the file in a folder in the `..\tidemos\rtdxtutorial` folder.

- 5 Type the following to check the number of available messages to read from the processor.

```
num_of_msgs = IDE_Obj.rtdx.msgcount('ochan');
```

`num_of_msgs` should be zero. Using this process to check the amount of data lets you or your program know how much data to expect.

- 6 Type the following to verify that your read channel `ochan` is enabled for communications.

```
IDE_Obj.rtdx.isenabled('ochan')
```

You should get back `ans = 0` — you have not enabled the channel yet.

- 7 Now enable and verify 'ochan'.

```
IDE_Obj.rtdx.enable('ochan');
IDE_Obj.rtdx.isenabled('ochan')
```

To show that `ochan` is ready, MATLAB software responds `ans = 1`. If not, try enabling `ochan` again.

8 Type

```
pause(5);
```

The `pause` function gives the processor extra time to process the data in `indata` and transfer the data to the buffer you configured for `ochan`.

9 Repeat the check for the number of messages in the queue. There should be 20 messages available in the buffer.

```
num_of_msgs = IDE_Obj.rtdx.msgcount('ochan')
```

With `num_of_msgs = 20`, you could use a looping structure to read the messages from the queue in to MATLAB software. In the next few steps of this tutorial you read data from the `ochan` queue to different data formats within MATLAB software.

10 Read one message from the queue into variable `outdata`.

```
outdata = IDE_Obj.rtdx.readmsg('ochan','int16')
```

```
outdata =
     2     3     4     5     6     7     8     9    10    11
```

Notice the `'int16'` `represent` option. When you read data from your processor you need to tell MATLAB software the data type you are reading. You wrote the data in step 4 as 16-bit integers so you use the same data type here.

While performing reads and writes, your process continues to run. You did not need to stop the processor to get the data or send the data, unlike using most debuggers and breakpoints in your code. You placed your data in memory across an RTDX channel, the processor used the data, and you read the data from memory across an RTDX channel, without stopping the processor.

11 You can read data into cell arrays, rather than into simple double-precision variables. Use the following function to read three messages to cell array `outdata`, an array of three, 1-by-10 vectors. Each message is a 1-by-10 vector stored on the processor.

```
outdata = IDE_Obj.rtdx.readmsg('ochan','int16',3)
```

```

outdata =
 [1x10 int16] [1x10 int16] [1x10 int16]

```

- 12** Cell array `outdata` contains three messages. Look at the second message, or matrix, in `outdata` by using dereferencing with the array.

```
outdata{1,2}
```

```

outdata =
     4     5     6     7     8     9    10    11    12    13

```

- 13** Read two messages from the processor into two 2-by-5 matrices in your MATLAB workspace.

```
outdata = IDE_Obj.rtdx.readmsg('ochan','int16',[2 5],2)
```

```

outdata =
 [2x5 int16] [2x5 int16]

```

To specify the number of messages to read and the data format in your workspace, you used the `siz` and `nummsgs` options set to `[2 5]` and `2`.

- 14** You can look at both matrices in `outdata` by dereferencing the cell array again.

```
outdata{1,:}
```

```

ans =
     6     8    10    12    14
     7     9    11    13    15
ans =
     7     9    11    13    15
     8    10    12    14    16

```

- 15** For a change, read a message from the queue into a column vector.

```
outdata = IDE_Obj.rtdx.readmsg('ochan','int16',[10 1])
```

```

outdata =
     8
     9
    10
    11
    12
    13
    14
    15
    16
    17

```

- 16** Embedded Coder provides a function for reading messages into matrices—“`readmat`”. Use `readmat` to read a message into a 5-by-2 matrix in MATLAB software.

```
outdata = readmat(IDE_Obj.rtdx,'ochan','int16',[5 2])
```

```
outdata =
```

```
9 14
10 15
11 16
12 17
13 18
```

Because a 5-by-2 matrix requires ten elements, MATLAB software reads one message into `outdata` to fill the matrix.

- 17** To check your progress, see how many messages remain in the queue. You have read eight messages from the queue so 12 should remain.

```
num_of_msgs = IDE_Obj.rtdx.msgcount('ochan')

num_of_msgs =
    12
```

- 18** To see the connection between messages and a matrix in MATLAB software, read data from 'ochan' to fill a 4-by-5 matrix in your workspace.

```
outdata = IDE_Obj.rtdx.readmat('ochan','int16',[4 5])

outdata =
    10    14    18    13    17
    11    15    19    14    18
    12    16    11    15    19
    13    17    12    16    20
```

Filling the matrix required two messages worth of data.

- 19** To verify that the last step used two messages, recheck the message count. You should find 10 messages waiting in the queue.

```
num_of_msgs = IDE_Obj.rtdx.msgcount('ochan')
```

- 20** Continuing with matrix reads, fill a 10-by-5 matrix (50 matrix elements or five messages).

```
outdata = IDE_Obj.rtdx.readmat('ochan','int16',[10 5])

outdata =
    12    13    14    15    16
    13    14    15    16    17
    14    15    16    17    18
    15    16    14    18    19
    16    17    18    19    20
    17    18    19    20    21
    18    19    20    21    22
    19    20    21    22    23
    20    21    22    23    24
    21    22    23    24    25
```

- 21** Recheck the number of messages in the queue to see that five remain.

- 22** “flush” lets you remove messages from the queue without reading them. Data in the message you remove is lost. Use flush to remove the next message in the read queue. Then check the waiting message count.

```
IDE_Obj.rtdx.flush('ochan',1)
num_of_msgs = IDE_Obj.rtdx.msgcount('ochan')

num_of_msgs =
    4
```

- 23** Empty the remaining messages from the queue and verify that the queue is empty.

```
IDE_Obj.rtdx.flush('ochan', 'all')
```

With the `all` option, `flush` discards the messages in the `ochan` queue.

Closing the Connections and Channels or Cleaning Up

One of the most important programmatic processes you should do in every RTDX session is to clean up at the end. Cleaning up includes stopping your processor, disabling the RTDX channels you enabled, disabling RTDX and closing your open channels. Performing this series of tasks prevents trouble caused by unexpected interactions with remaining handles, channels, and links from earlier development work.

Best practices suggest that you include the following tasks (or an subset that meets your development needs) in your development scripts and programs.

We use several functions in this section; each has a purpose — the operational details in the following list explain how and why we use each one. They are

- **close** — close the specified RTDX channel. To use the channel again, you must open and enable the channel. Compare `close` to `disable`. `close('rtdx')` closes the communications provided by RTDX. After you close RTDX, you cannot communicate with your processor.
- **disable** — remove RTDX communications from the specified channel, but does not remove the channel, or link. Disabling channels may be useful when you do not want to see the data that is being fed to the channel, but you may want to read the channel later. By enabling the channel later, you have access to the data entering the channel buffer. Note that data that entered the channel while it was disabled is lost.
- **halt** — stop a running processor. You may still have one or more messages in the host buffer.

Use the following procedure to shut down communications between MATLAB software and the processor, and end your session:

- 1 Begin the process of shutting down the processor and RTDX by stopping the processor. Type the following functions at the prompt.

```
if (isrunning(IDE_Obj)) % Use this test in scripts.  
    IDE_Obj.halt;      % Halt the processor.  
end                   % Done.
```

Your processor may already be stopped at this point. In a script, you might put the function in an `if`-statement as we have done here. When you direct a stopped processor to halt, the function returns immediately.

- 2 You have stopped the processor. Now disable the RTDX channels you opened to communicate with the processor.

```
IDE_Obj.rtdx.disable('all');
```

If required, using `disable` with channel name and processor identifier input arguments lets you disable only the channel you choose. When you have more than one board or processor, you may find disabling selected channels meets your needs.

When you finish your RTDX communications session, disable RTDX so that Embedded Coder releases your open channels before you close them.

```
IDE_Obj.rtdx.disable;
```

- 3 Use the following function syntaxes to close your open channels. Either close selected channels by using the channel name in the function, or use the `all` option to close the open channels.

- `IDE_Obj.rtdx.close('ichan')` to close your input channel in this tutorial.
- `IDE_Obj.rtdx.close('ochan')` to close your output channel in the tutorial.
- `IDE_Obj.rtdx.close('all')` to close your open RTDX channels, regardless of whether they are part of this tutorial.

Consider using the `all` option with the `close` function when you finish your RTDX work. Closing channels reduces unforeseen problems caused by channel objects that exist but do not get closed when you end your session.

- 4 When you created your RTDX object (`IDE_Obj = ticcs('boardnum',1)`) at the beginning of this tutorial, the `ticcs` function opened CCS IDE and set the visibility to 0. To avoid problems that occur when you close the interface to RTDX with

CCS visibility set to 0, make CCS IDE visible on your desktop. The following `if` statement checks the CCS IDE visibility and changes it if required.

```
if IDE_Obj.isvisible,
    IDE_Obj.visible(1);
end
```

Visibility can cause problems. When CCS IDE is running invisibly on your desktop, do not use `clear all` to remove your links for CCS IDE and RTDX. Without a `ticcs` object that references the CCS IDE you cannot access CCS IDE to change the visibility setting, or close the application. To close CCS IDE when you do not have an existing object, either create a new object to access the CCS IDE, or use Microsoft Windows Task Manager to end the process `cc_app.exe`, or close the MATLAB software.

- 5 You have finished the work in this tutorial. Enter the following commands to close your remaining references to CCS IDE and release the associated resources.

```
clear ('all'); % Calls the link destructors to remove all links.
echo off
```

`clear all` without the parentheses removes the variables from your MATLAB workspace.

You have completed the tutorial using RTDX. During the tutorial you

- 1 Opened connections to CCS IDE and RTDX and used those connections to load an executable program to your processor.
- 2 Configured a pair of channels so you could transfer data to and from your processor.
- 3 Ran the executable on the processor, sending data to the processor for processing and retrieving the results.
- 4 Stopped the executing program and closed the links to CCS IDE and RTDX.

This tutorial provides a working process for using Embedded Coder and your signal processing programs to develop programs for a range of Texas Instruments processors. While the processor may change, the essentials of the process remain the same.

Listing Functions

To review a complete list of functions and methods that operate with `ticcs` objects, either CCS IDE or RTDX, enter either of the following commands at the prompt.

```
help ticcs
help rtdx
```

If you already have a `ticcs` object `IDE_Obj`, you can use dot notation to return the methods for CCS IDE or RTDX by entering one of the following commands at the prompt:

- `IDE_Obj.methods`
- `IDE_Obj.rtdx.methods`

In either instance MATLAB software returns a list of the available functions for the specified link type, including both public and private functions. For example, to see the functions (methods) for links to CCS IDE, enter:

```
help ticcs
```

Constructing ticcs Objects

When you create a connection to CCS IDE using the “`ticcs`” command, you are creating a “`ticcs` object for accessing the CCS IDE and RTDX interface”. The `ticcs` object implementation relies on MATLAB software object-oriented programming capabilities.

The discussions in this section apply to the `ticcs` objects in Embedded Coder.

Like other MATLAB software structures, objects in Embedded Coder have predefined fields called object properties.

You specify object property values by one of the following methods:

- Setting the property values when you create the `ticcs` object
- Creating an object with default property values, and changing these property values later

For examples of setting `ticcs` object properties, refer to `ticcs`.

Constructor for ticcs Objects

The easiest way to create an object is to use the function `ticcs` to create an object with the default properties. Create an object named `IDE_Obj` to refer to CCS IDE by entering

```
IDE_Obj = ticcs
```

MATLAB software responds with a list of the properties of the object `IDE_Obj` you created along with the associated default property values.

```
ticcs object:
```

```

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

RTDX channels    : 0

```

Inspecting the output reveals two objects listed—a CCS IDE object and an RTDX object. CCS IDE and RTDX objects cannot be created separately. By design they maintain a member class relationship; the RTDX object is a class, a member of the CCS object class. In this example, `IDE_Obj` is an instance of the class `CCS`. If you enter

```
rx = IDE_Obj.rtdx
```

`rx` is a handle to the RTDX portion of the CCS object. As an alias, `rx` replaces `IDE_Obj.rtdx` in functions such as `readmat` or `writemsg` that use the RTDX communications features of the CCS link. Typing `rx` at the command line now produces

```
rx
```

```
RTDX channels    : 0
```

The object properties are described in the function reference, and in more detail in `tics Object Properties`. These properties are set to default values when you construct objects.

tics Properties and Property Values

Objects in Embedded Coder software have properties associated with them. Each property is assigned a value. You can set the values of most properties, either when you create the link or by changing the property value later. However, some properties have read-only values. And a few property values, such as the board number and the processor to which the link attaches, become read-only after you create the object. You cannot change those after you create your link.

Overloaded Functions for tics Objects

Several functions in this Embedded Coder have the same name as functions in other MathWorks toolboxes or in MATLAB software. These behave similarly to their original counterparts, but you apply these functions directly to an object. This concept of having

functions with the same name operate on different types of objects (or on data) is called *overloading* of functions.

For example, the `set` command is overloaded for `ticcs` objects. After you specify your link by assigning values to its properties, you can apply the functions in this toolbox (such as “readmat” for using RTDX to read an array of data from the processor) directly to the variable name you assign to your object, without specifying your object parameters again.

ticcs Object Properties

- “Quick Reference to `ticcs` Object Properties” on page 39-44
- “Details About `ticcs` Object Properties” on page 39-45

Embedded Coder provides an interface to your processor hardware so you can communicate with processors for which you are developing systems and algorithms. Each `ticcs` object comprises two objects—a CCS IDE object and an RTDX interface object. The objects are not separable; the RTDX object is a subclass of the CCS IDE object. Each of the objects has multiple properties. To configure the interface objects for CCS IDE and RTDX, you set parameters that define details such as the desired board, the processor, the timeout period applied for communications operations, and a number of other values. Because Embedded Coder uses objects to create the interface, the parameters you set are called properties and you treat them as properties when you set them, retrieve them, or modify them.

This section details the properties for the `ticcs` objects for CCS IDE and RTDX. First the section provides tables of the properties, for quick reference. Following the tables, the section offers in-depth descriptions of each property, its name and use, and whether you can set and get the property value associated with the property. Descriptions include a few examples of the property in use.

MATLAB software users may find much of this handling of objects familiar. Objects in Embedded Coder, behave like objects in MATLAB software and the other object-oriented toolboxes. For C++ programmers, discussion of object-oriented programming is likely to be a review.

Quick Reference to `ticcs` Object Properties

The following table lists the properties for the `ticcs` objects in Embedded Coder. The second column tells you which object the property belongs to. Knowing which property belongs to each object in a `ticcs` object tells you how to access the property.

Property Name	Applies to Which Connection?	User Settable?	Description
apiversion	CCS IDE	No	Reports the version number of your CCS API.
boardnum	CCS IDE	Yes/initially	Specifies the index number of a board that CCS IDE recognizes.
ccsappexe	CCS IDE	No	Specifies the path to the CCS IDE executable. Read-only property.
numchannels	RTDX	No	Contains the number of open RTDX channels for a specific link.
page	CCS IDE	Yes/default	Stores the default memory page for reads and writes.
procnum	CCS IDE	Yes/at start only	Stores the number CCS Setup Utility assigns to the processor.
timeout	CCS IDE	Yes/default	Contains the global timeout setting for the link.
version	RTDX	No	Reports the version of your RTDX software.

Some properties are read only — you cannot set the property value. Other properties, you can change. If the entry in the User Settable column is “Yes/initially”, you can set the property value only when you create the link. Thereafter it is read only.

Details About ticcs Object Properties

To use the links for CCS IDE and RTDX interface you set values for:

- `boardnum` — specify the board with which the link communicates.
- `procnum` — specify the processor on the board. If the board has multiple processors, `procnum` identifies the processor to use.
- `timeout` — specify the global timeout value. (Optional. Default is 10 s.)

Details of the properties associated with connections to CCS IDE and RTDX interface appear in the following sections, listed in alphabetical order by property name.

Many of these properties are object linking and embedding (OLE) handles. The MATLAB software COM server creates the handles when you create objects for CCS IDE and RTDX. You can manipulate the OLE handles using `get`, `set`, and `invoke` to work directly with the COM interface with which the handles interact.

apiversion

Property `appversion` contains a string that reports the version of the application program interface (API) for CCS IDE that you are using when you create a link. You cannot change this string. When you upgrade the API, or CCS IDE, the string changes to match. Use `display` to see the `apiversion` property value for a link. This example shows the `appversion` value for link `IDE_Obj`.

```
display(IDE_Obj)
```

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

  RTDX channels    : 0
```

Note that the API version is not the same as the CCS IDE version.

boardnum

Property `boardnum` identifies the board referenced by the IDE link handle object for CCS. When you create a link, you use `boardnum` to specify the board you are using. To get the value for `boardnum`, use “`ccsboardinfo`” or the CCS Setup utility from Texas Instruments software. The CCS Setup utility assigns the number for each board installed on your system.

ccsappexe

Property `ccsappexe` contains the path to the CCS IDE executable file `cc_app.exe`. When you use `ticcs` to create a link, MATLAB software determines the path to the CCS IDE executable and stores the path in this property. This is a read-only property. You cannot set it.

numchannels

Property `numchannels` reports the number of open RTDX communications channels for an RTDX link. Each time you open a channel for a link, `numchannels` increments by one. For new links `numchannels` is zero until you open a channel for the link.

To see the value for `numchannels` create a link to CCS IDE. Then open a channel to RTDX. Use `display` to see the RTDX link properties.

```
IDE_Obj=ticcs

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

  RTDX channels    : 0

rx=IDE_Obj.rtdx

  RTDX channels    : 0

open(rx,'ichan','r','ochan','w');

get(IDE_Obj.rtdx)

ans =

    numChannels: 2
           Rtdx: [1x1 COM ]
    RtdxChannel: {' ' ' ' '}
           procType: 103
           timeout: 10
```

page

Property `page` contains the default value CCS IDE uses when the user does not specify the page input argument in the syntax for a function that access memory.

procnum

Property `procnum` identifies the processor referenced by the IDE link handle object for CCS. When you create an object, you use `procnum` to specify the processor you are using .

The CCS Setup Utility assigns a number to each processor installed on each board. To determine the value of `procnum` for a processor, use “ccsboardinfo” or the CCS Setup utility from Texas Instruments software.

To identify a processor, you need both the `boardnum` and `procnum` values. For boards with one processor, `procnum` equals zero. CCS IDE numbers the processors on multiprocessor boards sequentially from 0 to the number of processors. For example, on a board with four processors, the processors are numbered 0, 1, 2, and 3.

rtdx

Property `rtdx` is a subclass of the `ticcs` link and represents the RTDX portion of the IDE link handle object for CCS. As shown in the example, `rtdx` has properties of its own that you can set, such as `timeout`, and that report various states of the link.

```
get(IDE_Obj.rtdx)

ans =

    version: 1
 numChannels: 0
      Rtdx: [1x1 COM ]
 RtdxChannel: {'' [] ''}
   procType: 103
   timeout: 10
```

In addition, you can create an alias to the `rtdx` portion of a link, as shown in this code example.

```
rx=IDE_Obj.rtdx

    RTDX channels    : 0
```

Now you can use `rx` with the functions in Embedded Coder, such as `get` or `set`. If you have two open channels, the display looks like the following example:

```
get(rx)

ans =

    numChannels: 2
      Rtdx: [1x1 COM ]
 RtdxChannel: {2x3 cell}
   procType: 98
```



```
timeout: 10
```

rtdxchannel

Property `rtdxchannel`, along with `numchannels` and `proctype`, is a read-only property for the RTDX portion of the IDE link handle object for CCS. To see the value of this property, use `get` with the link. Neither `set` nor `invoke` work with `rtdxchannel`.

`rtdxchannel` is a cell array that contains the channel name, handle, and mode for each open channel for the link. For each open channel, `rtdxchannel` contains three fields, as follows:

<code>.rtdxchannel{i,1}</code>	Channel name of the <i>i</i> th-channel, <i>i</i> from 1 to the number of open channels
<code>.rtdxchannel{i,2}</code>	Handle for the <i>i</i> th-channel
<code>.rtdxchannel{i,3}</code>	Mode of the <i>i</i> th-channel, either 'r' for read or 'w' for write

With four open channels, `rtdxchannel` contains four channel elements and three fields for each channel element.

timeout

Property `timeout` specifies how long CCS IDE waits for a process to finish. Two `timeout` periods can exist — one global, one local. You set the global `timeout` when you create the IDE link handle object for CCS. The default global `timeout` value is 10 s. However, when you use functions to read or write data to CCS IDE or your processor, you can set a local `timeout` that overrides the global value. If you do not set a specific `timeout` value in a read or write process syntax, the global `timeout` value applies to the operation. Refer to the help for the read and write functions for the syntax to set the local `timeout` value for an operation.

version

Property `version` reports the version number of your RTDX software. When you create a `ticcs` object, `version` contains a string that reports the version of the RTDX application that you are using. You cannot change this string. When you upgrade the API, or CCS IDE, the string changes to match. Use `display` to see the `version` property value for a link. This example shows the `apiversion` value for object `rx`.

```
get(rx) % rx is an alias for IDE_Obj.rtdx.
```

```
ans =  
  
    version: 1  
    numChannels: 0  
        Rtdx: [1x1 COM ]  
    RtdxChannel: {'' [] ''}  
    procType: 103  
    timeout: 10
```

Function List

The following is a complete list of functions for working with CCS v.3.3. It includes Automation Interface and other types of functions.

- activate
- add
- address
- animate
- build
- ccsboardinfo
- cd
- checkEnvSetup
- close
- configure
- dir
- disable
- display (IDE Object)
- enable
- flush
- getbuildopt
- halt
- info
- insert
- xmakefilesetup

- isenabled
- isreadable
- isrtdxcapable
- isrunning
- isvisible
- iswritable
- list
- load
- msgcount
- new
- open
- profile
- read
- readmat
- readmsg
- regread
- regwrite
- reload
- remove
- reset
- restart
- run
- save
- setbuildopt
- symbol
- ticcs
- visible
- write
- writemsg

IDE Project Generator

In this section...

“Introducing IDE Project Generator” on page 39-52

“IDE Project Generator and Board Selection” on page 39-52

“Generate an IDE Project” on page 39-54

“Model Reference” on page 39-57

Introducing IDE Project Generator

IDE Project Generator provides the following features for developing project and generating code:

- Support automated project building for Texas Instruments Code Composer Studio software that lets you create projects from code generated by Embedded Coder products. The project automatically populates CCS projects in the CCS development environment.
- Configure code generation using model Configuration Parameters and processor preferences block options
- Select from two system target files to generate code specific to your processor
- Configure project build process
- Automatically download and run your generated projects on your processor

Note: You cannot generate code for C6000 processors in big-endian mode. Code generation supports only little-endian processor data byte order.

IDE Project Generator and Board Selection

IDE Project Generator uses `ticcs` objects to connect to the IDE. Each time you build a model to generate a project, the build process starts by issuing the “`ticcs`” method, as shown here:

```
IDE_Obj=ticcs('boardnum',boardnum,'procnum',procnum)
```

The software attempts to connect to the board (`boardnum`) and processor (`procnum`) associated with the **Board name** and **Processor number** parameters located on the Target Hardware Resources pane in the model Configuration Parameters.

The result of the `ticcs` method changes, depending on the boards you configured in CCS. The following table describes how the software selects the board to connect to in your board configuration.

CCS Board Configuration State	Response by Software
Code Composer Studio or Embedded Coder software not installed.	Returns an error message asking you to verify that you installed both Code Composer Studio and Embedded Coder.
Code Composer Studio software does not have configured boards.	Returns an error message that the software could not find boards in your configuration. Use Setup Code Composer Studio to configure at least one board.
Code Composer Studio software has one configured board.	Attaches to the board regardless of the value of the Board parameter. You see a warning message telling you which board the software selected.
Code Composer Studio software has one board configured that does not match the value of the Board parameter. ^(*)	Returns a warning message that the software could not find the board specified in the block and connected to the board listed in the warning message. The software connects to the first board in your CCS configuration.
Code Composer Studio has more than one board configured. The value of the Board parameter is one of the configured boards.	Connects to the specified board.
Code Composer Studio has more than one board configured. The value of the Board parameter is not one of the configured boards. ^(*)	Returns a message asking you to select a board from the list of configured boards. You have two choices: <ul style="list-style-type: none"> • Select a board to use for project generation, and click OK. Your selection does not change the value of the Board parameter. The software connects to the selected board. • Click Abort to stop the project build and code generation process. The software does not connect to the IDE or board.

(*)You may encounter the situation where you do not have the valid board configured in CCS because of one of the following conditions:

- You changed your board configuration and saved the model. When you reopen the model, the board specified in **Board name** in the block is not in your configuration.
- You are working with a model from a source whose board configuration is not the same as yours.

Use “ccsboardinfo” at the MATLAB prompt to verify or review your configured boards.

Generate an IDE Project

- “Creating the Model” on page 39-55
- “Specify Configuration Parameters for Your Model” on page 39-55

In this tutorial you will use the Embedded Coder software to:

- Build a model.
- Generate a project from the model.
- Build the project and run the binary on a processor.

Note The model shows project generation. You cannot not build and run the model on your processor without additional blocks.

To generate a project from a model, complete the following tasks:

- 1 Create a model application.
- 2 Configure your model for your IDE, tool chain, and target hardware, as described in “Configure Target Hardware Resources” on page 36-3.
- 3 In the Configuration Parameters, also set:
 - Solver parameters such as simulation start and solver options
 - Software options such as processor configuration and processor compiler selection
- 4 Generate your project.
- 5 Review your project in CCS.

Creating the Model

To build a model, follow these steps:

- 1 Open the Simulink Library Browser.
- 2 Use Simulink blocks to create a model, or open one of the example models for Texas Instruments Code Composer Studio.
- 3 Name and save your model before continuing.

Specify Configuration Parameters for Your Model

The following sections describe how to configure the build and run parameters for your model. Generating a project, or building and running a model on the processor, starts with configuring model options in the Configuration Parameters dialog in Simulink software.

Setting Solver Parameters

After you have designed and implemented your digital signal processing model in Simulink software, complete the following steps to set the Configuration Parameters for the model:

- 1 Configure your model for your IDE, tool chain, and target hardware, as described in “Configure Target Hardware Resources” on page 36-3.
- 2 Select the Solver pane in the Configuration Parameters dialog.
- 3 Set **Start time** to **0.0** and **Stop time** to **inf** (model runs without stopping). If you set a stop time, your generated code does not honor the setting. Set this to **inf** for completeness.
- 4 Under **Solver options**, set **Type** to **fixed-step** and set **Solver** to **discrete (no continuous states)**. For PIL, set **Type** and **Solver** to any setting.
- 5 For **Fixed step size (fundamental sample time)**, enter **Auto**, and set **Tasking mode for periodic sample times** to **SingleTasking**.

Note Generated code does not honor Simulink software stop time from the simulation. Stop time is interpreted as **inf**. To implement a stop in generated code, add a Stop Simulation block in your model.

When you use PIL, you can set the **Solver options** to any selection from the **Type** and **Solver** lists.

Ignore the **Data Import/Export**, **Diagnostics**, and **Optimization** panes in the Configuration Parameters dialog. The default settings are valid for your new model.

Setting Code Generation Parameters

To configure your software to use the right processor files and to compile and run your model executable file, configure the **Code Generation** pane in the Configuration Parameters dialog.

- 1 In the Configuration Parameters dialog, select the **Code Generation** pane.
- 2 Use the **Browse** button to set **System target file** to `idelink_grt.tlc`.

Setting Coder Target Parameters

To configure code generation options and to compile and run your model executable file, configure the Coder Target pane in the Configuration Parameters dialog.

- 1 In the Configuration Parameters dialog, expand the node for the **Code Generation** pane and select the **Coder Target** pane.
- 2 Set the following options in the pane under **Vendor Tool Chain**:
 - **Configuration** should be Custom.
 - Set **Compiler options string** and **Linker options string** should be blank.
- 3 Under **Link Automation**, verify that **Export IDE link handle to base workspace** is selected and provide a name for the handle in **Coder Target handle name** (optional).
- 4 Set the following **Run-Time** options:
 - **Build action**: `Build_and_execute`.
 - **Interrupt overrun notification method**: None.

You have configured the your software options that let you generate a project for you processor. You may have noticed that you did not configure a few of the Configuration Parameters panes, such as **Comments**, **Symbols**, and **Optimization**.

For your new model, the default values for the options in these panes are right. For other models you develop, you may want to set the options in these panes to provide information during the build and to run TLC debugging when you generate code. Refer to your product documentation for more information about setting the Configuration Parameters.

Building Your Project

After you set the Configuration Parameters and configure the coder product to create the files you need, you direct the build process to create your project:

- 1 Press **OK** to close the Configuration Parameters dialog.
- 2 Click **Ctrl+B** to generate your project into CCS IDE.

When you click **Build** with **Create_project** selected for **Build action**, the automatic build process starts CCS IDE, populates a new project in the development environment, builds the project, loads the binary on the processor, and runs it.

- 3 To stop processor execution, use the **Halt** option in CCS or enter `IDE_Obj.halt` at the MATLAB command prompt. (Where “IDE_Obj” is the IDE link handle name you specified previously in **Configuration Parameters**.)

Model Reference

Model reference lets your model include other models as modular components. This technique provides useful features because it:

- Simplifies working with large models by letting you build large models from smaller ones, or even large ones.
- Lets you generate code for the modules in the model, and later, regenerate code for modules that change.
- Lets you develop the modules independently.
- Lets you reuse modules and models by reference, rather than including the model or module multiple times in your model. Also, multiple models can refer to the same model or module.

Your product documentation provides much more information about model reference.

How Model Reference Works

Model reference behaves differently in simulation and in code generation. For this discussion, you need to know the following terms:

- Top-model — The root model block or model. It refers to other blocks or models. In the model hierarchy, this is the topmost model.
- Referenced models — Blocks or models that other models reference, such as models the top-model refers to. The models or blocks below the top-model in the hierarchy are reference models.

The following sections describe briefly how model reference works. More details are available in your product documentation in the online Help system.

Model Reference in Simulation

When you simulate the top-model, the coder product detects that your model contains referenced models. Simulink software generates code for the referenced models and uses the generated code to build shared library files for updating the model diagram and simulation. It also creates an executable (a MEX file, `.mex`) for each reference model that is used to simulate the top-model.

When you rebuild reference models for simulations or when you run or update a simulation, Simulink software rebuilds the model reference files. Whether reference files or models are rebuilt depends on:

- Whether and how you change the models.
- The **Rebuild** parameter on the **Model Reference** pane in the Configuration Parameters dialog.

Model Reference in Code Generation

Embedded Coder software requires executables to generate code from models. If you have not simulated your model at least once, the coder product creates a `.mex` file for simulation.

Next, for each referenced model, the code generation process calls `make_rtw` and builds each referenced model. This build process creates a library file for each of the referenced models in your model.

After building the referenced models, the coder product calls `make_rtw` on the top-model, linking to the library files it created for the associated referenced models.

Using Model Reference

With few limitations or restrictions, Embedded Coder provides full support for generating code from models that use model reference.

Build Action Setting

The most important requirement for using model reference with the TI's processors is to set the **Build action** for the Model blocks in the simulation to `Archive_library`.

To set the build action

- 1 Open your model.
- 2 Select **Simulation > Model Configuration Parameters** from the model menus.
The Configuration Parameters dialog opens.
- 3 Expand the node for the Code Generation pane. Then select the **Coder Target** pane.
- 4 In the right pane, under **Run-Time**, select **Archive_library** from the **Build action** list.

If your top-model uses a reference model that does not have the build action set to **Archive_library**, the build process automatically changes the build action to **Archive_library** and issues a warning about the change.

As a result of selecting the **Archive_library** setting, other options are disabled:

- DSP/BIOS is disabled for the referenced models. Only the top-model supports DSP/BIOS operation.
- **Interrupt overrun notification method**, **Export IDE link handle to base workspace**, and **System stack size** are disabled for the referenced models.

Other Block Limitations

Model reference with Embedded Coder does not allow you to use the following blocks or S-functions in reference models:

- Blocks from the C62x DSP Library (in **c60001ib**) (because these are noninlined S-functions)
- Blocks from the C64x DSP Library (in **c60001ib**) (because these are noninlined S-functions)
- Noninlined S-functions
- Driver blocks, such as the ADC or DAC blocks from a Embedded Coder block library

Configuring processors to Use Model Reference

Processors that you plan to use in Model Referencing must meet some general requirements.

- The **System target file** on the **Code Generation** pane of the **Configuration Parameters** dialog must match the target hardware.
- When you generate code from a model that references another model, you need to configure both the top-level model and the referenced models for the same code generation processor.

- The External mode option is not supported in model reference processor builds. Embedded Coder product does not support External mode. If you select this option, it is ignored during code generation.
- To support model reference builds, your TMF must support use of the shared utilities folder, as described in “Supporting Shared Utility Directories in the Build Process” in the Simulink Coder documentation.

To use an existing processor, or a new processor, with Model Reference, you set the `ModelReferenceCompliant` flag for the processor. For information on how to set this option, refer to `ModelReferenceCompliant` in the online Help system.

If you start with a model that was created prior to version 2.4 (R14SP3), to make your model compatible with the model reference processor, use the following command to set the `ModelReferenceCompliant` flag to On:

```
set_param(bdroot, 'ModelReferenceCompliant', 'on')
```

Models that you develop with versions 2.4 and later of Embedded Coder automatically include the model reference capability. You do not need to set the flag.

Exporting Filter Coefficients from FDATool

In this section...

“About FDATool” on page 39-61

“Preparing to Export Filter Coefficients to Code Composer Studio Projects” on page 39-62

“Exporting Filter Coefficients to Your Code Composer Studio Project” on page 39-65

“Preventing Memory Corruption When You Export Coefficients to Processor Memory” on page 39-70

About FDATool

Signal Processing Toolbox™ software provides the Filter Design and Analysis tool (FDATool) that lets you design a filter and then export the filter coefficients to a matching filter implemented in a CCS project.

Using FDATool with CCS IDE enables you to:

- Design your filter in FDATool
- Use CCS to test your filter on a processor
- Redesign and optimize the filter in FDATool
- Test your redesigned filter on the processor

For instructions on using FDATool, refer to the section “Filter Design and Analysis Tool” in the Signal Processing Toolbox documentation.

Procedures in this chapter show how to use the FDATool export options to export filter coefficients to CCS. Using these procedures, you can perform the following tasks:

- Export filter coefficients from FDATool in a header file—“Exporting Filter Coefficients from FDATool to the CCS IDE Editor” on page 39-66
- Export filter coefficients from FDATool to processor memory—“Replacing Existing Coefficients in Memory with Updated Coefficients” on page 39-71

Caution For the best results, export coefficients in a header file. Exporting coefficients directly to processor memory can generate unexpected results or corrupt memory.

Also see the reference pages for the following functions. These primary functions allow you use to access variables and write them to processor memory from the MATLAB Command window.

- **address** — Return the address of a symbol so you can read or write to it.
- **ticcs** — Create a connection between MATLAB software and CCS IDE so you can work with the project in CCS from the MATLAB Command window.
- **write** — Write data to memory on the processor.

Preparing to Export Filter Coefficients to Code Composer Studio Projects

- “Features of a Filter” on page 39-62
- “Selecting the Export Mode” on page 39-63
- “Choosing the Export Data Type” on page 39-63

Features of a Filter

When you create a filter in FDATool, the filter includes defining features identified in the following table.

Defining Feature	Description
Structure	Structure defines how the elements of a digital filter—gains, adders/subtractors, and delays—combine to form the filter. See the Signal Processing Toolbox documentation in the Online Help system for more information about filter structures.
Design Method	Defines the mathematical algorithm used to determine the filter response, length, and coefficients.
Response Type and Specifications	Defines the filter passband shape, such as lowpass or bandpass, and the specifications for the passband.
Coefficients	Defines how the filter structure responds at each stage of the filter process.
Data Type	Defines how to represent the filter coefficients and the resulting filtered output. Using a floating-point or fixed-point coefficient alters the filter response and output data values.

When you export your filter, FDATool exports only the number of and value of the filter coefficients and the data type used to define the coefficients.

Selecting the Export Mode

You can export a filter by generating an ANSI C header file, or by writing the filter coefficients directly to processor memory. The following table summarizes when and how to use the export modes.

To...	Use Export Mode...	When to Use	Suggested Use
Add filter coefficients to a project in CCS	C header file	You implemented a filter algorithm in your program, but you did not allocate memory on your processor for the filter coefficients.	<ul style="list-style-type: none"> Add the generated ANSI C header file to a project. Building and loading this project into your processor allocates static memory locations on the processor and writes your filter coefficients to those locations. Edit the file so the header file allocates extra processor memory and then add the header file to your project. Refer to “Allocating Extra Memory for Filter Coefficients” on page 39-70 in the next section. <p>(For a sample generated header file, refer to “Reviewing ANSI C Header File Contents” on page 39-69.)</p>
Modify the filter coefficients in an embedded application loaded on a processor	Write directly to memory	You loaded a program on your processor. The program allocated space in your processor memory to store the filter coefficients.	<ul style="list-style-type: none"> Optimize your filter design in FDATool. Then, Write the updated filter coefficients directly to the allocated processor memory. Refer to section “Preventing Memory Corruption When You Export Coefficients to Processor Memory” on page 39-70 for more information.

Choosing the Export Data Type

The export process provides two ways you can specify the data type to use to represent the filter coefficients. Select one of the options shown in the following table when you export your filter.

Specify Data Type for Export	Description
Export suggested	Uses the data type that FDATool suggests to preserve the fidelity of the filter coefficients and the performance of your filter in the project
Export as	Lets you specify the data type to use to export the filter coefficients

FDATool exports filter coefficients that use the following data types directly without modifications:

- Signed integer (8, 16, or 32 bits)
- Unsigned integer (8, 16, or 32 bits)
- Double-precision floating point (64 bits)
- Single-precision floating point (32 bits)

Filters in FDATool in the Signal Processing Toolbox software use double-precision floating point. You cannot change the data type.

If you have installed DSP System Toolbox software, you can use the filter quantization options in FDATool to set the word and fraction lengths that represent your filter coefficients. For information about using the quantization options, refer to Filter Design and Analysis Tool in the Filter Design Toolbox documentation in the Online help system.

If your filter uses one of the supported data types, **Export suggested** specifies that data type.

If your filter does not use one of the supported data types, FDATool converts the unsupported data type to one of the supported types and then suggests that data type. For more information about how FDATool determines the data type to suggest, refer to “How FDATool Determines the Export Suggested Data Type” on page 39-65.

Follow these best-practice guidelines when you implement your filter algorithm in source code and design your filter in FDATool:

- Implement your filter using one of the data types FDATool exports without modifications.
- Design your filter in FDATool using the data type you used to implement your filter.

To Choose the Export Data Type

When you export your filter, follow this procedure to select the export data type so that the exported filter coefficients closely match the coefficients of your filter in FDATool.

- 1 In FDATool, select **Targets > Code Composer Studio IDE** to open the Export to Code Composer Studio IDE dialog.
- 2 Perform one of the following actions:
 - Select **Export suggested** to export the coefficients in the suggested data type.
 - Select **Export as** and choose the data type your filter requires from the list.

Caution If you select **Export as**, the exported filter coefficients can be very different from the filter coefficients in FDATool. As a result, your filter cutoff frequencies and performance may not match your design in FDATool.

How FDATool Determines the Export Suggested Data Type

By default, FDATool represents filter coefficients as double-precision floating-point data. When you export your filter coefficients, FDATool suggests the same data type.

If you set custom word and fraction lengths to represent your filter coefficients, the export process suggests a data type to maintain the best fidelity for the filter.

The export process converts your custom word and fraction lengths to a suggested export data type, using the following rules:

- Round the word length up to the nearest larger supported data type. For example, round an 18-bit word length up to 32 bits.
- Set the fraction length to the maintain the same difference between the word and fraction length in the new data type as applies in the custom data type.

For example, if you specify a fixed-point data type with word length of 14 bits and fraction length of 11 bits, the export process suggests an integer data type with word length of 16 bits and fraction length of 13 bits, retaining the 3 bit difference.

Exporting Filter Coefficients to Your Code Composer Studio Project

- “Exporting Filter Coefficients from FDATool to the CCS IDE Editor” on page 39-66

- “Reviewing ANSI C Header File Contents” on page 39-69

Exporting Filter Coefficients from FDATool to the CCS IDE Editor

In this section, you export filter coefficients to a project by generating an ANSI C header file that contains the coefficients. The header file defines global arrays for the filter coefficients. When you compile and link the project to which you added the header file, the linker allocates the global arrays in static memory locations in processor memory.

Loading the executable file into your processor allocates enough memory to store the exported filter coefficients in processor memory and writes the coefficients to the allocated memory.

Use the following steps to export filter coefficients from FDATool to the CCS IDE text editor.

- 1** Start FDATool by entering `fdatool` at the MATLAB command prompt.

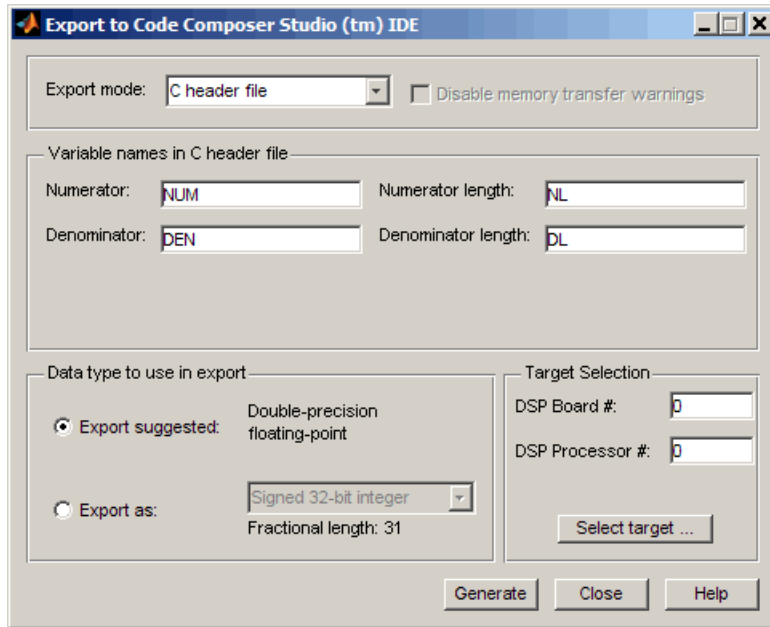
```
fdatool      % Starts FDATool.
```

- 2** Design a filter with the same structure, length, design method, specifications, and data type you implemented in your source code filter algorithm.

The following figure shows a Direct-form II IIR filter example that uses second-order sections.

- 3** Click **Store Filter** to store your filter design. Storing the filter allows you to recall the design to modify it.
- 4** To export the filter coefficients, select **Targets > Code Composer Studio IDE** from the FDATool menu bar.

The Export to Code Composer Studio IDE dialog opens, as shown in the following figure.



- 5 Set **Export mode** to C header file.



- 6 In **Variable names in C header file**, enter variable names for the **Numerator**, **Denominator**, **Numerator length**, and **Denominator length** parameters where the coefficients will be stored.

The dialog shows only the variables you need to export to define your filter.

Note: You cannot use reserved ANSI C programming keywords, such as `if` or `int` as variable names, or include invalid characters such as spaces or semicolons (`;`).

- 7 In **Data type to use in export**, select **Export suggested** to accept the recommended export data type. FDATool suggests a data type that retains filter coefficient fidelity.

You may find it useful to select the **Export as** option and select an export data type other than the one suggested.

Caution If you deviate from the suggested data type, the exported filter coefficients can be very different from the filter coefficients in FDATool. As a result, your filter cutoff frequencies and performance may not match your design in FDATool.

For more information about how FDATool decides which data type to suggest, refer to “How FDATool Determines the Export Suggested Data Type” on page 39-65.

- 8 If you know the board number and processor number of your target, enter **DSP Board #** and **DSP Processor #** values to identify your board.

When you have only one board or simulator, Embedded Coder software sets **DSP Board #** and **DSP Processor #** values for your board automatically.

If you have more than one board defined in CCS Setup:

- Click **Select target** to open the Selection Utility dialog.
 - From the list of boards and list of processors, select the board name and processor name to use.
 - Click **Done** to set the **DSP Board #** and **DSP Processor #** values.
- 9 Click **Generate** to generate the ANSI header file. FDATool prompts you for a file name and location to save the generated header file.

The default location to save the file is your MATLAB working folder. The default file name is `fdacoeffs.h`.

- 10 Click **OK** to export the header file to the CCS editor.

If CCS IDE is not open, this step starts the IDE.

The export process does not add the file to your active project in the IDE.

- 11 Drag your generated header file into the project that implements the filter.
- 12 Add a `#include` statement to your project source code to include the new header file when you build your project.

- 13** Generate a `.out` file and load the file into your processor. Loading the file allocates locations in static memory on the processor and writes the filter coefficients to those locations.

To see an example header file, refer to “Reviewing ANSI C Header File Contents” on page 39-69.

Reviewing ANSI C Header File Contents

The following program listing shows the exported header (`.h`) file that FDATool generates. This example shows a direct-form II filter that uses five second-order sections. The filter is stable and has linear phase.

Comments in the file describe the filter structure, number of sections, stability, and the phase of the filter. Source code shows the filter coefficients and variables associated with the filter design, such as the numerator length and the data type used to represent the coefficients.

```

/*
 * Filter Coefficients (C Source) generated by the Filter Design and Analysis Tool
 *
 * Generated by MATLAB(R) 7.8 and the Signal Processing Toolbox 6.11.
 *
 * Generated on: xx-xxx-xxxx 14:24:45
 */

/*
 * Discrete-Time IIR Filter (real)
 * -----
 * Filter Structure      : Direct-Form II, Second-Order Sections
 * Number of Sections   : 5
 * Stable                : Yes
 * Linear Phase         : No
 */

/* General type conversion for MATLAB generated C-code */
#include "tmwtypes.h"
/*
 * Expected path to tmwtypes.h
 * $MATLABROOT\extern\include\tmwtypes.h
 */
#define MWSPT_NSEC 11
const int NL[MWSPT_NSEC] = { 1,3,1,3,1,3,1,3,1,3,1 };
const real64_T NUM[MWSPT_NSEC][3] = {
    {
        0.802536131462,          0,          0
    },
    {
        0.2642710234701,    0.5285420469403,    0.2642710234701
    },
    {
        1,          0,          0
    },
    {

```

```

    0.1743690465012,    0.3487380930024,    0.1743690465012
},
#
{
    0.2436793028081,    0.4873586056161,    0.2436793028081
},
{
    1,                    0,                    0
},
{
    0.3768793219093,    0.7537586438185,    0.3768793219093
},
{
    1,                    0,                    0
}
};
const int DL[MWSPT_NSEC] = { 1,3,1,3,1,3,1,3,1,3,1 };
const real164_T DEN[MWSPT_NSEC][3] = {
    {
        1,                    0,                    0
    },
    {
        1,    -0.1842138030775,    0.1775781189277
    },
    {
        1,                    0,                    0
    }
},
# {
    {
        1,    -0.2160098642842,    0.3808329528195
    },
    {
        1,                    0,                    0
    }
}
};

```

Preventing Memory Corruption When You Export Coefficients to Processor Memory

- “Allocating Extra Memory for Filter Coefficients” on page 39-70
- “Using the Exported Header File to Allocate Extra Processor Memory” on page 39-71
- “Replacing Existing Coefficients in Memory with Updated Coefficients” on page 39-71
- “Changing Filter Coefficients Stored on Your Processor” on page 39-72

Allocating Extra Memory for Filter Coefficients

You can allocate extra memory by editing the generated ANSI C header file. You can then load the associated program file into your processor as described in “Using the Exported Header File to Allocate Extra Processor Memory” on page 39-71. Extra

memory lets you change filter coefficients and overwrite existing coefficients stored in processor memory more easily.

To prevent problems when you update filter coefficients in a project, , such as writing coefficients to unintended memory locations, use the **C header file** export mode option in FDATool to update filter coefficients in your program.

Using the Exported Header File to Allocate Extra Processor Memory

You can edit the generated header file so the linked program file allocates extra processor memory. By allocating extra memory, you avoid the problem of insufficient memory when you export new coefficients directly to allocated memory.

For example, changing the following command in the header file:

```
const real64_T NUM[47] = {...}
to
```

```
real64_T NUM[256] = {...}
```

allocates enough memory for NUM to store up to 256 numerator filter coefficients rather than 47.

Exporting the header file to CCS IDE does not add the filter to your project. To incorporate the filter coefficients from the header file, add a **#include** statement:

```
#include "headerfilename.h"
```

Refer to “Exporting Filter Coefficients to Your Code Composer Studio Project” on page 39-65 for information about generating a header file to export filter coefficients.

When you export filter coefficients directly to processor memory, the export process writes coefficients to as many memory locations as they need. The write process does not perform bounds checking. Plan memory allocation carefully, so that the software writes to the right locations and has enough memory for filter coefficients.

Replacing Existing Coefficients in Memory with Updated Coefficients

When you redesign a filter and export new coefficients to replace existing coefficients in memory, verify the following conditions for your new design:

- Your redesign did not increase the memory required to store the coefficients beyond the allocated memory.

Changes that increase the memory required to store the filter coefficients include the following redesigns:

- Increasing the filter order
- Changing the number of sections in the filter
- Changing the numerical precision (changing the export data type)
- Your changes did not change the export data type.

Caution Identify changes that require additional memory to store the coefficients before you begin your export. Otherwise, exporting the new filter coefficients may overwrite data in memory locations you did not allocate for storing coefficients. Also, exporting filter coefficients to memory after you change the filter order, structure, design algorithm, or data type can yield unexpected results and corrupt memory.

Changing the filter design algorithm in FDATool, such as changing from Butterworth to Maximally Flat, often changes the number of filter coefficients (the filter order), the number of sections, or both. Also, the coefficients from the new design algorithm may not work with your source code filter implementation.

If you change the design algorithm, verify that your filter structure and length are the same after you redesign your filter, and that the coefficients will work with the filter you implemented.

If you change the number of sections or the filter order, your filter will not work well unless your filter algorithm implementation accommodates the changes.

Changing Filter Coefficients Stored on Your Processor

This example writes filter coefficients to processor memory to replace the existing coefficients. To perform this process, you need the names of the variables in which your project stores the filter data.

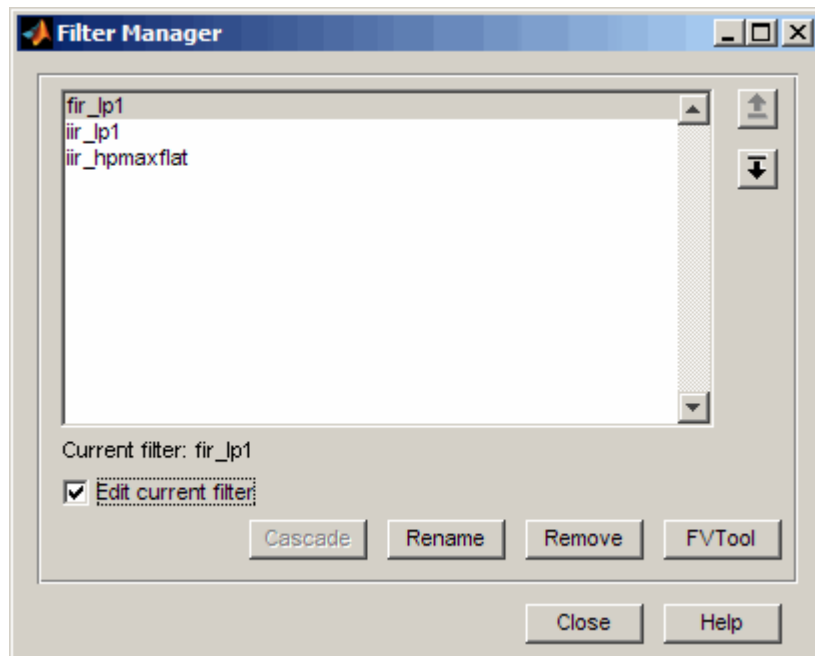
Before you export coefficients directly to memory, verify that your project allocated enough memory for the new filter coefficients. If your project allocated enough memory, you can modify your filter in FDATool and then follow the steps in this example to export the updated filter coefficients to the allocated memory.

If your new filter requires additional memory space, use a C header file to allocate memory on the processor and export the new coefficients as described in “Exporting Filter Coefficients to Your Code Composer Studio Project” on page 39-65.

For important guidelines on writing directly to processor memory, refer to “Preventing Memory Corruption When You Export Coefficients to Processor Memory” on page 39-70.

Follow these steps to export filter coefficients from FDATool directly to memory on your processor.

- 1 Load the program file that contains your filter into CCS IDE to activate the program symbol table. The symbol must contain the global variables you use to store the filter coefficients and length parameters.
- 2 Start FDATool.
- 3 Click **Filter Manager** to open the Filter Manager dialog, shown in the following figure.



- 4 Highlight the filter to modify on the list of filters, and select **Edit current filter**. The highlighted filter appears in FDATool for you to change.

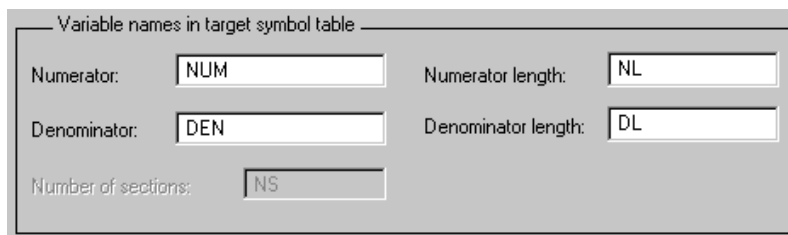
If you did not store your filter from a previous session, design the filter in FDATool and continue.

- 5 Click **Close** to dismiss the Filter Manager dialog.
- 6 Adjust the filter specifications in FDATool to modify its performance.
- 7 In FDATool, select **Targets > Code Composer Studio IDE** to open the Export to Code Composer Studio IDE dialog.

Keep the export dialog open while you work. When you do so, the contents update as you change the filter in FDATool.

Tip Click **Generate** to export coefficients to the same processor memory location multiple times without reentering variable names.

- 8 In the Export to Code Composer Studio dialog:
 - Set **Export mode** to **Write directly to memory**
 - Clear **Disable memory transfer warnings** to get a warning if your processor does not support the export data type.
- 9 In **Variable names in target symbol table**, enter the names of the variables in the processor symbol table that correspond to the memory allocated for the parameters, such as **Numerator** and **Denominator**. Your names must match the names of the filter coefficient variables in your program.



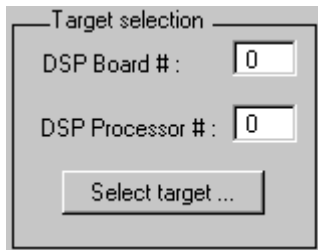
Variable names in target symbol table

Numerator:	<input type="text" value="NUM"/>	Numerator length:	<input type="text" value="NL"/>
Denominator:	<input type="text" value="DEN"/>	Denominator length:	<input type="text" value="DL"/>
Number of sections:	<input type="text" value="NS"/>		

- 10 Select **Export suggested** to accept the recommended export data type.

For more information about how FDATool determines the data type to suggest, refer to “How FDATool Determines the Export Suggested Data Type” on page 39-65.

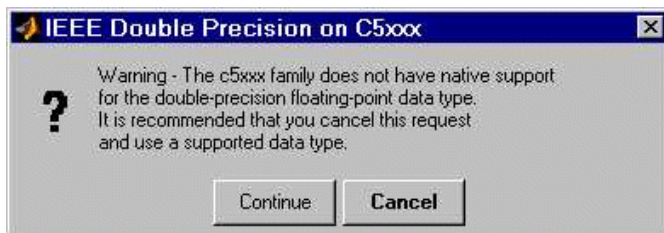
- 11 If you know the board number and processor number of your target, enter **DSP Board #** and **DSP Processor #** values to identify your board.



Note: When you have only one board or simulator, Embedded Coder sets **DSP Board #** and **DSP Processor #** to your board automatically.

If you have more than one board defined in CCS Setup:

- Click **Select target** to open the Selection Utility dialog.
 - Select the board name and processor name to use from the list of boards.
- 12** Click **Generate** to export your filter. If your processor does not support the data type you export, you see a warning similar to the following message.



You can continue to export the filter, or cancel the export process. To prevent this warning dialog from appearing, select **Disable memory transfer warnings** in the Export to Code Composer Studio IDE dialog.

- 13** (Optional) Continue to optimize filter performance by modifying your filter in FDATool and then export the updated filter coefficients directly to processor memory.
- 14** When you finish testing your filter, return to FDATool, and click **Store filter** to save your changes.

Using Makefiles with Code Composer Studio 3.x

In this section...

“Introduction” on page 39-76

“Set Up XMakefile for CCSv3” on page 39-76

“Prepare Your Model for CCSv3 and Makefiles” on page 39-77

“Create Target Configuration File in CCSv3” on page 39-78

“Load and Run the Embedded Software” on page 39-78

Introduction

This tutorial shows you how to use the XMakefile feature in your MathWorks software to build and run embedded software with Code Composer Studio 3.3 (CCSv3). For more information about XMakefile, see “Makefiles for Software Build Tool Chains” on page 36-15

Note: The Embedded Coder IDE Project Generator feature is not available for CCSv3 in the current release. For more information about IDE Project Generator, see “IDE Projects” on page 36-13

To build the target software, complete the process covered in this chapter:

- Set up XMakefile for CCSv3.
- Prepare your model for CCSv3.
- Create a Target Configuration File in CCSv3.
- Load and run the embedded software.

Set Up XMakefile for CCSv3

The XMakefile feature tells your MathWorks software how to create makefiles for a *configuration*, which is a specific combination of tool chain and embedded target hardware. Some configurations require additional information before you can use them.

Select and complete a configuration for Code Composer Studio 3.3 (CCSv3):

- 1 Enter `xmakefilesetup` at the MATLAB command prompt. This action opens the XMakefile User Configuration dialog.
- 2 Clear **Display operational configurations only**. This displays the configuration files, including ones that need updated path information.
- 3 For **Configurations**, select a configuration that matches your target and ends with `ccsv3`. Then click **Apply**.
- 4 If the configuration is incomplete, the software displays a series of **Browse For Folder** dialogs that include instructions to provide missing information.
- 5 Examine the **Tool Directories** tab to see if the paths are right.
- 6 When you have supplied the missing information, and the configuration is complete, click **OK** to close the XMakefile User Configuration dialog.

For example, to generate code for CCSv3 and a C6000 processor with DSP/BIOS:

- 1 Enter `xmakefilesetup` on the command line.
- 2 In the XMakefile dialog, clear **Display operational configurations only**, set **Configurations** to `ticcs_c6000_dsplibios_ccsv3`, and click **Apply**.
- 3 A **Browse For Folder** appears, stating “Select the C6000 Code Generation Tools root installation directory...”.

Browse and select a path such as `C:\Program Files\Texas Instruments\C6000 Code Generation Tools`.

- 4 Another **Browse For Folder** dialog appears, stating “Select the C6000 CSL root installation directory...”.

Browse and select a path such as `C:\Program Files\C6xCSL\`.

- 5 Examine the **Tool Directories** tab to see if the paths are right.
- 6 With the updated information, the `ticcs_c6000_dsplibios_ccsv3` configuration is operational. Click **OK** to save the updated configuration, and close the dialog.

Prepare Your Model for CCSv3 and Makefiles

- 1 Configure your model as described in “Configure Target Hardware Resources” on page 36-3
- 2 On the Coder Target pane, under the Tool Chain Automation tab, set **Build format** to **Makefile**.
- 3 Build your embedded software by pressing **CTRL+B**.

Create Target Configuration File in CCSv3

Before loading and running your target software, use the CCSv3 IDE to create a “target configuration file”. The TI Debug Server uses this file while it works with CCSv3 to load and run your target software. The XML-based target configuration file describes the target board and processor. The file name ends with a *.ccxml extension.

Create a target configuration file:

- 1 In the CCSv3, select **File > New > Target Configuration File** to display a **New Target Configuration** dialog:
 - For **File name**, update the file name that ends with **.ccxml** to describe your project and hardware.
 - Click **Finish**. This action displays a utility in the CCS editor pane for customizing the target configuration file.
- 2 Use the utility to select the **Connection** and **Device** type. Typing a partial string next to **Device** filters the list of devices.
- 3 Click **Save**.

Note: For more information about target configuration files, consult the Texas Instruments documentation for CCSv3.

Load and Run the Embedded Software

First set the Windows system variable, Path, so you can call the TI Debug Server Scripting (DSS) API from a folder.

- 1 In Windows, right-click **My Computer**, and select **Properties**. This action opens the System Properties dialog.
- 2 In **System Properties**, select the **Advanced** tab, and click **Environment Variables**. This action opens the Environment Variables dialog.
- 3 In **Environment Variables**, under **System variables**, select the Path variable, and click **Edit**. This action opens the Edit System Variable dialog.
- 4 In Edit System Variable, for **Variable value**, append a semicolon and the full path of the `\ccsv3\scripting\bin` subdirectory. For example, append `;%C:\ti\ccsv3\scripting\bin`.

Note: The path cannot contain spaces. Customize the installation directory when you install CCSv3 so it does not contain spaces.

For more information about using DSS, see http://processors.wiki.ti.com/index.php/Debug_Server_Scripting.

MathWorks provides an example JavaScript file, `runProgram.js`, for you to use with DSS. This script loads and runs the specified program on the target specified in the target configuration file. You can create a copy of this script and modify it to suit your needs. The location of `runProgram.js` is:

```
[MATLABROOT]\toolbox\idelink\extensions\ticcs\ccsdemos
```

The specific syntax for running `dss.bat` with `runProgram.js` is:

```
> dss runProgram.js targetConfigurationFile programFile
```

Replace *targetConfigurationFile* and *programFile* with paths and file names. For example, if you are using a working directory called the CCSv3 workspace, and the model name is `myProgram`, enter:

```
> dss runProgram.js c:\workspace\myC6416dsk.ccxml myProgram.out
```

This command builds and loads your software on the target or simulator.

You have completed the process of loading and running embedded software using XMakefile and CCSv3.

Reported Limitations and Tips

In this section...

“Example Programs Do Not Run Well with Incorrect GEL Files” on page 39-80
“Changing Values of Local Variables Does Not Work” on page 39-81
“Code Composer Studio Cannot Find a File After You Halt a Program” on page 39-81
“C54x XPC Register Can Be Modified Only Through the PC Register” on page 39-83
“Working with More Than One Installed Version of Code Composer Studio” on page 39-83
“Changing CCS Versions During a MATLAB Session” on page 39-84
“MATLAB Hangs When Code Composer Studio Cannot Find a Board” on page 39-84
“Using Mapped Drives” on page 39-85
“Uninstalling Code Composer Studio 3.3 Prevents Embedded Coder From Connecting” on page 39-85
“PostCodeGenCommand Commands Do Not Apply to IDE Projects” on page 39-86

Some long-standing issues apply to the Embedded Coder product. When you are using `ticcs` objects and the software methods to work with Code Composer Studio and supported hardware or simulators, recall the information in this section.

The latest issues in the list appear at the bottom. HIL refers to “hardware in the loop,” also called processor in the loop (PIL) here and in other applications, and sometimes referred to as function calls.

Example Programs Do Not Run Well with Incorrect GEL Files

To run the Embedded Coder examples, you must load the corresponding GEL files before you run the examples. For some boards, the examples run fine with the default CCS GEL file. Some boards need to run device-specific GEL files for the examples to work well.

Here are examples and boards which require specific GEL files.

- Board: C5416 DSK

Examples: `rtdxtutorial`, `rtdx1msdemo`

Emulator: XDS-510

GEL file to load: `c5416_dsk.gel`

In general, if an example does not run with the default GEL file, try using a device-specific GEL file by defining the file in the CCS Setup Utility.

Changing Values of Local Variables Does Not Work

If you halt the execution of your program on your DSP and modify a local variable's value, the new value may not be acknowledged by the compiler. If you continue to run your program, the compiler uses the original value of the variable.

This problem happens only with local variables. When you write to the local variable via the Code Composer Studio Watch Window or via a MATLAB object, you are writing into the variable's absolute location (register or address in memory).

However, within the processor function, the compiler sometimes saves the local variable's values in an intermediate location, such as in another register or to the stack. That intermediate location cannot be determined or changed/updated with a new value during execution. Thus the compiler uses the old, unchanged variable value from the intermediate location.

Code Composer Studio Cannot Find a File After You Halt a Program

When you halt a running program on your processor, Code Composer Studio may display a dialog that says it cannot find a source code file or a library file.

When you halt a program, CCS tries to display the source code associated with the current program counter. If the program stops in a system library like the runtime library, DSP/BIOS, or the board support library, it cannot find the source code for debug. You can either find the source code to debug it or select the **Don't show this message again** check box to ignore messages like this in the future.

For more information about how CCS responds to the halt, refer the online Help for CCS. In the online help system, use the search engine to search for the keywords "Troubleshooting" and "Support." The following information comes from the online help for CCS, starting with the error message:

File Not Found

The debugger is unable to locate the source file required to enable source-level debugging for this program.

To specify the location of the source file

- 1 Click **Yes**. The *Open* dialog appears.
- 2 In the *Open* dialog, specify the location and name of the source file then click **Open**.

The next section provides more details about file paths.

Defining a Search Path for Source Files

The *Directories* dialog enables you to specify the search path the debugger uses to find the source files included in a project.

To Specify Search Path Directories

- 1 Select **Option > Customize**.
- 2 In the *Customize* dialog, select the **Directories** tab. Use the scroll arrows at the top of the dialog to locate the tab.

The *Directories* dialog offers the following options.

- **Directories**. The **Directories** list displays the defined search path. The debugger searches the listed folders in order from top to bottom.

If two files have the same name and are located in different folders, the file located in the folder that appears highest in the **Directories** list takes precedence.

- **New**. To add a new folder to the **Directories** list, click **New**. Enter the full path or click **browse [...]** to navigate to a folder. By default, the new folder is added to the bottom of the list.
- **Delete**. Select a folder in the **Directories** list, then click **Delete** to remove that folder from the list.
- **Up**. Select a folder in the **Directories** list, then click **Up** to move that folder higher in the list.
- **Down**. Select a folder in the **Directories** list, then click **Down** to move that folder lower in the list.

- 3 Click **OK** to close the *Customize* dialog and save your changes.

C54x XPC Register Can Be Modified Only Through the PC Register

You cannot modify the XPC register value directly using `regwrite` to write into the register. When you are using extended program addressing in C54x, you can modify the XPC register by using `regwrite` to write a 23-bit data value in the PC register. Along with the 16-bit PC register, this operation also modifies the 7-bit XPC register that is used for extended program addressing. On the C54x, the PC register is 23 bits (7 bits in XPC + 16 bits in PC).

You can then read the XPC register value using `regread`.

Working with More Than One Installed Version of Code Composer Studio

When you have more than one version of Code Composer Studio installed on your machine, you cannot select which CCS version MATLAB Embedded Coder attaches to when you create a `ticcs` object. If, for example, you have both CCS for C5000 and CCS for C6000 versions installed, you cannot choose to connect to the C6000 version rather than the C5000 version.

When you issue the command

```
IDE_obj = ticcs
```

Embedded Coder starts the CCS version you last used. If you last used your C5000 version, the `IDE_obj` object accesses the C5000 version.

Workaround

To make your `ticcs` object access the right processor:

- 1 Start and close the CCS version you plan to use before you create the `ticcs` object in MATLAB.
- 2 Create the `ticcs` object using the `boardnum` and `procnum` properties to select your processor, if required.

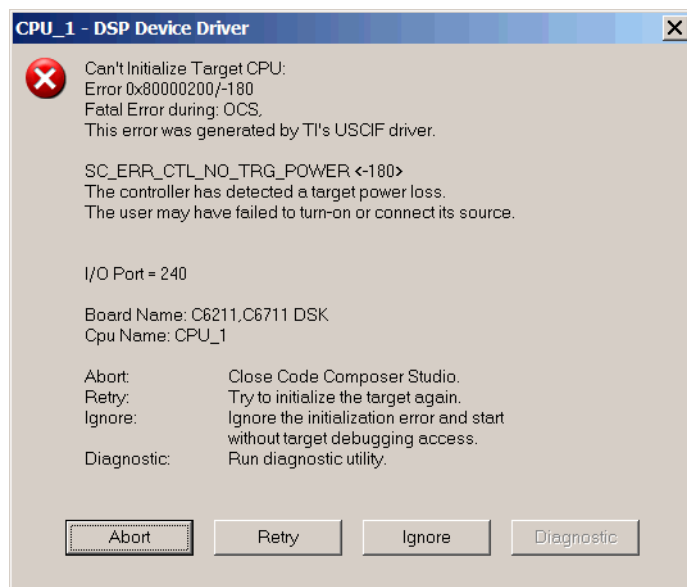
Recall that `ccsboardinfo` returns the `boardnum` and `procnum` values for the processors that CCS recognizes.

Changing CCS Versions During a MATLAB Session

You can use only one version of CCS in a single MATLAB session. Embedded Coder does not support using multiple versions of CCS in a MATLAB session. To use another CCS version, exit MATLAB software and restart it. Then create your links to the new version of CCS.

MATLAB Hangs When Code Composer Studio Cannot Find a Board

In MATLAB software, when you create a `ticcs` object, the construction process for the object automatically starts CCS. If CCS cannot find a processor that is connected to your PC, you see a message from CCS like the following DSP Device Driver dialog that indicates CCS could not initialize the processor.



Four options let you decide how to respond to the failure:

- **Abort** — Closes CCS and suspends control for about 30 seconds. If you used MATLAB software functions to open CCS, such as when you create a `ticcs` object, the system returns control to MATLAB command window after a considerable delay, and issues this warning:

??? Unable to establish connection with Code Composer Studio.

- **Ignore** — Starts CCS without connecting to a processor. In the CCS IDE you see a status message that says EMULATOR DISCONNECTED in the status area of the IDE. If you used MATLAB to start CCS, you get control immediately and Embedded Coder creates the `ticcs` object. Because CCS is not connected to a processor, you cannot use the object to perform processor operations from MATLAB, such as loading or running programs.
- **Retry** — CCS tries again to initialize the processor. If CCS continues not to find your hardware processor, the same DSP Device Driver dialog reappears. This process continues until either CCS finds the processor or you choose one of the other options to respond to the warning.

One more option, **Diagnostic**, lets you enter diagnostic mode if it is enabled. Usually, **Diagnostic** is not available for you to use.

Using Mapped Drives

Limitations in Code Composer Studio do not allow you to load programs after you set your CCS working folder to a read-only mapped drive. Load operations fail with an Application Error dialog.

The following combination of commands does not work:

- 1 `cd(IDE_obj, 'mapped_drive_letter')` % Change CCS working directory to read-only mapped drive.
- 2 `load(IDE_obj, 'program_file')` % Loading program fails.

Uninstalling Code Composer Studio 3.3 Prevents Embedded Coder From Connecting

Description On a machine where CCS 3.3 and CCS 3.1 are installed, uninstalling 3.3 makes 3.1 unusable from MATLAB. This is because the CCS 3.3 uninstaller leaves stale registry entries in the Windows Registry that prevent MATLAB from connecting to CCS 3.1.

Texas Instruments has documented this uninstall problem and the solution on their Web site.

Updated information on this issue may also be available from the Bug Reports section of www.mathworks.com at <http://www.mathworks.com/support/bugreports/379676>

PostCodeGenCommand Commands Do Not Apply to IDE Projects

PostCodeGenCommand commands, such as the “addCompileFlags” and “addLinkFlags” functions in the BuildInfo API do not alter code generated by Embedded Coder while **System Target File** is set to `idelink_ert.tlc` or `idelink_grt.tlc`.

Use the 'Compiler options string' and 'Linker options string' parameters located in the Configuration Parameters dialog (**Ctrl+E**) on the Code Generation > Coder Target pane instead. You can also automate this process using a model callback to SET_PARAM the 'CompilerOptionsStr' and 'LinkerOptionsStr' parameters.

Setting Up Code Composer Studio (ert.tlc System Target File)

In this section...

“Prepare Your Model for CCSv3.3” on page 39-87

“Prepare Your Model for CCSv4/5” on page 39-87

Prepare Your Model for CCSv3.3

For Code Composer Studio 3.3, select Texas Instruments Code Composer Studio v3.3 (C2000) as the toolchain, and press Ctrl + B to build and generate the .out (executable) file. To download and run the executable, use Code Composer Studio.

Prepare Your Model for CCSv4/5

- 1 Configure your model as described in “Coder Target Pane: Texas Instruments C2000 Processors”, setting **Toolchain** to Texas Instruments Code Composer Studio v4 (C2000) or Texas Instruments Code Composer Studio v5 (C2000).
- 2 If you want to load and run the application automatically after the build, on the **Coder Target** pane, select **Build**, load and run value from **Build action** and specify the *.ccxml file in **CCS hardware configuration file** that matches the selected target hardware.

Otherwise, you have to create your own .ccxml file that matches the target hardware and the connection. See “Create Target Configuration File in CCSv4/5” on page 39-87 to create your own .ccxml file (Target Configuration File) in CCSv4/5.

- 3 Build your embedded software by pressing **CTRL+B**.

Create Target Configuration File in CCSv4/5

Before loading and running your target software, use the CCSv4/5 IDE to create a “target configuration file”. The TI Debug Server uses this file while it works with CCSv4/5 to load and run your target software.

Create a target configuration file:

- 1 In the CCSv4/5, select **File > New > Target Configuration File** to display a **New Target Configuration** dialog:

- For **File name**, update the file name that ends with `.ccxml` to describe your project and hardware.
 - Click **Finish**. This action displays a utility in the CCSv4/5 editor pane for customizing the target configuration file.
- 2** Use the utility to select the **Connection** and **Device** type. Typing a partial string next to **Device** filters the list of devices.
 - 3** Click **Save**.

IDE Link Frequently Asked Question: Why do I get an error when I invoke TICCS?

In this section...

“Why do I get an error when I invoke TICCS?” on page 39-89

“How can I fix this problem?” on page 39-89

“What happens if I click Deselect All when CCS prompts that 'New components were detected'?” on page 39-91

“How do I use CCS Component Manager to enable IDE Link Components?” on page 39-91

Why do I get an error when I invoke TICCS?

IDE Link uses a plugin component that needs to be registered with CCS. If the plugin is not registered, you get the following error messages when you invoke TICCS.

Example 1:

```
cc = ticcs
```

```
Could not start the MATLAB(R) component in CCS.  
Please use IDE Link FAQ to troubleshoot the problem.
```

Example 2:

```
cc = ticcs
```

```
Could not start the MATLAB component in CCS.
```

How can I fix this problem?

To fix this problem, perform the following steps:

- 1 Register IDE Link components.
- 2 Enable IDE Link component.

Register IDE Link Components

The following instructions show you how to register the components required by IDE Link.

Note: Before you register the components, you must have write permission to modify the registry. If you do not have this permission, look for someone who has (e.g., system administrator) and have this person perform the registration.

- 1 Close Code Composer Studio IDE.
- 2 Close MATLAB.
- 3 Open a Microsoft Windows Command Prompt by clicking Start and then Programs > Accessories > Command Prompt.
- 4 Register the LinkCCS.dll component. In MATLAB Command Window, enter:

```
regsvr32 [MATLABROOT]\bin\win32\LinkCCS.dll
```

- 5 Verify that the preceding command displays the following message in a dialog box:

```
DllRegisterServer in [MATLABROOT]\bin\win32\LinkCCS.dll succeeded
```

- 6 Register the MWCCSStu.ocx component. In MATLAB Command Window, enter:

```
regsvr32 [MATLABROOT]\toolbox\idelink\extensions\ticcs\bin\win32\MWCCSStu.ocx
```

- 7 Verify that the preceding command displays the following message in a dialog box:

```
DllRegisterServer in [MATLABROOT]\toolbox\idelink\extensions\ticcs\bin\win32\MWCCSStu.ocx succeeded
```

After you register the components, enable the CCS component before using IDE Link.

To enable IDE Link Component

- 1 Open MATLAB
- 2 Enter `cc = ticcs` at the MATLAB prompt. CCS starts and prompts that "New components were detected".
- 3 Click Yes to enable components for compatible CCS releases. The first time CCS is invoked (either through TICCS or directly), it detects new components and asks if you want enable them. You see the following dialog:

Note: Click OK to enable the new component. For more information on ticcs or to see this FAQ again, enter `help ticcs` at the MATLAB prompt. For your convenience these instructions are available [here](#).

What happens if I click Deselect All when CCS prompts that 'New components were detected'?

You will not be able to use IDE Link. The New Components dialog box appears only once.

To enable the components after you click 'Deselect All', use the CCS Component Manager.

How do I use CCS Component Manager to enable IDE Link Components?

- 1** To open the CCS Component Manager, select Start and then Programs > Texas Instruments > Code Composer Studio 3.x > Component Manager. The left pane of the Component Manager presents the CCS installations select tree.
- 2** On the select tree, double-click The MathWorks, Inc. to see the MathWorks components.
- 3** Select the checkbox for [MATLABROOT]\toolbox\idelink\extensions\ticcs\bin\win32 to enable the component.
- 4** Select File > Save from the menu bar to save the new settings and then close the Component Manager.

Working with Texas Instruments Code Composer Studio 4 & 5 IDE

- “Set Up” on page 40-2
- “Code Composer Studio” on page 40-3
- “Getting Started” on page 40-4
- “Using Makefiles with Code Composer Studio 4/5” on page 40-5
- “Reported Limitations and Tips” on page 40-11

Set Up

Before you use Embedded Coder with Code Composer Studio (CCS IDE) for the first time, use the `checkEnvSetup` function to check for third-party tools and set environment variables. Run `checkEnvSetup` again whenever you configure CCS IDE to interact with a new board or processor, or upgrade the related third-party tools.

To verify that CCSv3 is installed on your machine and has at least one board configured, enter

```
ccsboardinfo
```

in the MATLAB Command Window. With CCS installed and configured, MATLAB software returns information about the boards that CCS recognizes on your machine, in a form similar to the following listing.

Board Num	Board Name	Proc Num	Processor Name	Processor Type
1	C6xxx Simulator (Texas Instrum	.0	6701	TMS320C6701
0	C6x13 DSK (Texas Instruments)	0	CPU	TMS320C6x1x

If MATLAB software does not return information about the boards, open your CCS installation and use the Setup Utility in CCS to configure at least one board.

As a final test, start CCS to verify that it runs. For Embedded Coder to operate with CCS, the CCS IDE must be able to run on its own.

Code Composer Studio

Feature Support

When you use Code Composer Studio 4 or Code Composer Studio 5 with Embedded Coder software, you can use:

- Makefiles to automate building and deploying software to target hardware. For more information, see “Makefiles for Software Build Tool Chains” and “Using Makefiles with Code Composer Studio 4/5” on page 40-5.
- Processor-in-the-loop (PIL) simulations to verify your software running on target hardware. For more information, see “PIL Simulation for IDE and Toolchain Targets”.
- Execution profiling with PIL to measure the performance of synchronous tasks running on target hardware. For more information, see “Execution Profiling During PIL Simulation”.

Features that require IDE projects (**Build format = Project**), such as IDE Project Generator and IDE Automation Interface, are not available for use with Code Composer Studio 4 & 5.

You can use CCSv4/5 with the Simulink Block Libraries for the following Texas Instruments processors:

- TI's C2000
- TI's C5000
- TI's C6000

Getting Started

In this section...
“Verifying Your Code Composer Studio Installation” on page 40-4
“Learning About Makefiles” on page 40-4

Verifying Your Code Composer Studio Installation

On your host computer, install CCSv4/5 and other third-party tools for your board and processor, and set the environment variables. Then use the `checkEnvSetup` function in MATLAB to verify that your setup includes the required software. For more information and examples, see `checkEnvSetup`.

Learning About Makefiles

To learn about using makefiles, see “Makefiles for Software Build Tool Chains” on page 36-15.

For an example of using CCSv4/5 with makefiles, model-block PIL, and the Serial Communications Interface (SCI), see “Performing a Model Block PIL Simulation via SCI Using Makefiles”.

Using Makefiles with Code Composer Studio 4/5

In this section...

“Introduction” on page 40-5

“Set Up XMakefile for CCSv4/5” on page 40-5

“Prepare Your Model for CCSv4/5 and Makefiles” on page 40-6

“Create Target Configuration File in CCSv4/5” on page 40-7

“Configure Windows Path for TI Debug Server Scripting (DSS)” on page 40-7

“Load and Run the Embedded Software Using DSS” on page 40-8

Introduction

This tutorial shows you how to use the XMakefile feature in your MathWorks software to build and run embedded software with Code Composer Studio 4 or 5 (CCSv4/5). For more information about XMakefile, see “Makefiles for Software Build Tool Chains” on page 36-15

Note: The Embedded Coder IDE Project Generator feature is not available for CCSv4/5 in the current release. For more information about IDE Project Generator, see “IDE Projects” on page 36-13

To build the target software, complete the process covered in this chapter:

- Set up XMakefile for CCSv4/5.
- Prepare your model for CCSv4/5.
- Create a Target Configuration File in CCSv4/5.
- Load and run the embedded software.

Set Up XMakefile for CCSv4/5

The XMakefile feature tells your MathWorks software how to create makefiles for a *configuration*, which is a specific combination of tool chain and embedded target hardware. Some configurations require additional information before you can use them.

Select and complete a configuration for Code Composer Studio 4 & 5 (CCSv4/5):

- 1 Enter `xmakefilesetup` at the MATLAB command prompt. This action opens the XMakefile User Configuration dialog.
- 2 Clear **Display operational configurations only**. This displays the configuration files, including ones that need updated path information.
- 3 For **Configurations**, select a configuration that matches your target and ends with `ccsv4` or `ccsv5`. Then click **Apply**.
- 4 If the configuration is incomplete, the software displays a series of **Browse For Folder** dialogs that include instructions to provide missing information.
- 5 Examine the **Tool Directories** tab to see if the paths are right.
- 6 When you have supplied the missing information, and the configuration is complete, click **OK** to close the XMakefile User Configuration dialog.

For example, to generate code for CCSv4/5 and a C6000 processor with DSP/BIOS:

- 1 Enter `xmakefilesetup` on the command line.
- 2 In the XMakefile dialog, clear **Display operational configurations only**, set **Configurations** to `ticcs_c6000_dspbios_ccsv4` or `ticcs_c6000_dspbios_ccsv5`, and click **Apply**.
- 3 A **Browse For Folder** appears, stating “Select the C6000 Code Generation Tools root installation directory...”.

Browse and select a path such as `C:\Program Files\Texas Instruments\C6000 Code Generation Tools`.

- 4 Another **Browse For Folder** dialog appears, stating “Select the C6000 CSL root installation directory...”.

Browse and select a path such as `C:\Program Files\C6xCSL\`.

- 5 Examine the **Tool Directories** tab and verify the paths shown there. Verify the DSP/BIOS and XDC installation folders.
- 6 With the updated information, the `ticcs_c6000_dspbios_ccsv4` or `ticcs_c6000_dspbios_ccsv5` configuration is operational. Click **OK** to save the updated configuration, and close the dialog.

Prepare Your Model for CCSv4/5 and Makefiles

- 1 Configure your model as described in “Configure Target Hardware Resources” on page 36-3, setting **IDE/Tool Chain** to Texas Instruments Code Composer

Studio v4 (makefile generation only) or Texas Instruments Code Composer Studio v5 (makefile generation only).

Choosing either of those options automatically sets **Build format** to Makefile.

- 2 Build your embedded software by pressing **CTRL+B**.

Create Target Configuration File in CCSv4/5

Before loading and running your target software, use the CCSv4/5 IDE to create a “target configuration file”. The TI Debug Server uses this file while it works with CCSv4/5 to load and run your target software. The XML-based target configuration file describes the target board and processor. The file name ends with a ***.ccxml** extension.

Create a target configuration file:

- 1 In the CCSv4/5, select **File > New > Target Configuration File** to display a **New Target Configuration** dialog:
 - For **File name**, update the file name that ends with **.ccxml** to describe your project and hardware.
 - Click **Finish**. This action displays a utility in the CCSv4/5 editor pane for customizing the target configuration file.
- 2 Use the utility to select the **Connection** and **Device** type. Typing a partial string next to **Device** filters the list of devices.
- 3 Click **Save**.

Note: For more information about target configuration files, consult the Texas Instruments documentation for CCSv4/5.

Configure Windows Path for TI Debug Server Scripting (DSS)

Set the Windows system variable, Path, so you can call the TI Debug Server Scripting (DSS) API from a folder.

- 1 In Windows, right-click **My Computer**, and select **Properties**. This action opens the System Properties dialog.
- 2 In **System Properties**, select the **Advanced** tab, and click **Environment Variables**. This action opens the Environment Variables dialog.

- 3 In **Environment Variables**, under **System variables**, select the Path variable, and click **Edit**. This action opens the Edit System Variable dialog.
- 4 In Edit System Variable, for **Variable value**, append a semicolon and the full path of the `\ccsv4\scripting\bin` or `\ccsv5\scripting\bin` subdirectory. For example, append `;C:\ti\ccsv4\scripting\bin`.

Note: The path cannot contain spaces. Customize the installation directory when you install CCSv4/5 so it does not contain spaces.

Load and Run the Embedded Software Using DSS

MathWorks provides an example JavaScript file, `runProgram.js`, for you to use with DSS. This script loads and runs the specified program on the target specified in the target configuration file. You can create a copy of this script and modify it to suit your needs. The location of `runProgram.js` is:

```
[MATLABROOT]\toolbox\idelink\extensions\ticcs\ccsdemos
```

Load and Run Embedded Software Automatically Using XMakefile Configuration

The syntax for running `dss.bat` with `runProgram.js` using XMakefile Configuration is:

```
> dss runProgramFile targetConfigurationFile [|||MW_XMK_GENERATED_TARGET_REF[E]|||]
```

For example, to automate, load, and run an embedded software on C2000 processor with CCSv4/5 using xmakefile configuration, the steps are as follows:

- 1 Enter `xmakefilesetup` on the [MATLAB] command prompt.
- 2 In the **XMakefile User Configuration** dialog box, clear **Display operational configurations only** check box.
- 3 Select `ticcs_c2000_ccsv4` or `ticcs_c2000_ccsv5` in **Configuration** parameter.
- 4 Click **Apply**.
- 5 Create a new Configuration as explained in the section, “Creating a New XMakefile Configuration”.
- 6 Select **Execute** tab. In this tab, enter values for the syntax

> dss runProgramFile targetConfigurationFile [|||MW_XMK_GENERATED_TARGET_REF[E]|||]
 in the **Execute tool** and **Arguments** parameters.

- In the **Execute tool** parameter, enter the executable name 'dss.bat'.
- In the **Arguments** parameter, replace runProgramFile and targetConfigurationFile with complete filepath.

When you enter the filepath, use double quotes or double back-slashes.

For example:

```
[MATLABROOT]\\toolbox\\idmlink\\extensions\\ticcs\\ccsdemos\\
\\runProgram.js
```

```
C:\\ccsv4\\CCSTargetConfigurations\\ezdsp28335.ccxml
```

```
[|||MW_XMK_GENERATED_TARGET_REF[E]|||]
```

or

```
"[MATLABROOT]toolbox\\idmlink\\extensions\\ticcs\\ccsdemos\\runProgram.js"
```

```
"C:\\ccsv4\\CCSTargetConfigurations\\ezdsp28335.ccxml"
```

```
[|||MW_XMK_GENERATED_TARGET_REF[E]|||]
```

You have completed the process of loading and running the embedded software using XMakefile on the target or simulator.

Load and Run Embedded Software Using Windows Command Prompt

The syntax for running dss.bat with runProgram.js using Windows command prompt is:

```
> dss runProgramFile targetConfigurationFile programFile
```

Replace runProgramFile, targetConfigurationFile, and the programFile with complete file path and the file name. For example, if you are using a working directory called 'CCSv4/5 workspace', and the model name 'myProgram', enter,

```
> dss c:\\ccsv4\\runProgram.js c:\\workspace\\myC6416dsk.ccxml
c:\\test\\myProgram.out
```

This command loads and runs the embedded software on the target or simulator.

For more information about using DSS, see http://processors.wiki.ti.com/index.php/Debug_Server_Scripting.

Troubleshooting DSS

With Code Composer Studio 4 & 5, using `runProgram.js` to download and run the generated program file can produce an error message similar to:

```
SEVERE: Could not open session. Found 2 devices matching: .*
Spectrum Digital DSK-EVM-eZdsp onboard USB Emulator_0/C64XP_0
Spectrum Digital DSK-EVM-eZdsp onboard USB Emulator_0/IcePick_C_0
```

If this happens, specify one of the devices in line 65 in `runProgram.js`. For example:

```
debugSession = debugServer.openSession(".*C64XP.*");
```

Alternatively, to connect to the first board & CPU detected, use ("`*`", "`*`") in line 65. For example:

```
debugSession = debugServer.openSession("*", "*");
```

Remember to restore the original code to work with other devices.

Advanced DSS Features

To use advanced DSS features, you can also use the CCSv4/5 example batch file, `loadti.bat`, as follows:

Change directories to the `loadti` subdirectory. For example:

```
> cd c:\ccs4_install\ccsv4\scripting\examples\loadti
```

Run `loadti.bat` using the following syntax:

```
> loadti -a -c=targetConfigurationFile programFile
```

Replace `targetConfigurationFile` with the complete path of the target configuration file.

Replace `programFile` with the name of the `.out` created using the XMakefile feature. For example:

```
> loadti -a -c=c:\workspace\myC6416dsk.ccxml myProgram.out
```

For more information about `loadti` and its options, type the following on your system command prompt

```
> loadti -help
```

Reported Limitations and Tips

In this section...

“Example Programs Do Not Run well with Incorrect GEL Files” on page 40-11

“PostCodeGenCommand Commands Do Not Apply to IDE Projects” on page 40-11

Some long-standing issues apply to the Embedded Coder product.

Example Programs Do Not Run well with Incorrect GEL Files

To run the Embedded Coder examples, you must load the GEL files before you run the examples. For some boards, the examples run fine with the default CCSv4/5 GEL file. Some boards need to run device-specific GEL files for the examples to work.

Here are examples and boards which require specific GEL files.

- Board: C5416 DSK

Examples: `rtdxtutorial`, `rtdx1msdemo`

Emulator: XDS-510

GEL file to load: `c5416_dsk.gel`

In general, if a example does not run with the default GEL file, try using a device-specific GEL file by defining the file in the CCSv4/5 Setup Utility.

PostCodeGenCommand Commands Do Not Apply to IDE Projects

PostCodeGenCommand commands, such as the “addCompileFlags” and “addLinkFlags” functions in the BuildInfo API do not alter code generated while **System Target File** is set to `idelink_ert.tlc` or `idelink_grt.tlc`.

Use the 'Compiler options string' and 'Linker options string' parameters located in the Configuration Parameters dialog (**Ctrl+E**) on the **Code Generation > Coder Target** pane instead. You can also automate this process using a model callback to SET_PARAM the 'CompilerOptionsStr' and 'LinkerOptionsStr' parameters.

Working with Texas Instruments C2000 Processors

- “Install Support for C2000 Processors” on page 41-2
- “Data Type Support” on page 41-6
- “Scheduling and Timing” on page 41-7
- “Sharing General Purpose Timers between C281x Peripherals” on page 41-13
- “Overview of Creating Models for C2000 Processors” on page 41-21
- “Using the c2000lib Blockset” on page 41-23
- “Configuring Timing Parameters for CAN Blocks” on page 41-28
- “ADC-PWM Synchronization via ADC Interrupt” on page 41-44
- “Configuring Acquisition Window Width for ADC Blocks” on page 41-49
- “Real-Time Data Exchange via RTDX™” on page 41-55
- “SPI-Based Control of PWM Duty Cycle” on page 41-57
- “HIL Verification of IIR Filter via SCI” on page 41-60
- “CAN-Based Control of PWM Duty Cycle” on page 41-66
- “CAN Calibration Protocol and External Mode” on page 41-69
- “Using the I2C Bus to Access a Connected EEPROM” on page 41-77
- “Using the IQmath Library” on page 41-81
- “Programming Flash Memory” on page 41-89
- “Configuring LIN Communications” on page 41-94
- “Open Examples for Embedded Coder Texas Instruments C2000” on page 41-96
- “Tips and Limitations” on page 41-98
- “RTDX Simulation Blocks” on page 41-99

Install Support for C2000 Processors

In this section...

“Workflows for Texas Instruments C2000” on page 41-2

“Install Code Composer Studio and control SUITE” on page 41-2

“Install Texas Instruments C2000” on page 41-3

“Configuring and Verifying” on page 41-5

Workflows for Texas Instruments C2000

Embedded Coder Support Package for Texas Instruments C2000 Processors supports two workflows, `ert.tlc` and `idelink_ert.tlc` for **System target file**. Both the workflows support Code Composer Studio 3.3, 4 & 5 (CCSv3/4/5).

- The `idelink_ert.tlc` workflow supports IDE Automation Interface and IDE Project Generator component with Code Composer Studio 3.3 (CCSv3). For CCSv4/5, the workflow supports makefile generation only.
- If you are a new user, the `ert.tlc` system target file is the preferred option over the `idelink_ert.tlc`. This option provides a uniform model configuration workflow across the phases of development such as simulation, target prototyping and production code generation. This workflow also supports ‘auto-download’ feature.

Go to the vendor Web sites to find 64-bit compatible drivers for installing CCSv3 on 64-bits machine with drivers such as, Spectrum Digital Emulators or BlackHawk Emulators. Most vendors have 64-bit drivers that are compatible with CCSv3.

For more information about Embedded Coder support with `idelink_ert.tlc`, see “Code Composer Studio” on page 39-3.

For more information about Embedded Coder support for CCSv3 and CCSv4/5 with `ert.tlc`, see “Setting Up Code Composer Studio (ert.tlc System Target File)”.

Install Code Composer Studio and control SUITE

For the host—side CAN functionality, you must install 32-bit version of MATLAB on 64-bit machines.

Install the software in the following order:

- 1 Install TI CCS.
- 2 Install TI Service Release for CCS.
- 3 Install the TI Code Generation Tools for Windows.
- 4 If you are using a Spectrum Digital board, download and install the matching Spectrum Digital Driver.
- 5 If you are using RTDX for C28x host/target communications, download and install TI DSP/BIOS.
- 6 If you program flash memory with standalone code, download the TI Flash API for your target hardware.

Configure CCSv3 as follows:

- 1 In CCSv3, open **Help > About > Component manager > Build tools > TMS320C28XX** and select (check) **C2000 Code Generation Tools** .
- 2 With the Component manager open, open **Target Content (DSP/BIOS) > TMS320C28XX** and select **Texas Instruments DSP/BIOS**.
- 3 Save, exit, and restart CCSv3.

Install Texas Instruments C2000

Uninstall unsupported or untested versions of the third party products *before* installing supported versions. This prevents errors that occur when the Windows Environment Variables point to the unsupported versions.

Before you install the software, if you install the Texas Instruments (TI) Code Composer Studio CCSv3 on Windows 7, then, make sure that:

- You have administrator privileges.
- You have set the User Account Control (UAC) settings to the lowest level.
- You install CCSv3 in a folder other than Program Files.

Using this installation process, you download and install the following items on your host computer:

- Related Embedded Coder software features.
- Simulink block library called “ Support Package for Texas Instruments TI C2000 Hardware” .

- Related examples.

To install Embedded Coder Texas Instruments C2000 support package:

- 1** In a MATLAB command window, enter `supportPackageInstaller`. This starts the **Support Package Installer** wizard.
- 2** On the **Select an action** screen, select **Install from Internet** and click **Next**.
- 3** On the **Select support package to install** screen, select **Texas Instruments C2000** from the **Support for** list.
- 4** Specify the path to install in the **Installation folder**. Make sure you have the write privilege for the installation folder.
- 5** Click **Next** and follow the instructions on the subsequent pages of the **Support Package Installer** wizard.
- 6** On the **Confirm installation** screen, click **Install**.
- 7** The **Install/update complete** screen indicates the support package installation is complete and you need to continue with additional steps for installing the third party tools.

Click **Continue** and follow the instructions on the subsequent screens.

- 8** Specify the `CCSv3`, `CCSv4`, or `CCSv5` folder path.
- 9** On the **Validate controlSuite installation** screen, specify the `controlSUITE` installation folder path and click **Validate**.

Note Before you generate code for Texas Instruments processors, you must configure your environment by creating the system environment variables used by code generation software. To configure your environment type, execute the function `checkEnvSetup` in MATLAB Command Window. To know more on `checkEnvSetup`, see “`checkEnvSetup`”.

On the **Confirm third party installation** screen, click **Confirm**.

- 10** When Support Package Installer confirms that the installation is complete, click **Finish**. By default, Support Package Installer displays the examples for Texas Instruments C2000. You can use Texas Instruments C2000.

Note You can also use `targetupdater` function in the MATLAB command window to install/update the latest third party software for the support package.

Configuring and Verifying

To determine whether Embedded Coder software is present on your system, enter this command at the MATLAB prompt:

```
c2000lib
```

MATLAB displays the C2000 block libraries.

If you do not see the listed libraries, or MATLAB does not recognize the command, install the Embedded Coder software. Without the software, you cannot use Simulink and Simulink Coder software to develop applications targeted to the TI boards.

To verify that Code Composer Studio 3.3 (CCSv3) is present on your machine, enter this command at the MATLAB prompt:

```
ccsboardinfo
```

With CCSv3 installed and configured, MATLAB returns a list of the boards that CCSv3 recognizes on your machine like the following example:

Board Num	Board Name	Proc Num	Processor Name	Processor Type
1	F2812 Simulator	0	CPU	TMS320C28xx
0	F2812 PP Emulator	0	CPU_1	TMS320C28xx

If MATLAB does not return information about boards, revisit the installation and setup instructions in your CCSv3 documentation. If you have not done so already, install the third-party “Board Support Packages” for your boards.

As a final test, launch CCSv3 to verify that it runs. For Embedded Coder software to operate with CCSv3, the CCSv3 IDE must be able to run on its own.

Note: For a model to work in the targeting environment, select the discrete-time solver in the **Solver** pane of the Simulink Configuration Parameters dialog. Targeting does not work with continuous-time solvers.

To change the discrete-time solver, select **Simulation > Model Configuration Parameters** in the Simulink Editor. In the Configuration Parameters dialog, select the **Solver** pane, and set the **Solver** parameter to **Discrete (no continuous states)**.

Data Type Support

TI C2000 DSPs support 16 and 32-bit data types, but does not have native 8-bit data types. Simulink models and Embedded Coder software support many data types, including 8-bit data types.

If you select `int8` or `uint8` in your model, your simulation runs with 8-bit data, but in the generated code, that data will be represented as 16-bit. This may cause instances where data overflow and wraparound occurs in the simulation, but not in the generated code.

For example, to make the overflow behavior of the simulation and generated code match for a Simulink Add block in your model, select **Saturate on integer overflow** in that block.

Scheduling and Timing

In this section...

“Overview” on page 41-7

“Timer-Based Interrupt Processing” on page 41-7

“Asynchronous Scheduling” on page 41-8

Overview

Normally the code generated by Embedded Coder software runs in the context of a timer interrupt. Model blocks run in a periodical fashion clocked by the periodical interrupt whose period is tied to the base sample time of the model.

This execution scheduling model, however, is not flexible enough for many systems, especially control and communication systems, which must respond to external events in real time. Such systems require the ability to handle various hardware interrupts in an asynchronous fashion.

Embedded Coder software lets you model and generate code for such systems by creating tasks driven by Hardware Interrupt blocks in addition to the tasks that are left to be handled in the context of the timer interrupt.

Timer-Based Interrupt Processing

For code that runs in the context of the timer interrupt, each iteration of the model solver is run after an interrupt has been posted and serviced by an interrupt service routine (ISR). The code generated for the C280x, C281x, and C28x3x uses `CPU_timer0`.

The timer is configured so that the model's base rate sample time corresponds to the interrupt rate. The timer period and prescaler are calculated and set up to produce the desired rate as follows:

$$BaseRateSampleTime = \frac{TimerPeriod}{TimerClockSpeed}$$

The minimum achievable base rate sample time depends on the model complexity. The maximum value depends on the maximum timer period value ($2^{32}-1$ for the F2812,

F2808, and F28x35) and the CPU clock speed. The CPU clock speed is 100 MHz for the F2808, and 150 MHz for the F2812 and F28335.

If the blocks in the model inherit their sample time value, and a sample time is not explicitly defined, the default value is 0.2 s.

High-Speed Peripheral Clock

The Event Managers and their general-purpose timers, which drive PWM waveform generation use the high-speed peripheral clock (HISCLK). By default, this clock is selected in Embedded Coder software. This clock is derived from the system clock (SYSCLKOUT):

$$\text{HISCLK} = [\text{SYSCLKOUT} / (\text{high-speed peripheral prescaler})]$$

The high-speed peripheral prescaler is determined by the HSPCLK bits set in SysCtrl. The default value of HSPCLK is 1, which corresponds to a high-speed peripheral prescaler value of 2.

For example, on the F2812, the HISCLK rate becomes
 $\text{HISCLK} = 150 \text{ MHz} / 2 = 75 \text{ MHz}$

Asynchronous Scheduling

This example shows how to use the c28x peripherals and Hardware Interrupt blocks to control the execution of function-call subsystems in an asynchronous fashion.

Required Hardware:

- Spectrum Digital F2808/F2812/F28335 eZdsp
- Texas Instruments F2808/F28035/F28069/F28335/F28343 ControlCARD with docking station
- Texas Instruments F28069 ControlSTICK
- Oscilloscope and probes

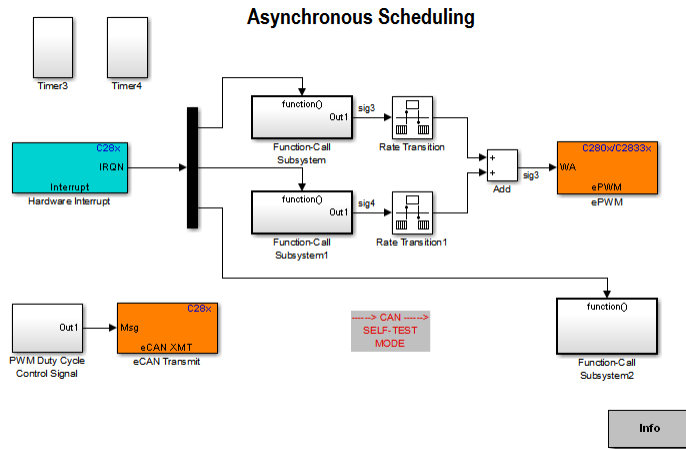
Available versions of this example:

- **F281x** based board: c281x_asynscheduling
- **F280x/F2823x/F2833x** based board: c280x_2833x_asynscheduling

- **Piccolo F2803x/F2806x** based board: c280xx_asyncscheduling

Model

The following figure shows the example model.



Description

The EV Timer blocks are used to configure timer interrupts to be raised on period match. Hardware Interrupt block installs the ISRs for the timer interrupts as well as for an eCAN message receive interrupt. The ISRs in turn call function-call subsystems connected to the Hardware Interrupt block output ports.

The outputs of first two subsystems are running counters. Their sum is used to control the duty cycle of the PWMB for F2812 or ePWM2 for F2808/F28335. The PWM waveform duty cycle increases linearly from 0 to 100 %.

The third subsystem contains an eCAN Rcv block whose message output drives a PWM block (PWMA for F2812 or ePWM1 for F2808/F28335) to control its duty cycle. The duty cycle varies from 0 to 100 % as the eCAN messages are received from an eCAN Xmt block.

NOTE: The **Self-Test** mode of eCAN_A is enabled to connect eCAN_A transmitter and receiver internally to avoid external connection between transmitter and receiver. Disabling the **Self-Test** mode for eCAN_A requires transmitter and receiver to be connected together externally.

How to Run the Example on the Board

The following sections explain how to configure, build, and deploy the executable.

With Code Composer Studio v3.3 (CCSv3.3) - Default

- Open the model corresponding to the family of processor you are using.
- Go to **Peripherals > eCAN_A** and check **Self-Test Mode**.
- Click **Incremental build** to generate, build, load and run the program.
- Observe the PWM waveform on the oscilloscope.

With Code Composer Studio v4 (CCSv4)

CCSv4 support is provided via a makefile based approach. This requires running `xmakefilesetup` to set the environment. You can ignore the following if you have already gone through these steps. For more information, please refer to the Xmakefile Documentation

- Run Xmakefile User Configuration.
- Set the Template parameter to **gmake**.
- Uncheck the **Display operational configuration only** checkbox.
- Set the Configuration parameter to **ticcs_c2000_ccsv4**, click **Apply**, if you haven't used this tool before, a dialog will ask you to browse to your CCSv4 installation folder. Pick the right folder and click **OK**.
- Click on the **Tool Directories** and make sure that the paths are pointing to directories matching your installation. For example:

- 1 *C:\Program Files\Texas Instruments\ccsv4* for **CCS Installation**
- 2 *C:\Program Files\Texas Instruments\ccsv4\tools\compiler\C2000* for **Code Generation Tools**
- 3 *C:\Program Files\Texas Instruments\bios_5_41_10_36* for **DSP/BIOS Installation**

Code Generation and Creation of an .out file using CCSv4:

- Open the model corresponding to the family of processor you are using.
- Open the Coder Target pane. Under the Target Hardware Resources tab, set the **IDE/Tool chain** parameter to

Texas Instruments Code Composer Studio v4 (makefile generation only).

- Change the selected board and processor to match your hardware.
- Go to **Peripherals > eCAN_A** and check **Self-Test Mode**.
- Click **Incremental build** or press **Ctrl+B** to generate and build an .out executable file.
- Open CCSv4 and Connect to the Hardware. Load the program on to the Hardware using CCSv4 and run.
- Observe the PWM waveform on the oscilloscope.

With Code Composer Studio v5 (CCSv5)

CCSv5 support is provided via a makefile based approach. This requires running `xmakefilesetup` to set the environment. You can ignore the following if you have already gone through these steps. For more information, please refer to the Xmakefile Documentation

- Run XMakefile User Configuration.
- Set the Template parameter to **gmake**.
- Uncheck the **Display operational configuration only** checkbox.
- Set the Configuration parameter to **ticcs_c2000_ccsv5**, click **Apply**, if you haven't used this tool before, a dialog will ask you to browse to your CCSv5 installation folder.

Pick the right folder and click **OK**.

- Click on the **Tool Directories** and make sure that the paths are pointing to directories matching your installation. For example:

- 1 `C:\ti\ccsv5\` for **CCS Installation**
- 2 `C:\ti\ccsv5\tools\compiler\c2000\` for **Code Generation Tools**
- 3 `C:\ti\bios_5_41_11_38\` for **DSP/BIOS Installation**

Code Generation and Creation of an .out file using CCSv5:

- Open the model corresponding to the family of processor you are using.
- Open the Coder Target pane. Under the Target Hardware Resources tab, set the **IDE/Tool chain** parameter to Texas Instruments Code Composer Studio v5 (makefile generation only).

- Change the selected board and processor to match your hardware.
- Go to **Peripherals > eCAN_A** and check **Self-Test Mode**.
- Click **Incremental build** or press **Ctrl+B** to generate and build an .out executable file.
- Open CCSv5 and Connect to the Hardware. Load the program on to the Hardware using CCSv5 and run.
- Observe the PWM waveform on the oscilloscope.

Sharing General Purpose Timers between C281x Peripherals

In this section...

“Sharing General Purpose Timers between CAP and eCAN” on page 41-14

“Sharing General Purpose Timers between CAP and SPI” on page 41-17

TMS320x281x DSP devices have four General Purpose (GP) timers. Each Event Manager (EV) module includes two GP timers:

- EVA includes GP Timer 1 and GP Timer 2.
- EVB includes GP Timer 3 and GP Timer 4.

You can use the GP Timers independently or to operate peripherals associated with the EV Manager, such as PWM, QEP, and CAP.

The following table describes the timer-peripheral mapping of the c281xlib block library.

GP Timer Use for C281x Peripheral Blocks

	GP Timer 1	GP Timer 2	GP Timer 3	GP Timer 4
PWM1-PWM6	#			
PWM7-PWM12			#	
QEP1-QEP2		#		
QEP3-QEP4				#
CAP1-CAP3	#	#		
CAP4-CAP6			#	#

Each PWM and QEP peripheral has access to only one timer, while each CAP peripheral has access to two timers. In the PWM and QEP blocks, you can set the **Module** option to A or B to determine which unique timer-peripheral combination the block configures. By comparison, in the CAP block, you can use the **Time base** option to select one of two timers for each CAP peripheral.

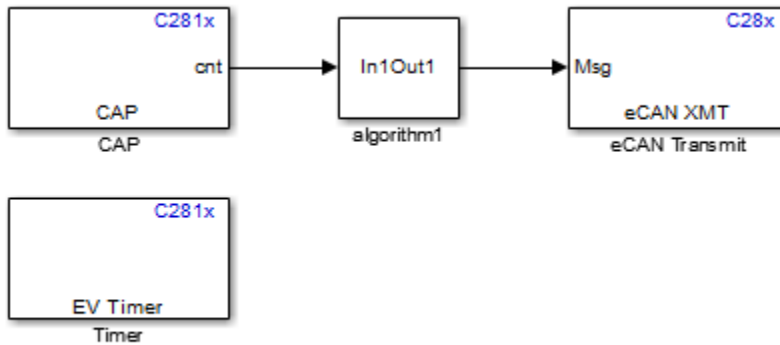
Each GP timer is available to multiple peripherals. For example:

- PWM1-PWM6 and CAP1-CAP3 share GP Timer 1

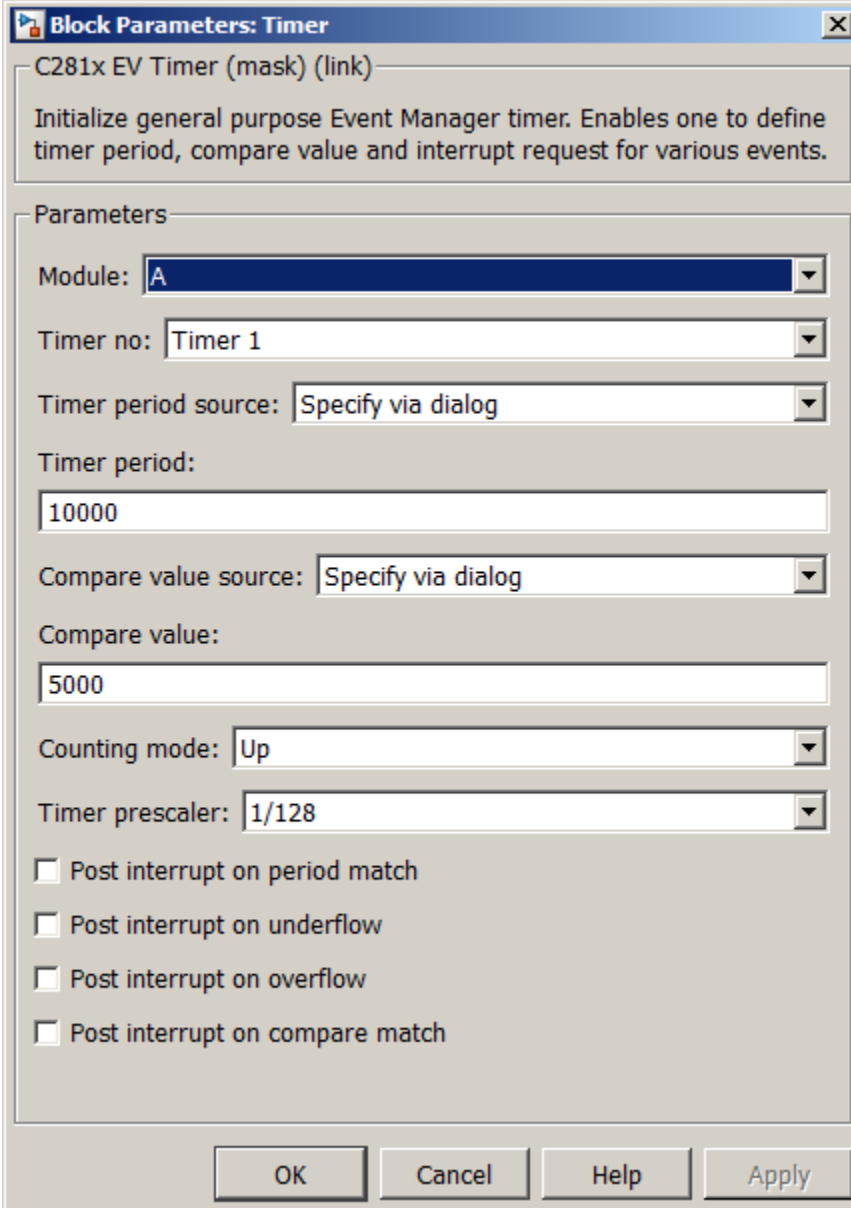
- PWM7-PWM12 and CAP4-CAP6 share GP Timer 3
- QEP1-QEP2 and CAP1-CAP3 share GP Timer 2
- QEP3-QEP4 and CAP4-CAP6 share GP Timer 4

The PWM, QEP, CAP, and Timer blocks each provide independent access to key timer registers. If the blocks in your model share a specific GP timer, check that the timer-related settings are compatible. If the peripheral settings for a shared timer are not compatible, the software generates an error when you update the model or generate code.

Sharing General Purpose Timers between CAP and eCAN



The model contains Timer and CAP blocks that both use Timer 1 (GP Timer 1).



Block Parameters: Timer [X]

-C281x EV Timer (mask) (link)-

Initialize general purpose Event Manager timer. Enables one to define timer period, compare value and interrupt request for various events.

Parameters

Module: **A**

Timer no: **Timer 1**

Timer period source: **Specify via dialog**

Timer period: **10000**

Compare value source: **Specify via dialog**

Compare value: **5000**

Counting mode: **Up**

Timer prescaler: **1/128**

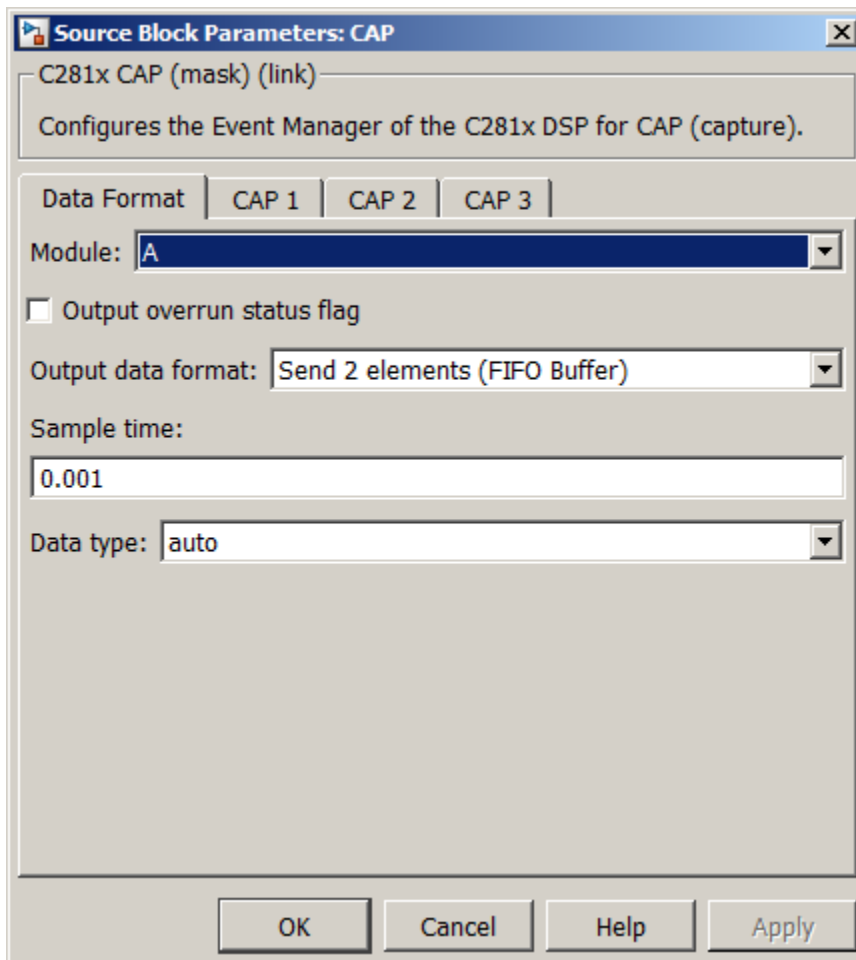
Post interrupt on period match

Post interrupt on underflow

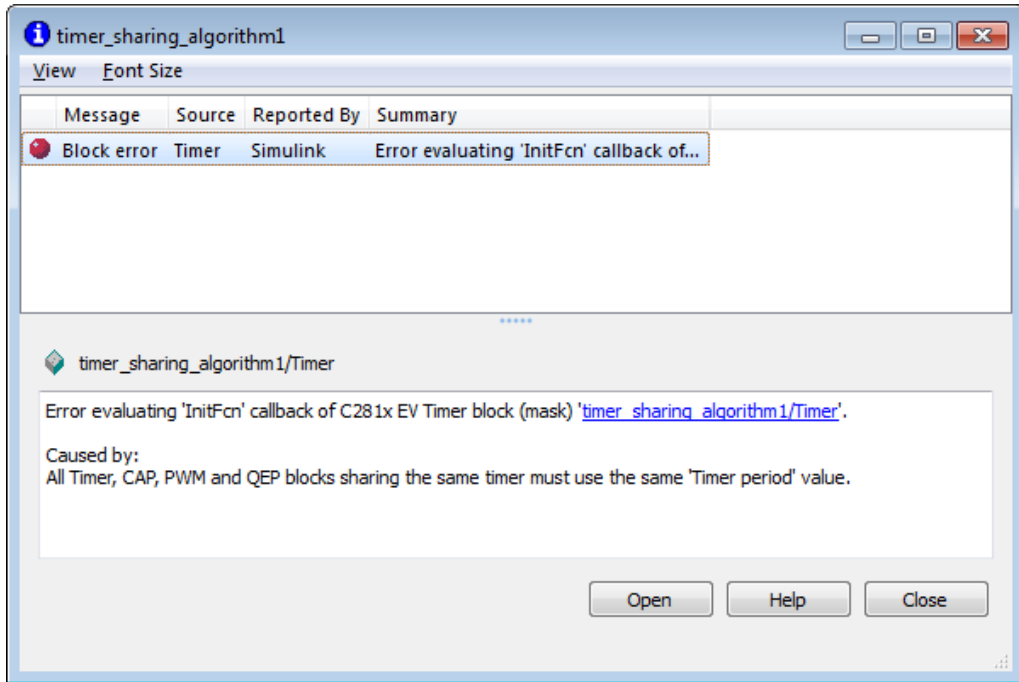
Post interrupt on overflow

Post interrupt on compare match

OK Cancel Help Apply

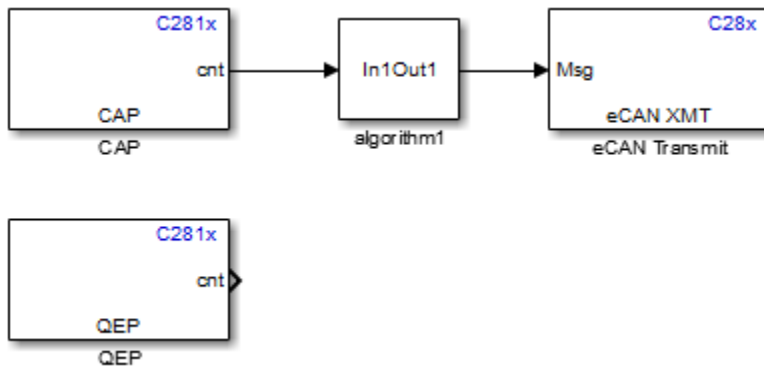


Both blocks have the same values for **Timer prescaler** and **Counting mode**. However, each block has different values for **Timer period**. The value of **Timer period** for Timer 1 is 65535 in the CAP block and 10000 in the Timer block.

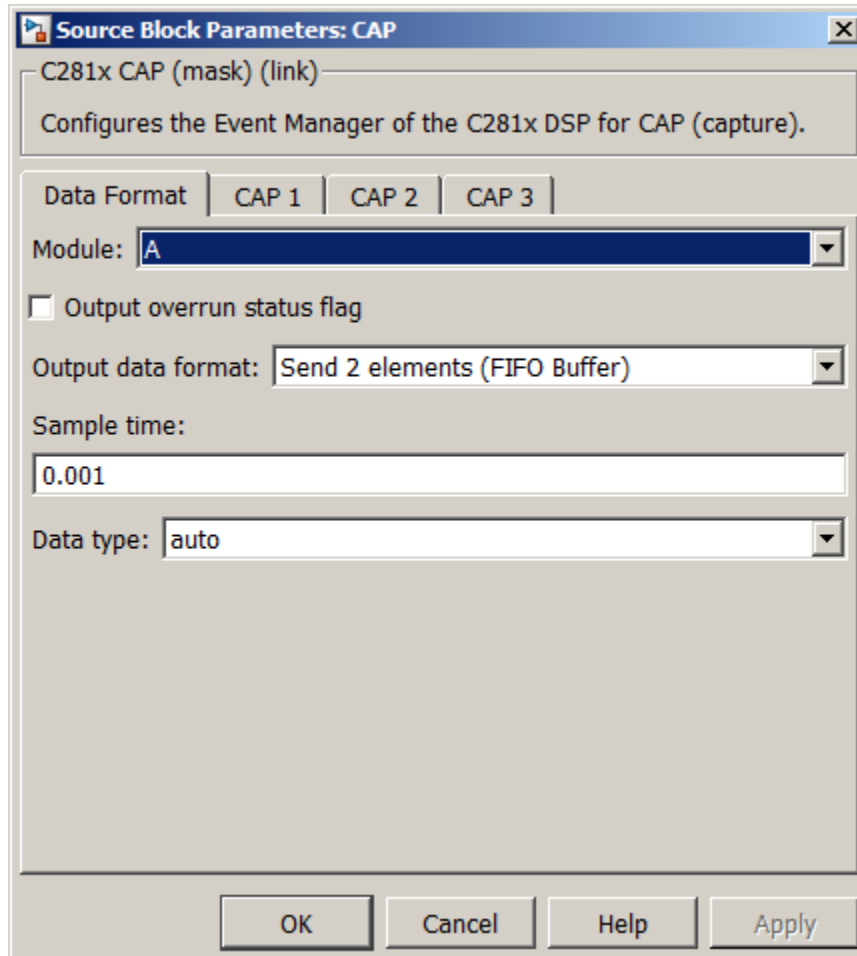


Since both blocks configure the same timer, and settings conflict, the software generates an error when you update the model.

Sharing General Purpose Timers between CAP and SPI



The model contains QEP and CAP blocks that both use Timer 2. In the CAP block, the **Time base** option shows which timer the block uses. In the QEP block, setting **Module** to A configures the block to use QEP1–QEP2. GP Timer Use for C281x Peripheral Blocks shows that QEP1–QEP2 use Timer 2.



Source Block Parameters: QEP

-C281x QEP (mask) (link)

Configures quadrature encoder pulse circuit associated with the selected Event Manager module to decode and count quadrature encoded pulses applied to related input pins (QEP1 and QEP2 for EVA or QEP3 and QEP4 for EVB). Depending on the selected counting mode, the output is either the pulse count or the rotor speed (when a pulse signal comes from an optical encoder mounted on a rotating machine).

Parameters

Module: **A**

Counting mode: **Counter**

Initial count :
0

Enable QEP index

Timer period:
65535

Sample time:
0.001

Data type: **auto**

OK Cancel Help Apply

Currently, both blocks define different clock sources for Timer 2. The CAP block uses **Internal** as a **Clock source**. The QEP block, which does not have a configurable **Clock source** setting, uses the QEP circuit as a clock source. If you build the model, the software generates the following error message.



To avoid generating errors when you build the model, change **Clock source** in the CAP block to QEP device.

Overview of Creating Models for C2000 Processors

In this section...

“Accessing the Embedded Coder Block Library” on page 41-21

“Building Your Model” on page 41-21

Accessing the Embedded Coder Block Library

After you have installed the supported development board, start MATLAB.

You can open the `c2000lib` blockset in the Simulink library browser, or by typing the following command at the MATLAB command prompt:

```
c2000lib
```

Create your real-time model for your application the same way you create other Simulink models. Select blocks to build your model from the following sources or products:

- The libraries in the `c2000lib` block library (for handling input and output functions for on your target hardware)
- Simulink Coder software
- Discrete time blocks from Simulink
- Another blockset that meets your needs and operates in the discrete time domain

Building Your Model

With this configuration, you can generate a real-time executable and download it to your TI development board. Simulink Coder software automatically generates C code and inserts the I/O device drivers as specified by the hardware blocks in your block diagram. These device drivers are inserted in the generated C code.

During the same build operation, block parameter dialog entries are combined into a project file for CCS for your TI C2000 board. If you selected the **Build and execute** build action in the configuration settings, the TI cross-compiler builds an executable file. After automatically downloading the executable file to the target, the build process runs the file on the board's DSP.

Note: After using the run-time `Build` option to generate and build code for your application, you must perform the following reset sequence before you can run that code on your board. If you want to rerun your application manually once it has been generated, you must also use this procedure.

F2812, F2808, and F28335 eZdsp Reset Sequence

- 1 Reset the board CPU.
- 2 Load your code onto the target hardware.
- 3 Run your code on the target hardware.

Using the c2000lib Blockset

In this section...

- “Introduction” on page 41-23
- “Hardware Setup” on page 41-23
- “Starting the c2000lib Library” on page 41-24
- “Setting Up the Model” on page 41-24
- “Adding Blocks to the Model” on page 41-25
- “Generating Code from the Model” on page 41-26

Introduction

This section uses an example to show how to create a Simulink model that uses Embedded Coder blocks to target your board. The example creates a model that performs PWM duty cycle control via pulse width change. It uses the C2812 ADC block to sample an analog voltage and the C2812 PWM block to generate a pulse waveform. The analog voltage controls the duty cycle of the PWM and you can observe the duty cycle change on the oscilloscope. This model is also provided in the Examples library. The model in the Examples library also includes a model simulation.

Hardware Setup

The following hardware is required for this example:

- Spectrum Digital eZdsp F2812
- Function generator
- Oscilloscope and probes

To connect the hardware:

- 1** Connect the function generator output to the ADC input ADCINA0 on the eZdsp F2812.
- 2** Connect the output of PWM1 on the eZdsp F2812 to the analog input of the oscilloscope.
- 3** Connect VREFLO to AGND on the eZdsp F2812. See the section on the Analog Interface in Chapter 2 of the *eZdsp™ F2812 Technical Reference*, available from the Spectrum Digital Web site at <http://c2000.spectrumdigital.com/ezf2812/>

Starting the c2000lib Library

At the MATLAB prompt, type the following command:

```
c2000lib
```

This command opens the `c2000lib` library blockset, which contains libraries of blocks designed for targeting your board.

Setting Up the Model

Preliminary tasks for setting up a new model include configuring the Target Hardware Resources and setting the simulation parameters.

- 1 Configure the Target Hardware Resources for `idmlink_ert.tlc` System Target file, as described in “Code Generation: Target Hardware Resources Pane”. For `ert.tlc`, configure the Target Hardware Resources as described in .
- 2 The following settings are made, referenced in the table below by their locations in the **Simulation > Model Configuration Parameters** dialog:

Pane	Field	Setting
Solver	Stop time	10
Solver	Type	Fixed-step
Data Import/Export	Save to workspace - Time	tout
Data Import/Export	Save to workspace - Output	yout
Hardware Implementation	Device type	C2000
Code Generation	Target selection - System target file	idmlink_grt.tlc, idmlink_ert.tlc , or ert.tlc

Note: Generated code does not honor Simulink stop time from the simulation. Stop time is interpreted as `inf`. To implement a stop in generated code, you must put a Stop Simulation block in your model.

- 3 From your model's main menu, select **Simulation > Model Configuration Parameters** to verify and set the simulation parameters for this model. Parameters you set in this dialog belong to the model you are building. They are saved with

the model and stored in the model file. Refer to your Simulink documentation for information on the Configuration Parameters dialog.

- 4 Use the **Code Generation** pane to set options for the real-time model.
- 5 Use the **Browse** button to locate and select a target configuration file, `idelink_grt.tlc`, `idelink_ert.tlc`, or `ert.tlc`. When you select `idelink_grt.tlc` or `idelink_ert.tlc`, your coder product chooses the system target file, and make command.

When you select `ert.tlc`, the **Target hardware** parameter and the **Toolchain** parameter are displayed. Select TI 281x as **Target hardware** and Texas Instruments Code Composer Studio v5 (c2000) for **Toolchain**.

Adding Blocks to the Model

- 1 Open or double-click the C281x library, `c281xlib`.
- 2 Drag the ADC block into your model. Double-click the ADC block in the model and set **Sample time** to 64/80000.
- 3 Drag the PWM block into your model. Double-click the PWM block in the model and set the following parameters.

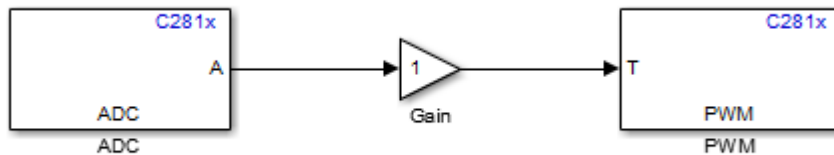
Pane	Field	Parameter
Timer	Module	A
	Waveform period source	Specify via dialog
	Waveform period units	Clock cycles
	Waveform period	64000
	Waveform type	Asymmetric
Outputs	Enable PWM1/PWM2	Selected
	Duty cycle source	Input port
Logic	PWM1 control logic	Active high
	PWM2 control logic	Active low
Deadband	Use deadband for PWM1/PWM2	Selected

Pane	Field	Parameter
	Deadband prescaler	16
	Deadband period	12
ADC Control	ADC start event	Period interrupt

- Enter Simulink in the MATLAB Command Window to open the Simulink Library browser. Drag a Gain block from the Math Operations library into your model. Double-click the Gain block in the model and set the following parameters in the Function Block Parameters dialog. Click **OK**.

Pane	Field	Parameter
Main	Gain	30
	Multiplication	Element-wise (K.*u)
	Sample time	-1
Signal Attributes	Output data type mode	uint(16)
	Integer rounding mode	Floor
Parameter Attributes	Parameter data type mode	Inherit from input mode

- Connect the ADC block to the Gain block and the Gain block to the PWM block.

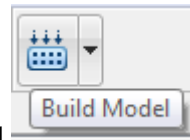


Generating Code from the Model

This section summarizes how to generate code from your real-time model.

There are three ways to start the automatic code generation process:

-



In the Simulink Editor, click **Build Model**.

- With your model open and in focus, press **Ctrl+b**.
- Press the Build button located on the **Code Generation** pane of the Configuration Parameters dialog.

Note: For `idelink_ert.tlc` with CCSv3, you see your project with the files in place in the folder structure.

Configuring Timing Parameters for CAN Blocks

In this section...
“The CAN Blocks” on page 41-28
“Setting Timing Parameters” on page 41-28
“Parameter Tuning and Signal Logging” on page 41-33

The CAN Blocks

The bit rate of these four CAN blocks cannot be set directly:

C281x eCAN Receive

C281x eCAN Transmit

C280x/C28x3x eCAN Receive

C280x/C28x3x eCAN Transmit

Setting Timing Parameters

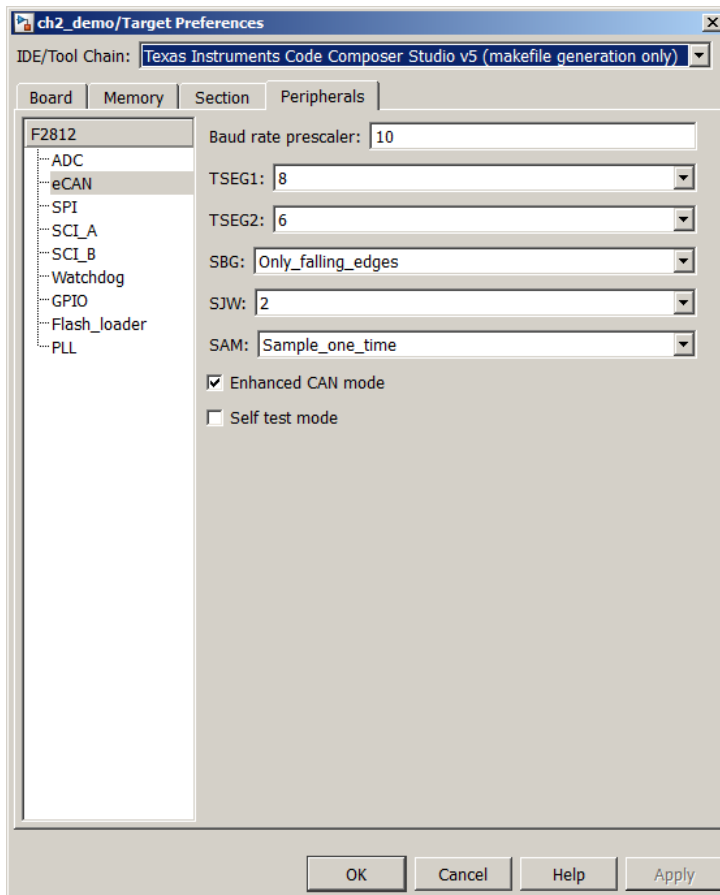
- “Accessing the Timing Parameters” on page 41-28
- “Determining Timing Parameter Values” on page 41-31
- “Working with CAN Bit Timing” on page 41-32

Accessing the Timing Parameters

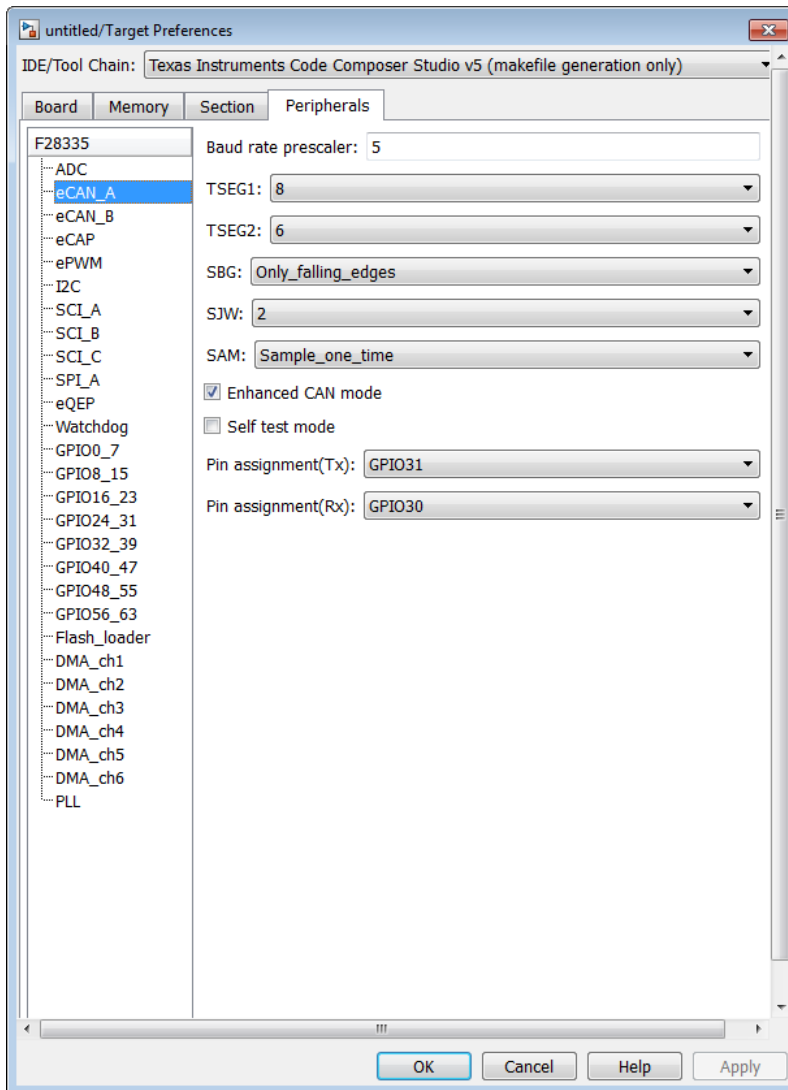
To set the Bitrate for a block whose bitrate cannot be set directly:

- 1 Configure the Target Hardware Resources tab, as described in “Configure Target Hardware Resources”.
- 2 Under the **Peripherals** tab, use the **TSEG1**, **TSEG2**, and **BaudRatePrescaler (BRP)** parameters to set the bitrate.

For example, the Target Hardware Resources tab for the F2812 eZdsp, this dialog is shown in the following figure.



The C280x/C28x3x blocks have two independent eCAN modules.



The following sections describe the series of steps and rules that govern the process of setting these timing parameters.

Determining Timing Parameter Values

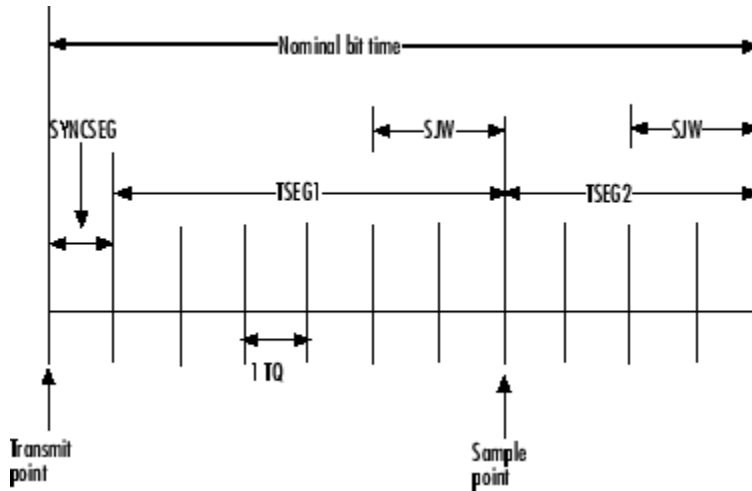
To determine the values for the timing parameters, complete the following steps:

- 1 Determine the CAN Baudrate specification based on your application.
- 2 Determine the frequency of the **CAN module clock**. For example:
 - CAN module clock = 100 MHz for the F2808 (Same as SYSCLKOUT)
 - CAN module clock = 150 MHz for the F2812 (Same as SYSCLKOUT)
 - CAN module clock = 75 MHz for the F28x3x (150 SYSCLKOUT/2)
- 3 Estimate the value of the **BaudRatePrescaler (BRP)**.
- 4 Solve this equation for BitTime:

$$\text{BitTime} = \text{CAN module clock frequency} / (\text{BRP} * \text{Baudrate})$$
- 5 Solve this equation for Baudrate:

$$\text{Baudrate} = \text{CAN module clock frequency} / (\text{BRP} * \text{BitTime})$$
- 6 Estimate values for **TSEG1** and **TSEG2** that satisfy $\text{BitTime} = \text{TSEG1} + \text{TSEG2} + 1$.
- 7 Use the following rules to determine the values of **TSEG1** and **TSEG2**:
 - TSEG1 >= TSEG2**
 - $\text{IPT (Information Processing Time)} = 3 / \text{BRP}$
 - $\text{IPT} \leq \text{TSEG1} \leq 16 \text{ TQ}$
 - $\text{IPT} \leq \text{TSEG2} \leq 8 \text{ TQ}$
 - $1 \text{ TQ} \leq \text{SJW} \leq \min(4 \text{ TQ}, \text{TSEG2})$
 where IPT is Information Processing Time, TQ is Time Quanta, and SJW is Synchronization Jump Width, also set in the Target Hardware Resources dialog.
- 8 Iterate steps 4 through 7 until the values selected for TSEG1, TSEG2, and BRP meet the criteria.

The following illustration shows the relationship between the eCAN bit timing parameters.



Working with CAN Bit Timing

Assume that CAN Module Clock Frequency = 75 MHz, and a Bitrate of 1 Mbits/s is required.

- 1 Set the BRP to 5. Then substitute the values of CAN Module Clock Frequency, BRP, and Bitrate into the following equation, solving for BitTime:

$$\text{BitTime} = \text{CAN Module Clock Frequency} / (\text{BRP} * \text{Bitrate})$$

$$\text{BitTime} = 75\text{e}6 / (5 * 1\text{e}6) = 15\text{TQ}$$

- 2 Set the values of **TSEG1** and **TSEG2** to 8TQ and 6TQ respectively. Substitute the values of *BitTime* from the previous equation, and the chosen values for *TSEG1* and **TSEG2** into the following equation:

$$\text{BitTime} = \text{TSEG1} + \text{TSEG2} + 1$$

$$15\text{TQ} = 8\text{TQ} + 6\text{TQ} + 1$$

- 3 Finally, check the selected values against the rules:

$$\text{IPT} = 3/\text{BRP} = 3/10 = .3$$

$$\text{IPT} \leq \text{TSEG1} \leq 16 \text{ TQ} \text{ True! } .3 \leq 8\text{TQ} \leq 16\text{TQ}$$

$$\text{IPT} \leq \text{TSEG2} \leq 8\text{TQ} \text{ True! } .3 \leq 6\text{TQ} \leq 8\text{TQ}$$

$$1\text{TQ} \leq \text{SJW} \leq \min(4\text{TQ}, \text{TSEG2}) \text{ which means that SJW can be set to either 2, 3, or 4}$$

4 When the chosen values satisfy the criteria, so further iteration is not required.

The following table provides example values for several bit rates when CAN Module Clock Frequency = 75 MHz, as it is with the F28335. Other combinations of the register values are possible.

Bitrate	TSEG1	TSEG2	Bit Time	BRP	SJW
0.25 Mbit/s	8	6	15	20	2
0.5 Mbit/s	8	6	15	10	2
1 Mbit/s	8	6	15	5	2

The following table provides example values for several bit rates when CAN Module Clock Frequency = 100 MHz, as it is with the F2808. Other combinations of the register values are possible.

Bitrate	TSEG1	TSEG2	Bit Time	BRP	SJW
0.25 Mbit/s	6	3	10	40	2
0.5 Mbit/s	5	4	10	20	2
1 Mbit/s	6	3	10	10	2

The following table provides example values for several bit rates when CAN Module Clock Frequency = 150 MHz, as it is with the F2812. Other combinations of the register values are possible.

Bitrate	TSEG1	TSEG2	Bit Time	BRP	SJW
0.25 Mbit/s	8	6	10	40	2
0.5 Mbit/s	7	7	10	20	2
1 Mbit/s	8	6	10	10	2

Parameter Tuning and Signal Logging

- “Overview” on page 41-34
- “Using External Mode” on page 41-34
- “Using a Third Party Calibration Tool” on page 41-42

Overview

Embedded Coder software supports parameter tuning and signal logging either using Simulink External mode or with a third party calibration tool. In both cases the model must include a CAN Calibration Protocol block.

Using External Mode

The Simulink External mode feature enables you to log signals and tune parameters without requiring a calibration tool. This section describes the steps for converting a model to use External mode.

External mode is supported using the CAN Calibration Protocol block and ASAP2 interface. The CAN Calibration Protocol block is used to communicate with the target, download parameter updates, and upload signal information. The ASAP2 interface is used to get information about where in the target memory a parameter or signal lives.

Note: You must configure the host-side CAN application channel. See “Configuring the Host Vector CAN Application Channel” on page 41-35.

To prepare your model for External mode, follow these steps:

- 1 Add a CCP driver block.
- 2 Identify signals you want to tune, and associate them with `Simulink.Parameter` or `canlib.Parameter` objects with `ExportedGlobal` storage class. It is important to set the data type and value of the object. See “Using Supported Objects and Data Types” on page 41-36.
- 3 Identify signals you want to log, and associate them with `canlib.Signal` objects. It is important to set the data type of the `canlib.Signal`. See “Using Supported Objects and Data Types” on page 41-36.

For information about visualizing logged signal data, see “Viewing and Storing Signal Data” on page 41-37.

- 4 Load the `Simulink.Parameter` or `canlib.Parameter` and `canlib.Signal` data objects into the base workspace.
- 5 Configure the model for building, as described in “Manual Configuration For External Mode” on page 41-40.
- 6 Build the model, and download the executable to the target

- 7 After downloading the executable to the target, you can switch the model to External mode, as described in “Manual Configuration For External Mode” on page 41-40.
- 8 You can now connect to the target using External mode by clicking the **Connect** button.
- 9 If you have set up tunable parameters, you can now tune them. See “Tuning Parameters” on page 41-37.

See the following topics for more information:

- “Configuring the Host Vector CAN Application Channel” on page 41-35
- “Using Supported Objects and Data Types” on page 41-36
- “Tuning Parameters” on page 41-37
- “Viewing and Storing Signal Data” on page 41-37
- “Manual Configuration For External Mode” on page 41-40
- “Limitations” on page 41-41

Configuring the Host Vector CAN Application Channel

External mode expects that the host-side CAN connection is using the 'MATLAB 1' application channel. To configure the application channel used by the Vector CAN drivers, enter the following in the MATLAB Command Window:

```
TargetsComms_VectorApplicationChannel.configureApplicationChannels
```

The Vector CAN Configuration tool appears. Use this tool to configure your host-side CAN channel settings.

If you try to connect using an application channel other than 'MATLAB 1', then you see the following warning in the command window:

```
Warning:
It was not possible to connect to the target using CCP.
An error occurred when issuing the CONNECT command.
```

If you have not already installed the Vector CAN drivers, you will get the following error message:

```
??? Error using ==>
TargetsComms_VectorApplicationChannel.TargetsComms_VectorApplicationChannel>
TargetsComms_VectorApplicationChannel.configureApplicationChannels at 40
Unable to launch the application channel configuration utility.
The "vcancnf" utility was not found on the Windows System Path.
To fix this error, make sure the required CAN drivers are installed on this computer;
refer to the product documentation for details.
```

If you want to use CAN to transmit or receive CAN messages between your host PC and your target, you require Vector-Informatik CAN hardware supported by the Vector CAN Driver Library. Choose the driver libraries to support profiling, downloading, and External mode.

Note: For CANcaseXL, install *both* the Vector XL-driver library and Vector CAN Driver Library `vcand32.dll`.

For older CAN hardware, you must install the Vector CAN Driver Library `vcand32.dll`.

Make sure that the library, `vcand32.dll`, is placed in the Windows `system32` folder.

Using Supported Objects and Data Types

Supported objects:

- `Simulink.Parameter` or `canlib.Parameter` for parameter tuning
- `canlib.Signal` for signal logging

Supported data types:

- `uint8`, `int8`
- `uint16`, `int16`
- `uint32`, `int32`
- `single`

You need to define data objects for the signals and parameters of interest for ASAP 2 file generation. For ease of use, create a MATLAB file to define the data objects, so that you only have to set up the objects once.

To set up tunable parameters and signal logging:

- 1 Associate the parameters to be tuned with `Simulink.Parameter` or `canlib.Parameter` objects with `ExportedGlobal` storage class. It is important to set the data type and value of the parameter object. See the following code for an example of how to create such a `Simulink.Parameter` object for tuning:

```
stepSize = Simulink.Parameter;  
stepSize.DataType = 'uint8';  
stepSize.CoderInfo.StorageClass = 'ExportedGlobal';
```

```
stepSize.Value = 1;
```

- Associate the signals to be logged with `canlib.Signal` objects. It is important to set the data type of the `canlib.Signal`. The following code example shows how to declare such a `canlib.Signal` object for logging:

```
counter = canlib.Signal;
counter.DataType = 'uint8';
```

- Associate the data objects you defined in the MATLAB file with parameters or signals in the model. For the previous code examples, you could set the **Constant value** in a Source block to `stepSize`, and set a **Signal name** to `counter` in the Signal Properties dialog. Remember that `stepSize` and `counter` are data objects defined in the code.

Tuning Parameters

To tune a parameter, follow these steps:

- Set `dataobject.value` in the workspace while the model is running in External mode. For example, to tune the parameter `stepSize` (that is, to change its value) from 1 to 2, enter the following at the command line:

```
stepSize.value = 2
```

You see output similar to the following:

```
stepSize =

Simulink.Parameter (handle)
  CoderInfo: [1x1 Simulink.ParamCoderInfo]
  Description: ''
  DataType: 'uint8'
  Min: []
  Max: []
  DocUnits: ''
  Value: 2
  Complexity: 'real'
  Dimensions: [1 1]
```

- Return to your model, and update the model (press **Ctrl+D**) to apply the changed parameter.

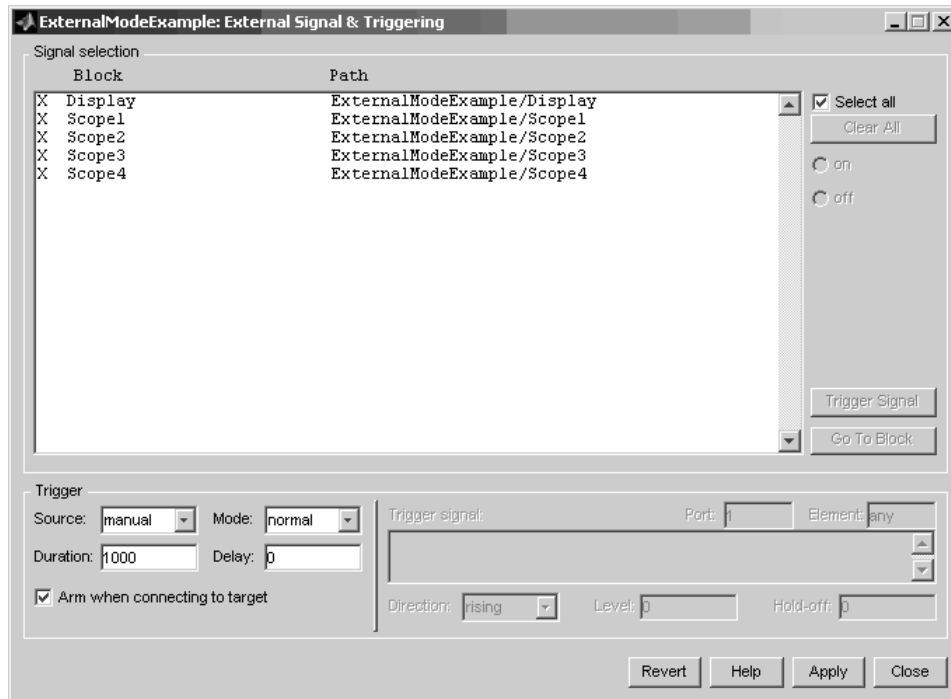
Viewing and Storing Signal Data

To view the logged signals, attach a supported scope type to the signal (see “Limitations” on page 41-41 for supported scope types).

By default, data from each scope is logged.

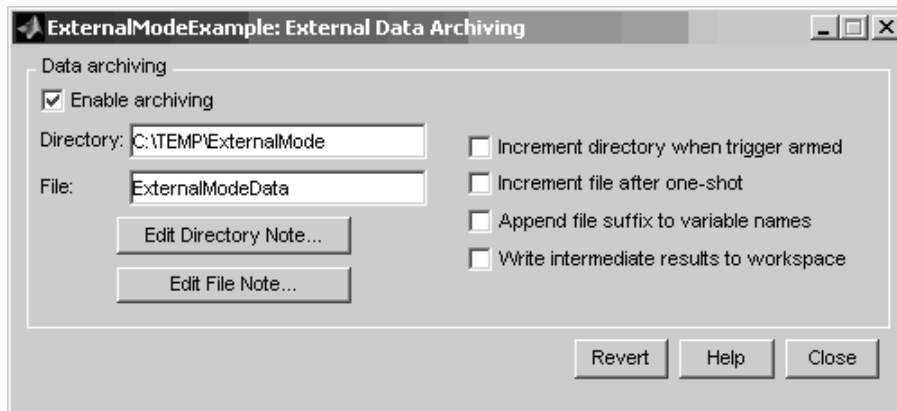
To customize which signals are logged:

- 1 In the Simulink Editor, select **Code > External Mode Control Panel**.
- 2 In External Mode Control Panel, click the **Signal & Triggering** button.
- 3 In External Signal & Triggering, clear the check boxes for scope data you do not want to log.

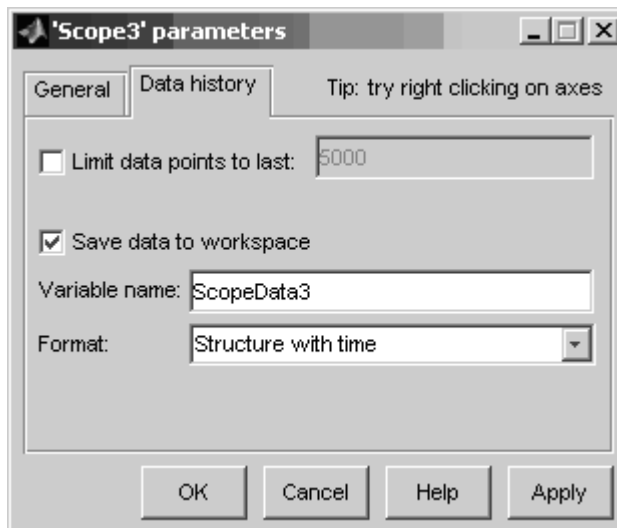


Storing signal data for further analysis. It is possible to store the logged data for further analysis in MATLAB.

- 1 To use the Data Archiving feature of External mode, click **Data Archiving** in the External Mode Control Panel. The External Data Archiving dialog appears.



- a Select the check box **Enable archiving**
 - b Edit the **Folder** and **Filename** and review the other settings.
 - c Close the dialog.
- 2 Open the Scope parameters, and select the check box **Save data to workspace**.



- 3 You may want to edit the **Variable name** in the edit box. The data that is displayed on the scope at the end of the External mode session is available in the workspace with this variable name.

The data that was previously displayed in the scope is stored in `.mat` files as previously setup using Data Archiving.

For example, at the end of an External mode session, the following variable and files could be available in the workspace and current folder:

- A variable `ScopeData5` with the data currently displayed on the scope:

```
ScopeData5
ScopeData5 =
    time: [56x1 double]
    signals: [1x1 struct]
    blockName: 'mpc755rt_ccp/Scope1'
```

- In the current folder, `.mat` files for the three previous **Durations** of scope data:

```
ExternalMode_0.mat
ExternalMode_2.mat
ExternalMode_1.mat
```

Manual Configuration For External Mode

You can configure models manually for build and execution with External mode.

To configure a model to be built for External mode:

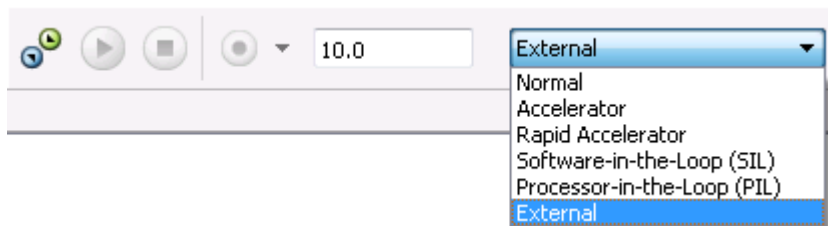
- 1 Select **Simulation > Model Configuration Parameters**.
- 2 In the Configuration Parameters dialog, open the **Optimization > Signals and Parameters** pane.
- 3 Select the **Inline parameters** check box and click the **Configure** button.
- 4 In the Model Parameter Configuration dialog that opens, define the (global) tunable parameters for your models.
- 5 Select **Simulation > Mode > Normal**. Or, set Simulation mode as shown here.



- 6 In the Configuration Parameters dialog, open the **Code Generation > Interface** pane.
- 7 Set the **Interface** parameter to **ASAP2**.

After you build the model, you can configure it for External mode execution:

- 1 Select **Simulation > Mode > External**.



- 2 In the Configuration Parameters dialog, open the **Code Generation > Interface** pane.
- 3 Set the **Interface** parameter to **External mode**.

Limitations

Multiple signal sinks (e.g. scopes) are not supported.

Only the following kinds of scopes are supported with External mode Logging:

- Simulink Scope block
- Simulink Display block
- Viewer type: scope — To use this option, right-click a signal in the model, and select **Create & Connect Viewer > Simulink > Scope**. The other scope types listed there are not supported (e.g., floating scope).

Before connecting to External mode, you also need to right-click the signal, and select **Signal Properties**. In the dialog, select the **Test point** check box, and click **OK**.

GRT is supported but only for parameter tuning.

It is not possible to log signals with sample rates in excess of 10 kHz.

Subsystem builds are not supported for External mode, only top-level builds are supported.

Logging and tuning of nonscalars is not supported. It is possible to log nonscalar signals by breaking the signal down into its scalar components. For an example of how to do this signal deconstruction, see the CCP example models, which use a Demux and Signal Conversion block with contiguous copy.

Logging and tuning of complex numbers is not supported. It is possible to work with complex numbers by breaking the complex number down into its real and imaginary components. This breakdown can be performed using the following blocks in the Simulink Math Operations library: Complex to Real-Imag, Real-Imag to Complex, Magnitude-Angle to Complex, Complex to Magnitude-Angle.

Using a Third Party Calibration Tool

Embedded Coder allows an ASAP2 data definition file to be generated during the code generation process. This file can be used by a third party tool to access data from the real-time application while it is executing.

ASAP2 is a data definition standard by the Association for Standardization of Automation and Measuring Systems (ASAM). ASAP2 is a standard description for data measurement, calibration, and diagnostic systems. Embedded Coder software lets you export an ASAP2 file containing information about your model during the code generation process.

Before you begin generating ASAP2 files with Embedded Coder software, see “Generating an ASAP2 File” in the product help for Simulink Coder. That section describes how to define the signal and parameter information required by the ASAP2 file generation process.

Select the ASAP2 option before the build process as follows:

- 1 Select **Simulation > Model Configuration Parameters**.

The Configuration Parameters dialog appears.

- 2 Select the Code Generation > Interface pane.
- 3 Select the ASAP2 option from the **Interface** drop-down menu, in the **Data exchange** frame.
- 4 Click **Apply**.

The build process creates an ASAM-compliant ASAP2 data definition file for the generated C code.

- The standard ASAP2 file generation does not include the memory address attributes in the generated file. Instead, it leaves a placeholder that must be replaced with the actual address by postprocessing the generated file.
- The map file options in the template project need to be set up a certain way for this procedure to work. If you have created your own template projects, and you do not have the right settings, you see the following instructions:

```
Warning: It was not possible to do ASAP2 processing on your
.map file.This is because your IDE project template is not
configured to generate a .map file in the correct format.
To generate a .map file in the correct format you need to
setup the following options in your IDE project template:
Generate section map should be checked on
Generate register map should be checked off
Generate symbol table should be checked on
Format list file into pages should be checked off
Generate summary should be checked off
Page width should be equal to 132 characters
Symbol columns should be 1
You can change these options via Project -> Project Options
-> Linker/Locator -> Map File -> Map File Format.
```

Embedded Coder software performs this postprocessing for you. To do this, it first extracts the memory address information from the map file generated during the link process. Secondly, it replaces the placeholders in the ASAP2 file with the actual memory addresses. This postprocessing is performed automatically and does not require additional input from you.

ADC-PWM Synchronization via ADC Interrupt

This example shows how to use the ADC and PWM blocks. In the generated code, changes in the voltage of the ADC input alter the duty cycle of the PWM output. The period of the PWM waveform remains constant. This example also shows how to use the Hardware Interrupt block to synchronize the update of the PWM duty cycle with the ADC conversion.

Required Hardware:

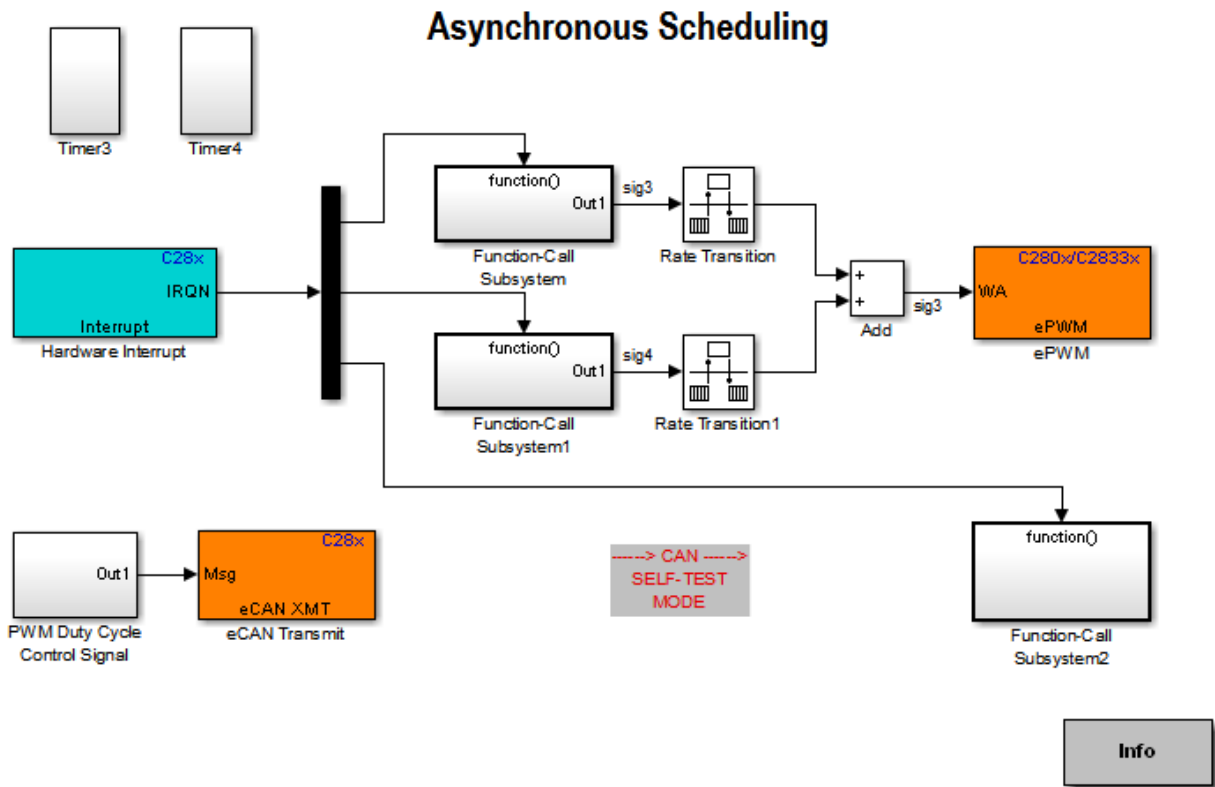
- Spectrum Digital F2808/F2812/F28335 eZdsp
- Texas Instruments F2808/F28027/F28035/F28044/F28069/F28335/F28343/F28346 ControlCARD with docking station
- Texas Instruments F28027/F28069 ControlSTICK
- Oscilloscope and probes
- Function generator

The following models are available for this example:

- F281x based board: `c281x_adcpwmasynctest`
- F280x/F2823x/F2833x based board: `c280x_2833x_adcpwmasynctest`
- Piccolo F2802x/F2803x/F2806x based board: `c280xx_adcpwmasynctest`

To open the model, enter the model name in the MATLAB Command Window.

The following figure shows the example model.



Copyright 2007-2013 The MathWorks, Inc.

This model shows how to use the ADC block to sample an analog voltage and how to use the PWM block to generate a pulse waveform. The analog voltage controls the duty cycle of the PWM waveform. Duty cycle changes can be observed on the oscilloscope. "Hardware Interrupt" installs an Interrupt Service Routine (ISR) for ADC interrupt and schedules the execution of the connected subsystem (ADC-PWM Subsystem) when ADC interrupt (ADCINT) is received.

"ADC-PWM Subsystem" consists of an ADC driving the duty cycle input port of the PWM. PWM is configured to trigger ADC start of conversion (SOC).

Before using the model, connect the function generator output to the ADC input ADCINA0 on the board. Connect the output of the PWM1 to the analog input of the oscilloscope.

Run the model using CCSv3

- Open the model corresponding to the family of processor you are using.
- Each model is configured for a default processor. Open the Target Hardware Resources tab to select a different processor.
- Click **Incremental build** or press **Ctrl+b** to generate, build, load and run the program.
- Observe the change of the PWM waveform on the oscilloscope.

Run the model using CCSv4

CCSv4 support is provided via a makefile based approach. This requires running `xmakefilesetup` to set the environment. You can ignore the following if you have already gone through these steps. For more information, see “Using Makefiles with Code Composer Studio 4/5” on page 40-5.

- Enter `xmakefilesetup` in a MATLAB Command Window.
- In the XMakefile User Configuration dialog that opens, set the **Template** parameter to `gmake`.
- Clear the **Display operational configuration only** check box.
- Set the **Configuration** parameter to `ticcs_c2000_ccsv4`, click **Apply**.
- If you haven't used this configuration before, a dialog will ask you to browse to your CCSv4 installation folder. Pick the right folder and click **OK**. Click the Tool Directories and make sure that the paths are pointing to directories matching your installation. For example:
 - **CCS Installation:** `C:\Program Files\Texas Instruments\ccsv4\`
 - **Code Generation Tools:** `C:\Program Files\Texas Instruments\ccsv4\tools\compiler\C2000\`
 - **DSP/BIOS Installation:** `C:\Program Files\Texas Instruments\bios_5_41_10_36\`

Code Generation and Creation of an .out file using CCSv4:

- Open the model that corresponds to the processor family you are using.
- Open the Target Hardware Resources tab, as described in “Configure Target Hardware Resources”, and set the **IDE/Tool chain** parameter to **Texas Instruments Code Composer Studio v4 (makefile generation only)**.

- Change the selected board and processor to match your hardware.
- Click **OK** to save the changes on the Target Hardware Resources tab.
- Click **Incremental build** or press **Ctrl+b** to generate and build an .out executable file.
- Using CCSv4, connect to your board, load and run the .out file.
- Observe the PWM waveform on the oscilloscope.

Run the model using CCSv5

CCSv5 support is provided via a makefile based approach. This requires running `xmakefilesetup` to set the environment. You can ignore the following if you have already gone through these steps. For more information, see “Using Makefiles with Code Composer Studio 4/5” on page 40-5.

- Enter `xmakefilesetup` in a MATLAB Command Window.
- In the XMakefile User Configuration dialog that opens, set the **Template** parameter to `gmake`.
- Clear the **Display operational configuration only** check box.
- Set the **Configuration** parameter to `ticcs_c2000_ccsv5`, click **Apply**.
- If you haven't used this configuration before, a dialog will ask you to browse to your CCSv5 installation folder. Pick the right folder and click **OK**. Click the Tool Directories and make sure that the paths are pointing to directories matching your installation. For example:
 - **CCS Installation:** `C:\ti\ccsv5\`
 - **Code Generation Tools:** `C:\ti\ccsv5\tools\compiler\c2000\`
 - **DSP/BIOS Installation:** `C:\ti\bios_5_41_11_38\`

Code Generation and Creation of an .out file using CCSv5:

- Open the model.
- Open the Target Hardware Resources tab, as described in “Configure Target Hardware Resources”, and set the **IDE/Tool chain** parameter to `Texas Instruments Code Composer Studio v5 (makefile generation only)`.
- Change the selected board and processor to match your hardware.
- Click **OK** to save the changes on the Target Hardware Resources tab.

- Click **Incremental build** or press **Ctrl+b** to generate and build an .out executable file.
- Using CCSv5, connect to your board, load and run the .out file.
- Observe the PWM waveform on the oscilloscope.

Configuring Acquisition Window Width for ADC Blocks

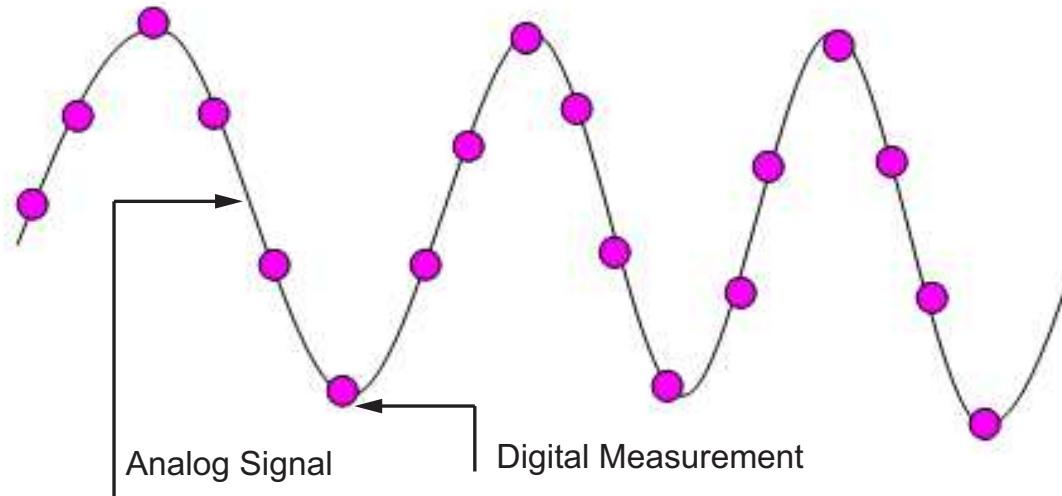
In this section...

“What Is an Acquisition Window?” on page 41-49

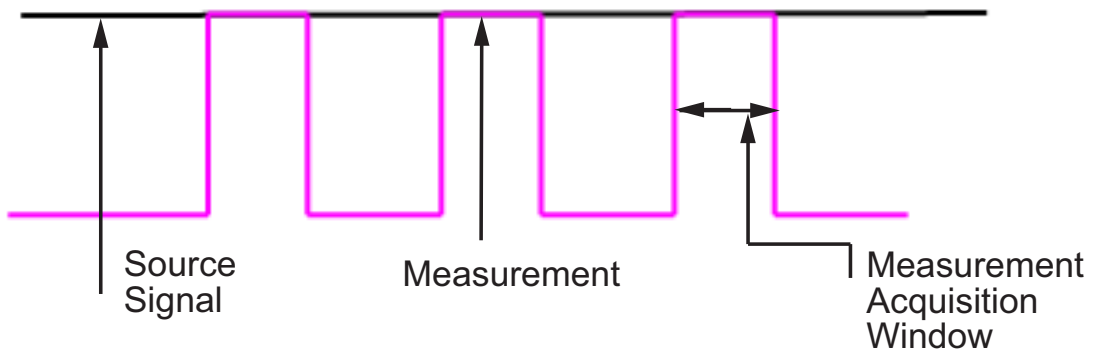
“Configuring ADC Parameters for Acquisition Window Width” on page 41-51

What Is an Acquisition Window?

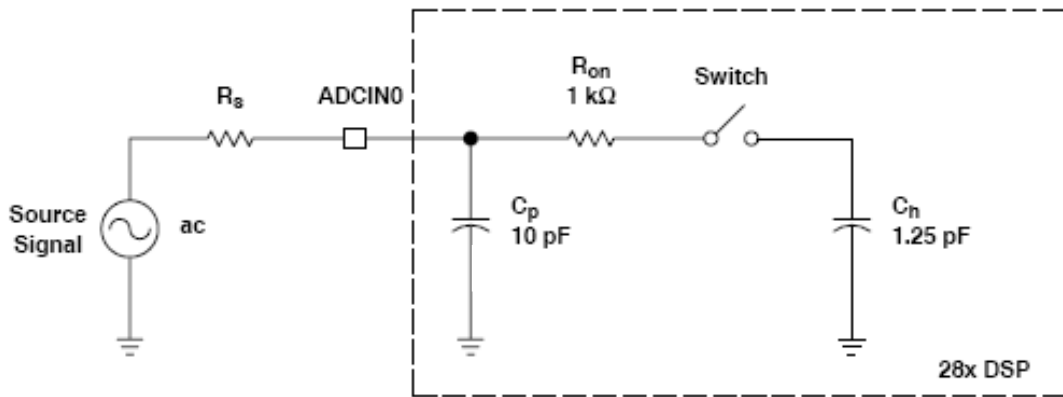
ADC blocks take a signal from an analog source and measure it with a digital device. The digital device does not measure in a continuous process, but in a series of discrete measurements, close enough together to approximate the source signal with the required accuracy.



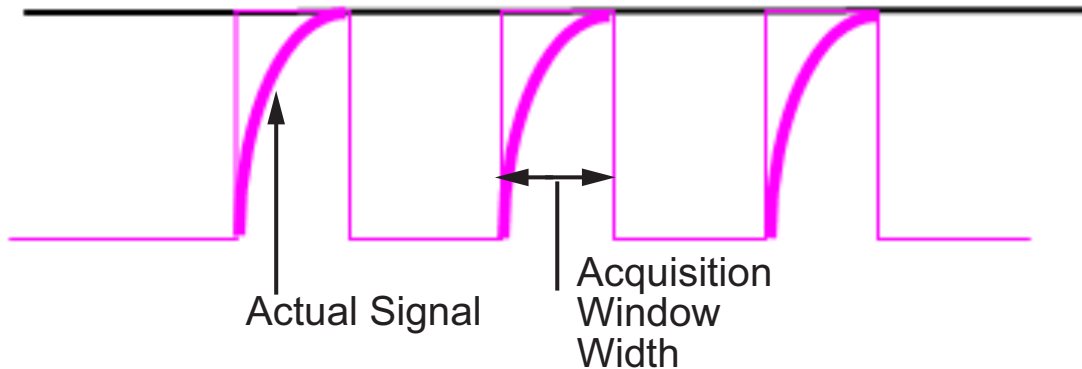
The digital measurement itself is not an instantaneous process, but is a measurement window, where the signal is acquired and measured, as shown in the following figure.



Ideally, as soon as the measurement window is opened, the actual signal coming in would be measured perfectly. In reality the signal does not reach its full magnitude immediately. The measurement process can be modeled by a circuit similar to the one shown in the following figure for the ADC found on the F2812 eZdsp



where the measurement circuit is characterized by a certain capacitance. In the preceding figure, when the switch is closed, the measurement begins. In this circuit, which is characterized by its capacitance, the signal received is not in a form of a step function as shown by the ideal measurement, but a ramp up to the true signal magnitude. The following figure shows what happens to the signal when the sampler switch is closed and the signal is received to be measured.



Because the signal acquisition is not instantaneous, it is very important to set a wide enough acquisition window to allow the signal to ramp up to full strength before the measurement is taken. If the window is too narrow, the measurement is taken before the signal has reached its full magnitude, resulting in erroneous data. If the window is too wide, the source signal itself may change, and the sampling may be too infrequent to reflect the actual value, also resulting in erroneous data. You must calculate the width of the acquisition window based on the circuit characteristics of resistance and capacitance of your specific circuit. Then, using the ADC parameters described in the following section, you can configure the acquisition window width.

Configuring ADC Parameters for Acquisition Window Width

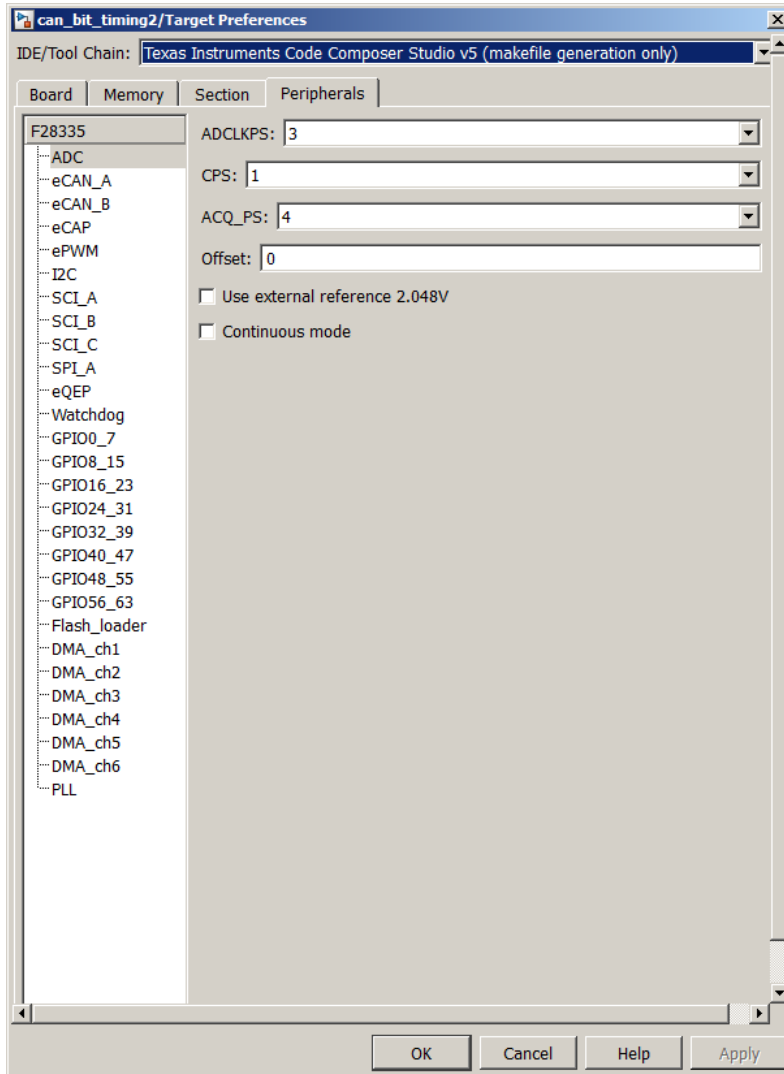
- “Accessing the ADC Parameters” on page 41-51
- “Configure Acquisition Window Width Using ADC Parameters” on page 41-53

Accessing the ADC Parameters

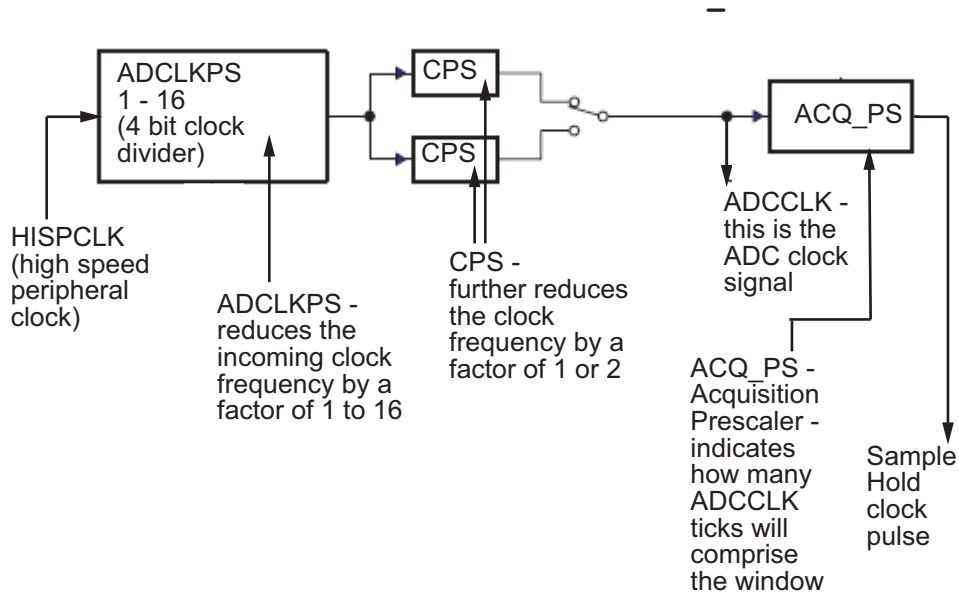
The ADC parameters can be set from the **Peripherals tab** of the Target Hardware Resources tab.

- You can set **ACQ_PS** — Acquisition Prescaler — to a value from 0 to 15. To obtain the actual value, increment the setting by 1. This produces an actual range from 1 to 16.
- You can set **ADCLKPS** — AD Clock Prescaler — to a value from 0 to 15. To obtain the actual value, increment the setting by 1. This produces an actual range from 1 to 16.

- You can set **CPS** — Clock Prescaler — to a value from 0 to 1. To obtain the actual value, increment the setting by 1. This produces an actual range from 1 to 2.



These three prescalers serve to reduce the speed of the clock and to set the acquisition window width. The following diagram shows how these prescalers are used.



In the preceding diagram, the high speed peripheral clock frequency is received and then divided by the **ADCLKPS**. The reduced clock frequency is then further divided by **CPS**. The resulting frequency is the **ADCCLK** signal. The value of **ACQ_PS** then determines how many **ADCCLK** ticks comprise one S/H (sample and hold) period, or in other words, the length of the acquisition window.

Configure Acquisition Window Width Using ADC Parameters

The following examples show how you can use ADC parameters to configure the acquisition window width:

Example 1:

If the HISPCLK = 30 MHz, and **ADCLKPS**=1 (which is a value of 2), the result is 15 MHz.

If **CPS**= 1 (which is a value of 2), then **ADCCLK** = 7.5 MHz.

If **ACQ_PS** = 0 (which is a value of 1), then the sample/hold period is 1 **ADCCLK** tick, or .1333 microseconds.

Example 2:

If the $HISPCLK = 30$ MHz, and $ADCLKPS=1$ (which is a value of 2), the result is 15 MHz.

If $CPS= 1$ (which is a value of 2), then $ADCCLK = 7.5$ MHz.

If $ACQ_PS = 15$ (which is a value of 16), then the sample/hold period is 16 $ADCCLK$ ticks, or 2.1333 microseconds.

Note: $HISPCLK$ is set automatically for the user, and it is not possible to change the rate. For more information, see “High-Speed Peripheral Clock” on page 41-8

Real-Time Data Exchange via RTDX™

This example shows how to use the **From RTDX** and **To RTDX** blocks to send data from host to target and receive data from target to host. The model implements a simple sine wave generator that runs on the F28335 board.

The Real-Time Data Exchange via RTDX supports only Code Composer Studio v3.3.

Required Hardware:

- Spectrum Digital F2808 eZdsp
- Spectrum Digital F2812 eZdsp
- Spectrum Digital F28044 eZdsp
- Spectrum Digital F28335 eZdsp
- Texas Instruments F2808/F28027/F28035/F28069/F28335/F28343/F28346 ControlCARD with docking station
- Texas Instruments F28027/F28069 ControlSTICK

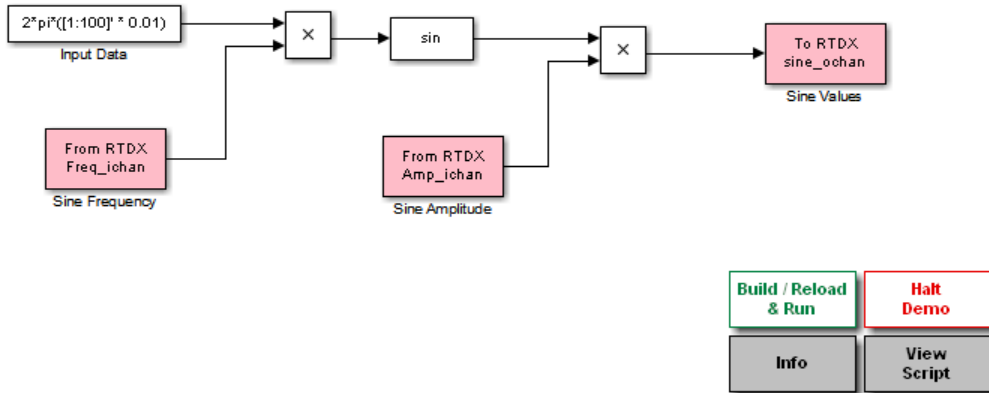
Available versions of this example:

This example can be run on different boards by changing the processor selection on the Target Hardware Resources tab of the model: `c28x_rtdxtest`

Model

The following figure shows the example model.

Real-Time Data Exchange via RTDX



Copyright 2005-2013 The MathWorks, Inc.

Description

The first step to run this example is to generate code for the model and to build it to run on the target. Once the model is running, we use a MATLAB script to read the sine wave values from the channel specified by the "To RTDX" block and plot them. To change the frequency and amplitude of the sine wave while it is running on the target, we use the same MATLAB script to write data to the channels specified by the "From RTDX" blocks. Note that the RTDX™ channel names specified in the RTDX blocks in the Simulink model are the same channel names used in the MATLAB script. Open the example script to see the specific MATLAB code used to establish RTDX communication.

How to Run the Example

- Open the model `e28x_rtdxtest`.
- The model is configured by default for the F28335 processor. Open the Target Hardware Resources tab to select a different processor.
- Click **OK** to save the changes on the Target Hardware Resources tab.
- Click **Build/Reload & Run** to generate, build, load and run the program.

SPI-Based Control of PWM Duty Cycle

This example shows how to the use of the SPI Receive, SPI Transmit, PWM, and Memory Copy blocks to generate a PWM waveform.

Required Hardware:

- Spectrum Digital F2808/F2812/F28335 eZdsp
- Texas Instruments F2808/F28027/F28035/F28044/F28069/F28335/F28343/F28346 ControlCARD with docking station
- Texas Instruments F28027/F28069 ControlSTICK
- Oscilloscope and probes

Available versions of this example:

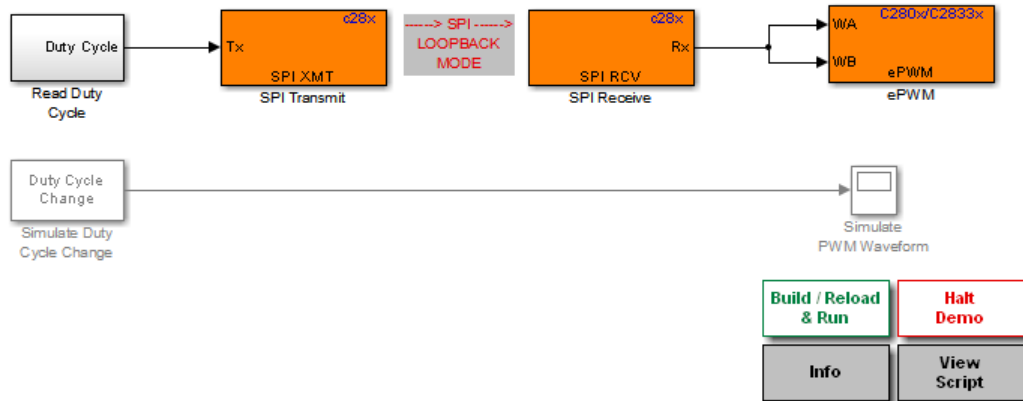
- **F281x** based board: c281x_spitest
- **F280x/F2823x/F2833x** based board: c280x_2833x_spitest
- **Piccolo F2802x/F2803x/F2806x** based board: c280xx_spitest

Note: This model only runs with **Code Composer Studio v3.3**

Model

The following figure shows the example model.

SPI-Based Control of PWM Duty Cycle



Copyright 2007-2013 The MathWorks, Inc.

Description

Memory Copy block labeled "Read Duty Cycle" reads duty cycle value stored at the memory location 0x8FF0. "C28xx SPI Transmit" block transmits this value, while "C28xx SPI Receive" receives it. The duty cycle value is then sent to the PWM1. The duty cycle of the generated pulse waveform is determined by the relative ratio of the received pulse width value and the pulse period which is fixed at 64000 clock cycles. The duty cycle toggles between 25%, 50%, and 75% based on user selection.

PWM Duty Cycle can be changed while the generated code is running using `write` function. This method writes selected value to the memory location 0x8FF0. Select either 25%, 50%, or 75% value from the GUI that opens up when "Build/Reload & Run" is pressed.

'**Enable loopback**' mode is checked for 'SPI_A' on the peripherals tab of the Target Hardware Resources tab. This is required to get the transmitted data back to the processor. If this mode is disabled, SPI_A transmitter and receiver have to be connected.

Note that "Simulate Duty Cycle Change" and "Simulate PWM Waveform" are dedicated to simulation only. Use these blocks to simulate and observe effects achieved with the generated code.

Hardware Connections

Connect the output of PWM1 on the board to the analog input of the oscilloscope.

Note that SPI module is set to work in "loopback" mode. External SPI hardware is not needed. As an alternative, the user may disable SPI loopback mode and bridge SPI transmit and receive pins on each of the following:

- F2812eZdsp (connector P8 pins 23 and 24)
- F2808 eZdsp (connector P8 pins 9 and 10)
- F28335 eZdsp connector P8 pins 9 and 10)
- Texas Instruments F28027/F28069 controlSTICK (connector J1 pins 20 and 24)
- Texas Instruments controlCARD with docking station (GPIO 16 and 17)

How to Run the Example in Simulation

- Open this model.
- Click "Start simulation" button.
- Double-click on the "Duty Cycle Change" block and select a new duty cycle value.
- Observe the PWM waveform change.

How to Run the Example on the Board

- Open the model corresponding to the family of processor you are using.
- Each model is configured for a default processor, open the Target Hardware Resources tab to select a different processor.
- Go to **Peripherals > SPI_A**. Check **Enable loopback**. If your processor is F2803x/F2806x, select **GPIO16** for **SOMI pin assignment** and **GPIO17** for **SIMO pin assignment**.
- Click **Build/Reload & Run** to generate, build, load and run the program.
- Select a new duty cycle value in the dialog that opens.
- Observe the PWM waveform on the oscilloscope.

HIL Verification of IIR Filter via SCI

This example shows how to perform HIL verification of an IIR filter using serial communication interface between a host simulation and generated code on a TI C28x processor.

Required hardware:

- Spectrum Digital F2808 eZdsp
- Spectrum Digital F2812 eZdsp
- Spectrum Digital F28044 eZdsp
- Spectrum Digital F28335 eZdsp

On the following boards the embedded xds100 JTAG can be used as a serial connection over USB. Jumper and switch settings with corresponding FTDI drivers are required to use this serial communication technique.

- Texas Instruments F2808/F28027/F28035/F28044/F28069/F28335/F28343/F28346 ControlCARD with docking station
- Texas Instruments F28027/F28069 ControlSTICK

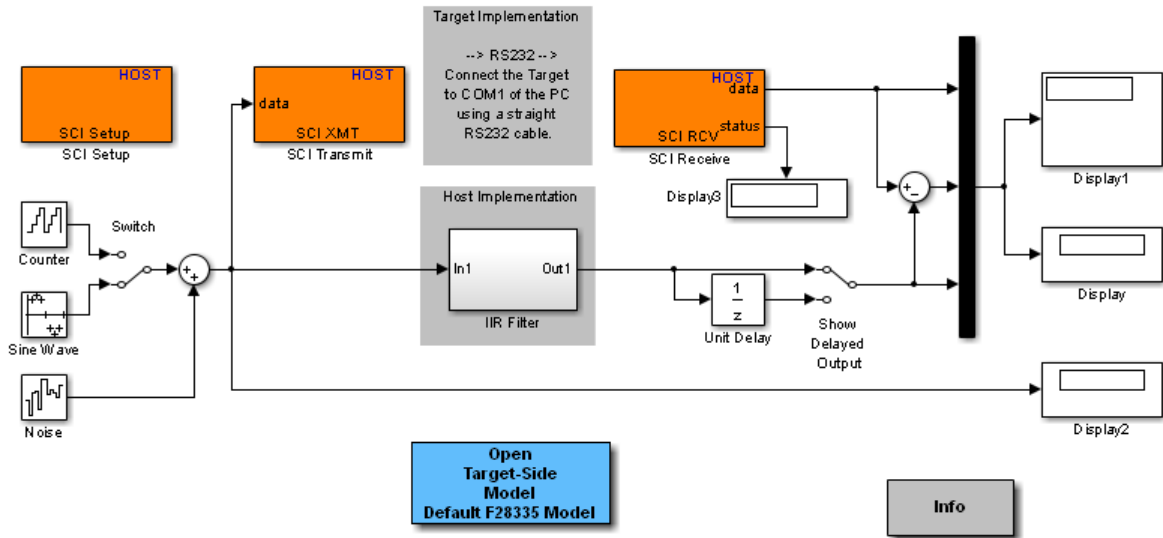
Available versions of this example:

This example model c2000scitest can be used on a variety of boards.

Host Model

The following figure shows the host model.

HIL Verification of IIR Filter via SCI

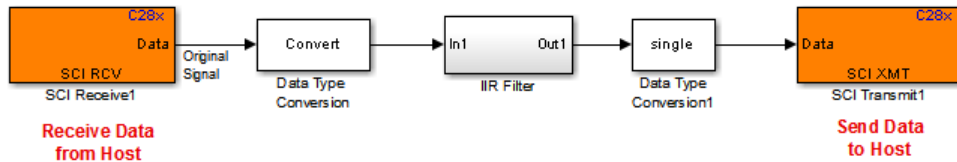


Copyright 2005-2012 The MathWorks, Inc.

Target Model

The following figure shows the target model.

HIL Verification of IIR Filter Using SCI



Copyright 2006-2013 The MathWorks, Inc.

Description

This example shows HIL verification of an IIR filter using serial communication interface between a host simulation and generated code on an c28x processor-based target board.

Based on the position of the manual switch, either a saw-tooth or sine waveform is passed to the Host-side SCI Transmit block and sent to the target. The stream of data is received from the target the via Host-side SCI Receive block and sent to the Simulink scope block.

The same algorithm runs on the target and the host side at the same time. The scope displays three signals showing the result of the simulation compared with the result of the algorithm running on the target. Because the algorithm uses an IIR filter, its output is therefore dependent on every sample time that occurs from the beginning of the simulation. Using an IIR filter clearly shows the time consistency between the simulation and the generated code.

You can add a delay in the simulation by using the "Show Delayed Output" switch. This delay can be used to verify the time consistency between both models.

Hardware Connections

- Connect the RS232 connector of your computer to the RS232 connector of the F28xx chip-based board.

- F2812 eZdsp is not equipped with RS232 transceivers. You must add RS232 transceivers to this board to run this example.
- ControlCARDS with Docking Stations, ControlSTICK boards can support serial over the xds100 USB JTAG connection. Jumper and switch settings with corresponding FTDI drivers are required to use this serial communication technique.

How to Run the Example

The following sections explain how to configure, build, and deploy the executable.

Note: The host model is set to use "COM1". Configure the SCI Setup, SCI Transmit, and SCI Receive blocks to use a different COM port.

With Code Composer Studio v3.3 (CCSv3.3) - Default

- Open the model c2000scitest
- Double-Click on "Open Target-Side Model Default F28335". This opens the target model.
- The model is configured by default for the F28335 processor. Open the Target Hardware Resources tab to select a different processor.
- Go to "Peripherals" > "SCI_A", and change the "Communication mode" to "Protocol".
- Click "OK" to save the changes on the Target Hardware Resources tab.
- Click "Build Model" button or press "Ctrl+B" to generate, build, load and run the program on the target.
- Click "Run" to simulate the model c2000scitest

Change the manual switch position to select a different input signal.

With Code Composer Studio v4 (CCSv4)

CCSv4 support is provided via a makefile based approach. This requires running xmakefilesetup to set the environment. You can ignore the following if you have already gone through these steps. For more information, please refer to the Xmakefile Documentation

- Run XMakefile User Configuration.
- Set the Template parameter to "gmake".
- Uncheck the "Display operational configuration only" checkbox.

- Set the Configuration parameter to "ticcs_c2000_ccsv4", click "Apply", if you haven't used this tool before, a dialog will ask you to browse to your CCSv4 installation folder. Pick the right folder and click "OK".
- Click on the "Tool Directories" and make sure that the paths are pointing to directories matching your installation. For example:
 - 1 *C:\Program Files\Texas Instruments\ccsv4* for "CCS Installation".
 - 2 *C:\Program Files\Texas Instruments\ccsv4\tools\compiler\C2000* for "Code Generation Tools".
 - 3 *C:\Program Files\Texas Instruments\bios_5_41_10_36* for "DSP/BIOS Installation".

Code Generation and Creation of an .out file using CCSv4:

- Open the model c2000scitest.
- Double-Click on "Open Target-Side Model Default F28335". This opens the target model.
- Open the Target Hardware Resources tab and set the **IDE/Tool chain** parameter to Texas Instruments Code Composer Studio v4 (makefile generation only).
- Change the selected board and processor to match your hardware.
- Go to "Peripherals" > "SCI_A", and change the "Communication mode" to "Protocol".
- Click "OK" to save the changes on the Target Hardware Resources tab.
- Click "Build Model" button or press "Ctrl+B" to generate and build an .out executable file.
- Using CCSv4, connect to the Hardware and load and run the .out file.
- Now simulate the model c2000scitest.

Change the manual switch position to select a different input signal.

With Code Composer Studio v5 (CCSv5)

CCSv5 support is provided via a makefile based approach. This requires running xmakefilesetup to set the environment. You can ignore the following if you have already gone through these steps. For more information, please refer to the Xmakefile Documentation

- Run XMakefile User Configuration.

- Set the Template parameter to "gmake".
- Uncheck the "Display operational configuration only" checkbox.
- Set the Configuration parameter to "ticcs_c2000_ccsv5", click "Apply", if you haven't used this tool before, a dialog will ask you to browse to your CCSv5 installation folder. Pick the right folder and click "OK".
- Click on the "Tool Directories" and make sure that the paths are pointing to directories matching your installation. For example:
 - 1 `C:\ti\ccsv5\` for "CCS Installation".
 - 2 `C:\ti\ccsv5\tools\compiler\c2000\` for "Code Generation Tools".
 - 3 `C:\ti\bios_5_41_11_38\` for "DSP/BIOS Installation".

Code Generation and Creation of an .out file using CCSv5:

- Open the model c2000scitest.
- Double-Click on "Open Target-Side Model Default F28335". This opens the target model.
- Open the Target Hardware Resources tab and set the **IDE/Tool chain** parameter to Texas Instruments Code Composer Studio v4 (makefile generation only).
- Change the selected board and processor to match your hardware.
- Go to "Peripherals" > "SCI_A", and change the "Communication mode" to "Protocol".
- Click "OK" to save the changes on the Target Hardware Resources tab.
- Click "Build Model" button or press "Ctrl+B" to generate and build an .out executable file.
- Using CCSv5, connect to the Hardware and load and run the .out file.
- Now simulate the model c2000scitest.

Change the manual switch position to select a different input signal.

CAN-Based Control of PWM Duty Cycle

This example shows how to use the CAN Receive, CAN Transmit, PWM, and C2812 Memory Copy blocks to generate a pulse waveform.

Required Hardware:

- Spectrum Digital F2808/F2812/F28335 eZdsp
- Texas Instruments F2808/F28035/F28069/F28335/F28343 ControlCARD with docking station
- Texas Instruments F28069 ControlSTICK
- Oscilloscope and probes

Available versions of this example:

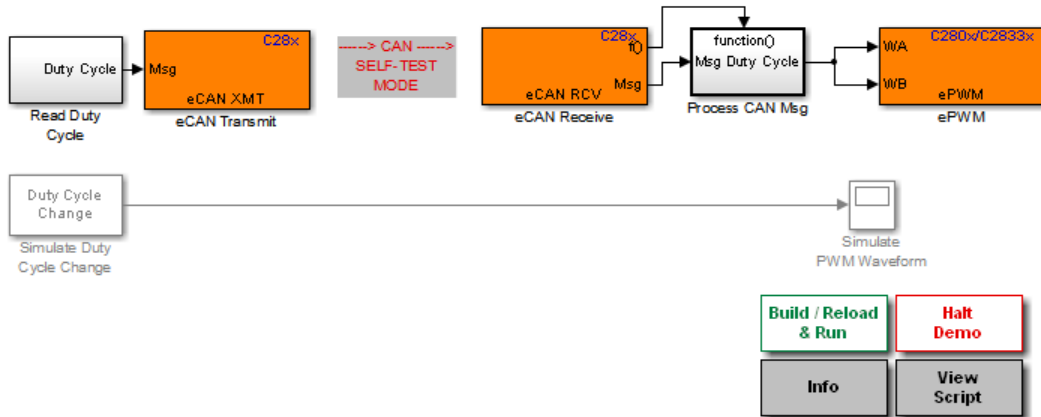
- **F281x** based board: c281x_cantest
- **F280x/F2823x/F2833x** based board: c280x_2833x_cantest
- **Piccolo F2803x/F2806x** based board: c280xx_cantest

Note: These models only runs with **Code Composer Studio v3.3**

Model

The following figure shows the example model.

CAN-Based Control of PWM Duty Cycle



Description

Memory Copy block labeled "Read Duty Cycle" reads duty cycle value stored at the memory location 0x8FF0. "Transmit CAN Msg" block uses mailbox 5 to transmit this value packed in a standard CAN message frame, while "Receive CAN Msg" uses mailbox 0 to receive it. When the CAN message is received, "Process CAN Msg" unpacks the duty cycle value and sends it to the PWM1. The duty cycle of the generated pulse waveform is determined by the relative ratio of the received pulse width value and the pulse period which is fixed at 64000 clock cycles. The duty cycle toggles between 25%, 50%, and 75% based on user selection.

PWM Duty Cycle can be changed while the generated code is running using the write function. This method writes selected value to the memory location 0x8FF0. Select either 25%, 50%, or 75% value from the GUI that opens up when "Build/Reload & Run" is pressed.

NOTE: 'Self-Test' mode is enabled for eCAN_A to receive the data transmitted back to the processor in the example models. If 'Self-test' mode is disabled, the CAN transmitter and receiver have to be physically connected for the example to work.

Note that "Simulate Duty Cycle Change" and "Simulate PWM Waveform" are dedicated to simulation only. Use these blocks to simulate and observe effects achieved with the generated code.

Hardware Connections

Connect the output of PWM1 on the board to the analog input of the oscilloscope.

Note that CAN module is set to work in "Self Test Mode". External CAN hardware is not needed, as the CAN handshaking signals are emulated in software.

How to Run the Example in Simulation

- Open the model corresponding to the family of processor you are using.
- Click **Start simulation** button
- Double-click on the "**Duty Cycle Change**" block and select a new duty cycle value
- Observe the PWM waveform change

How to Run the Example on the Board

- Open the model corresponding to the family of processor you are using.
- Each model is configured for a default processor, open the Target Hardware Resources tab to select a different processor.
- Go to **Peripherals > eCAN_A** and check **Self-Test Mode**.
- Click the "**Build/Reload & Run**" button to generate, build, load and run the program.
- Select a new duty cycle value in the dialog that opens.
- Observe the PWM waveform on the oscilloscope.

CAN Calibration Protocol and External Mode

This example shows how to use of the CAN Calibration Protocol (CCP) block. You can use either external mode or a third party calibration tool to interact with the real-time application running on the target hardware. CCP provides a means of monitoring signals and altering the parameter values in the application code running on the target.

Required hardware:

- Before running the model in external mode, install Vector-Informatik CAN hardware and drivers on your host computer, and set the baud rate to 1M to match processor CAN configurations.
- Spectrum Digital F2812/F2808/F28335 eZdsp board, F28069 controlSTICK or F2808/F28035/F28044/F28069/F28335 ControlCARD with Docking Station
- The model runs in MATLAB Win32 as Vector-Informatik supports win32

Available versions of this example:

- The example model `c28x_ccp` can be used for TI Piccolo F2803x/F2806x, F28x3x, F280x, F281x series processors. Default F28335 processor is selected.
- The target hardware can be selected from Target Hardware Resources tab for the model

Required hardware:

- Before running the model in external mode, install Vector-Informatik CAN hardware and drivers on your host computer, and set the baud rate to 1M to match processor CAN configurations.
- Spectrum Digital F2812/F2808/F28335 eZdsp board, F28069 controlSTICK or F2808/F28035/F28044/F28069/F28335 ControlCARD with Docking Station
- The model runs in MATLAB Win32 as Vector-Informatik supports win32

Available versions of this example:

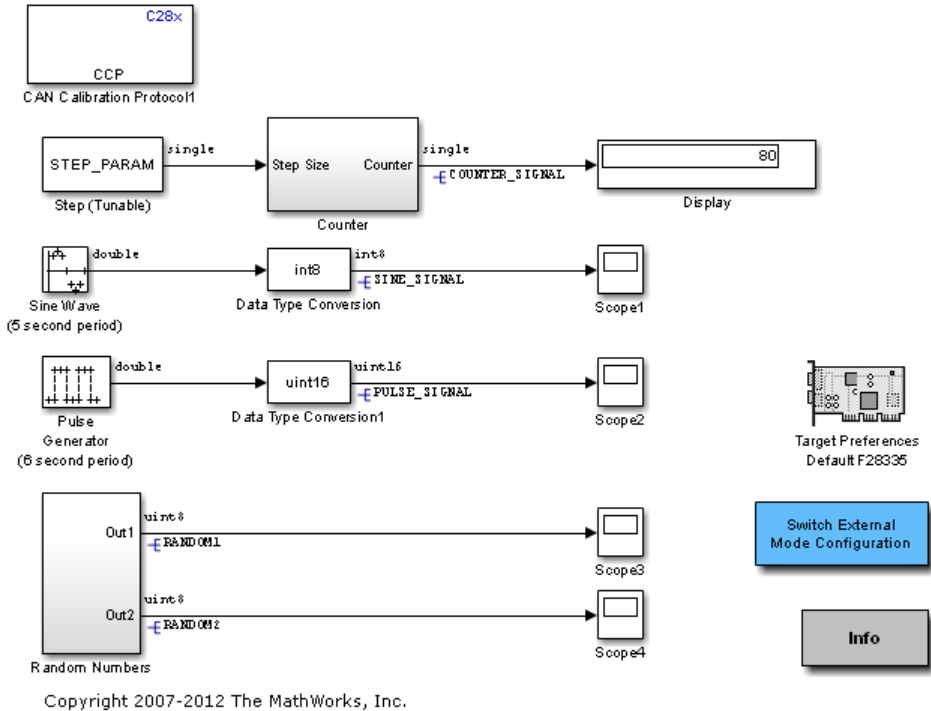
- The example model `c28x_ccp` can be used for TI Piccolo F2803x/F2806x, F28x3x, F280x, F281x series processors. Default F28335 processor is selected.
- The target hardware can be selected from Target Hardware Resources tab for the model

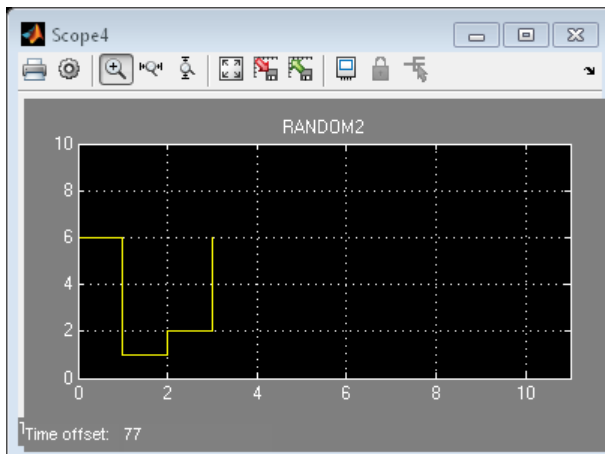
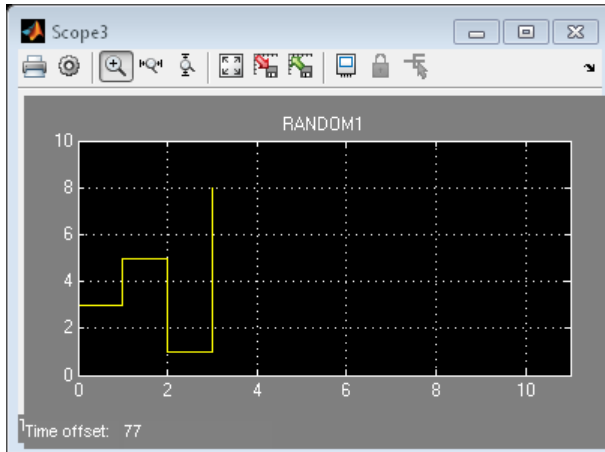
Model

The following figure shows the example model.

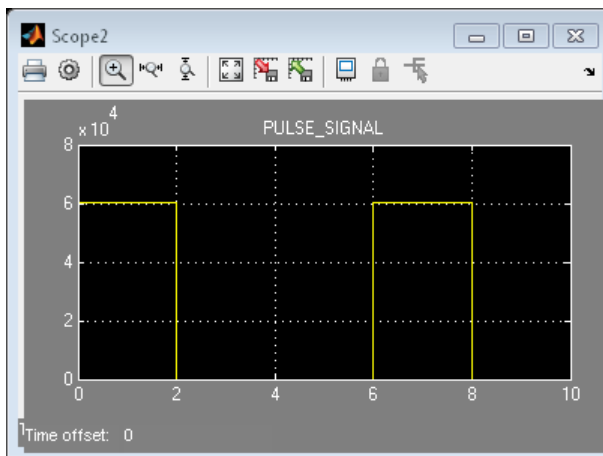
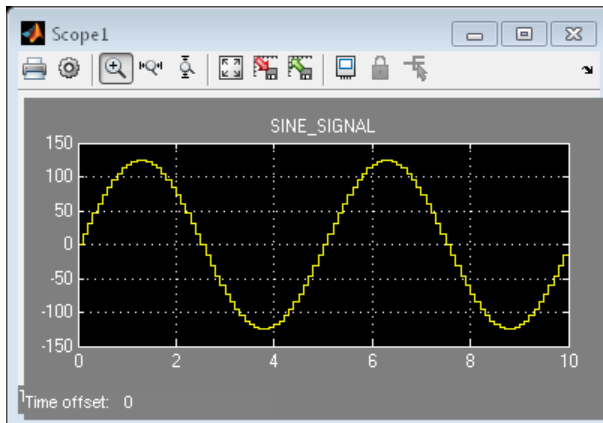
CAN Calibration Protocol and Simulink External Mode

Note: F2802x processors do not have an eCAN module





Scope Snapshots



Description

This example shows how to use the c28x CCP block. During code generation, for this model an ASAP2 file is generated. The ASAP2 file contains symbol and memory address information that is used either by external mode or a third party calibration tool. You can use either external mode or a calibration tool to log signals and update parameters on the real-time application.

How to Run the Example

The following sections explain how to configure, build, and deploy the executable.

With Code Composer Studio v3.3 (CCSv3.3) - Default

- While opening the model `c28x_ccp`, a script will be processed as a Pre-Load Function callback. This call can be viewed by selecting File > Model Properties > Callbacks > PreLoadFcn.
- The `c2000_ccp_data.m` callback is setting Simulink signals and parameters to resolve Simulink objects used for CCP DAQ lists and Simulink tunable parameters as described in the CCP documentation. This step is required to interact with these signals in real-time using CCP.
- The example model is configured for default processor F28335. Open the Target Hardware Resources tab to select a different processor.
- Build the model. You can press **Ctrl+B** or click **Incremental Build** button on the toolbar of your model.
- Follow the progress of the build in the MATLAB command window, and wait until the project is created and built in Code Composer Studio™.

With Code Composer Studio v4 (CCSv4)

CCSv4 support is provided via a makefile based approach. This requires running `xmakefilesetup` to set the environment. You can ignore the following if you have already gone through these steps. For more information, please refer to the Xmakefile Documentation

- Run XMakefile User Configuration.
- Set the Template parameter to **gmake**.
- Uncheck the **Display operational configuration only** checkbox.
- Set the Configuration parameter to **ticcs_c2000_ccsv4**, click **Apply**, if you haven't used this tool before, a dialog will ask you to browse to your CCSv4 installation folder. Pick the right folder and click **OK**.
- Click on the **Tool Directories** and make sure that the paths are pointing to directories matching your installation. For example:

- 1 *C:\Program Files\Texas Instruments\ccsv4* for **CCS Installation**
- 2 *C:\Program Files\Texas Instruments\ccsv4\tools\compiler\C2000* for **Code Generation Tools**
- 3 *C:\Program Files\Texas Instruments\bios_5_41_10_36* for **DSP/BIOS Installation**

Code Generation and Creation of an .out file using CCSv4:

- While opening the model `c28x_ccp`, a script will be processed as a Pre-Load Function callback. This call can be viewed by selecting `File > Model Properties > Callbacks > PreLoadFcn`.
- The `c2000_ccp_data.m` callback is setting Simulink signals and parameters to resolve Simulink objects used for CCP DAQ lists and Simulink tunable parameters as described in the CCP documentation. This step is required to interact with these signals in real-time using CCP.
- Open the Target Hardware Resources tab and set the **IDE/Tool chain** parameter to **Texas Instruments Code Composer Studio v4** (makefile generation only).
- Change the selected board and processor to match your hardware.
- Click **OK** to save the changes on the Target Hardware Resources tab.
- Build the model. You can press **Ctrl+B** or click **Incremental Build** button on the toolbar of your model.
- Open CCSv4 and Connect to the Hardware. Load the program on to the Hardware using CCSv4 and run.

With Code Composer Studio v5 (CCSv5)

CCSv5 support is provided via a makefile based approach. This requires running `xmakefilesetup` to set the environment. You can ignore the following if you have already gone through these steps. For more information, please refer to the Xmakefile Documentation

- Run XMakefile User Configuration.
- Set the Template parameter to **gmake**.
- Uncheck the **Display operational configuration only** checkbox.
- Set the Configuration parameter to **ticcs_c2000_ccsv5**, click **Apply**, if you haven't used this tool before, a dialog will ask you to browse to your CCSv5 installation folder. Pick the right folder and click **OK**.
- Click on the **Tool Directories** and make sure that the paths are pointing to directories matching your installation. For example:

- 1 `C:\ti\ccsv5\` to set **CCS Installation**
- 2 `C:\ti\ccsv5\tools\compiler\c2000\` to set **Code Generation Tools**
- 3 `C:\ti\bios_5_41_11_38\` to set **DSP/BIOS Installation**

Code Generation and Creation of an .out file using CCSv5

- While opening the model `c28x_ccp`, a script will be processed as a Pre-Load Function callback. This call can be viewed by selecting File > Model Properties > Callbacks > PreLoadFcn.
- The `c2000_ccp_data.m` callback is setting Simulink signals and parameters to resolve Simulink objects used for CCP DAQ lists and Simulink tunable parameters as described in the CCP documentation. This step is required to interact with these signals in real-time using CCP.
- Open the Target Hardware Resources tab and set the **IDE/Tool chain** parameter to Texas Instruments Code Composer Studio v5 (makefile generation only).
- Change the selected board and processor to match your hardware.
- Click **OK** to save the changes on the Target Hardware Resources tab.
- Build the model. You can press **Ctrl+B** or click **Incremental Build** button on the toolbar of your model.
- Open CCSv5 and Connect to the Hardware. Load the program on to the Hardware using CCSv5 and run.

Signal Logging and Parameter Tuning with External Mode

The model, `c28x_ccp`, contains a variety of signals and parameters that can be logged or changed while the application is executing in real-time. Parameter tuning and signal logging is enabled by the CCP block included in the model. Additionally, when the model is built, it must be configured to generate an ASAP2 file. The ASAP2 file contains information about the signals and parameters that are available for logging or tuning in the real-time application.

The model contains a simple counter with parameterized step size, `STEP_PARAM`. You can update the parameter `STEP_PARAM` and log the output of the counter, `COUNTER_SIGNAL`. Additional signals available for logging are `SINE_SIGNAL`, `PULSE_SIGNAL` and the `RANDOMx` signals.

Before running the model in external mode you must have Vector-Informatik CAN hardware installed on your host computer.

To run the model, `c28x_ccp`, in external mode:

- Check that the model is already built and running on the target.
- Configure the model to use External mode.

- Run the model in external mode by selecting Simulation > Connect To Target.
- You can now view logged data in the scope windows.
- You can change the parameter STEP_PARAM in the real-time application: first change its value, "STEP_PARAM.Value", in the MATLAB workspace; then update the model to apply this new value to the real-time application; to update the model select Edit > Update Diagram or press **Ctrl+D**.

Signal Logging and Parameter Tuning with a Third Party Calibration Tool

As an alternative to using external mode, you can use a third party calibration tool for signal logging and parameter tuning. The same signals and parameters may be logged or updated as those that are available with external mode.

Using the I2C Bus to Access a Connected EEPROM

This example shows how to use the I2C blocks to access an EEPROM connected to a chip using the I2C bus.

Required Hardware:

- Spectrum Digital F2808/F28335 eZdsp board

Note: C2000 (except c281x) controlSTICK or ControlCARD with Docking Station board is not equipped with I2C EEPROM. You must add I2C EEPROM to this board to run the example.

Available versions of this example:

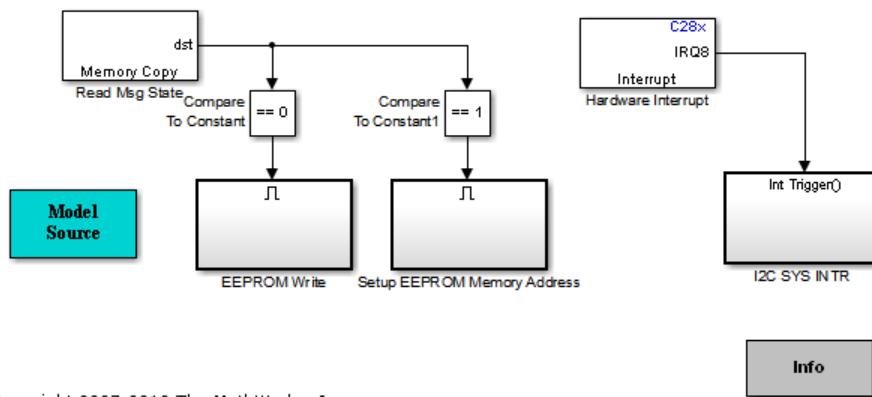
- For c28x Processors: c28x_i2ctest

Model

The following figure shows the example model.

Using the I2C bus to access a connected EEPROM

Note: This program will access the on-board I2C EEPROM provided on the F28335/F2808 eZdsp.



Copyright 2007-2013 The MathWorks, Inc.

Description

This example shows how to access the EEPROM connected to the I2C bus at I2C slave address 0x50 on an F2808/F28335 eZdsp board.

This program writes 4 bytes to EEPROM and reads back the data from the corresponding EEPROM address to show that communication has occurred.

The I2C Transmit block in the "EEPROM Write" subsystem writes the free-running counter data from the "EEPROM Data" subsystem to the address specified by "EEPROM Memory Addr" subsystem.

When an I2C interrupt is detected, "I2C SYS INTR" subsystem function is executed. As a result, the I2C Receive block reads the EEPROM data in "I2C SYS INTR:ARDY Subsystem:Subsystem". The global array "EEPROM_DATA" contains the received data. You can monitor the value of this array in Code Composer Studio™ Watch Window.

How to Run the Example

The following sections explain how to configure, build, and deploy the executable.

With Code Composer Studio v3.3 (CCSv3.3) - Default

- Open this model `c28x_i2ctest`.
- The model is configured for a default processor F28335. Open the Target Hardware Resources tab to select a different processor.
- In the Target Hardware Resources tab, under the Peripherals tab, select I2C, and check **Enable system interrupt**.
- Once editable, check **Enable SCD Interrupt** and **Enable ARDY interrupt**.
- Click **OK** to save the changes on the Target Hardware Resources tab.
- Click **Incremental build** on the Simulink toolbar or press **Ctrl+B** to generate, build, load and run the program.

With Code Composer Studio v4 (CCSv4)

CCSv4 support is provided via a makefile based approach. This requires running `xmakefilesetup` to set the environment. You can ignore the following if you have already gone through these steps. For more information, please refer to the Xmakefile Documentation

- Run XMakefile User Configuration.
- Set the Template parameter to **gmake**.

- Uncheck the **Display operational configuration only** checkbox.
- Set the Configuration parameter to **ticcs_c2000_ccsv4**, click **Apply**, if you haven't used this tool before, a dialog will ask you to browse to your CCSv4 installation folder. Pick the right folder and click **OK**.
- Click on the **Tool Directories** and make sure that the paths are pointing to directories matching your installation. For example:
 - 1 *C:\Program Files\Texas Instruments\ccsv4* for **CCS Installation**.
 - 2 *C:\Program Files\Texas Instruments\ccsv4\tools\compiler\C2000* for **Code Generation Tools**.
 - 3 *C:\Program Files\Texas Instruments\bios_5_41_10_36* for **DSP/BIOS Installation**.

Code Generation and Creation of an .out file using CCSv4:

- Open the model *c28x_i2ctest*.
- Open the Target Hardware Resources tab and set the **IDE/Tool chain** parameter to Texas Instruments Code Composer Studio v4 (makefile generation only).
- In the Target Hardware Resources tab, under the Peripherals tab, select I2C, and check **Enable system interrupt**.
- Once editable, check **Enable SCD Interrupt** and **Enable ARDY interrupt**.
- Click **OK** to save the changes on the Target Hardware Resources tab.
- Click **Incremental build** on the Simulink toolbar or press **Ctrl+B** to generate and build an .out executable file.
- Open CCSv4 and connect to the Hardware. Load the program on to the Hardware using CCSv4 and run.

With Code Composer Studio v5 (CCSv5)

CCSv5 support is provided via a makefile based approach. This requires running *xmakefilesetup* to set the environment. You can ignore the following if you have already gone through these steps. For more information, please refer to the Xmakefile Documentation

- Run XMakefile User Configuration.
- Set the Template parameter to **gmake**.
- Uncheck the **Display operational configuration only** checkbox.

- Set the Configuration parameter to **ticcs_c2000_ccsv5**, click **Apply**, if you haven't used this tool before, a dialog will ask you to browse to your CCSv5 installation folder. Pick the right folder and click **OK**.
- Click on the **Tool Directories** and make sure that the paths are pointing to directories matching your installation. For example:

- 1 *C:\ti\ccsv5* for **CCS Installation**.
- 2 *C:\ti\ccsv5\tools\compiler\c2000* for **Code Generation Tools**.
- 3 *C:\ti\bios_5_41_11_38* for **DSP/BIOS Installation**.

Code Generation and Creation of an .out file using CCSv5:

- Open the model *c28x_i2ctest*.
- Open the Target Hardware Resources tab and set the **IDE/Tool chain** parameter to Texas Instruments Code Composer Studio v5 (makefile generation only).
- Change the selected board and processor to match your hardware.
- In the Target Hardware Resources tab, under the Peripherals tab, select I2C, and check **Enable system interrupt**.
- Once editable, check **Enable SCD Interrupt** and **Enable ARDY interrupt**.
- Click **OK** to save the changes on the Target Hardware Resources tab.
- Click **Incremental build** on the Simulink toolbar or press **Ctrl+B** to generate and build an .out executable file.
- Open CCSv5 and connect to the Hardware. Load the program on to the Hardware using CCSv5 and run.

Using the IQmath Library

In this section...

“About the IQmath Library” on page 41-81

“Fixed-Point Numbers” on page 41-82

“Building Models” on page 41-86

About the IQmath Library

- “Introduction” on page 41-81
- “Common Characteristics” on page 41-82
- “References” on page 41-82

Introduction

The C28x IQmath Library blocks perform processor-optimized fixed-point mathematical operations. These blocks correspond to functions in the Texas Instruments C28x IQmath Library, an assembly-code library for the TI C28x family of digital signal processors.

Note: The implementation of this library for the TI C28x processor produces the same simulation and code-generation output as the TI version of this library, but it does not use a global Q value, as does the TI version. The Q format is dynamically adjusted based on the Q format of the input data.

The IQmath Library blocks generally input and output fixed-point data types and use numbers in Q format. The C28x IQmath Library block reference pages discuss the data types accepted and produced by each block in the library. For more information, consult the “Fixed-Point Numbers” on page 41-82 and “Q Format Notation” on page 41-84 topics, as well as the Fixed-Point Designer product documentation, which includes more information on fixed-point data types, scaling, and precision issues.

You can use IQmath Library blocks with some core Simulink blocks and Fixed-Point Designer blocks to run simulations in Simulink models before generating code. Once you develop your model, you can generate equivalent code that is optimized to run on a TI C28x DSP. During code generation, a call is made to the IQmath Library for each

IQmath Library block in your model to create target-optimized code. To learn more about creating models that include IQmath Library blocks and blocks from other blocksets, consult “Building Models” on page 41-86.

Common Characteristics

The following characteristics are common to IQmath Library blocks:

- Sample times are inherited from driving blocks.
- Blocks are single rate.
- Parameters are not tunable.
- Blocks support discrete sample times.

To learn more about characteristics particular to each block in the library, see the individual block reference pages.

References

For detailed information on the IQmath library, see the user's guide for the *C28x IQmath Library - A Virtual Floating Point Engine*, Literature Number SPRC087, available at the Texas Instruments Web site. The user's guide is included in the zip file download that also contains the IQmath library (registration required).

Fixed-Point Numbers

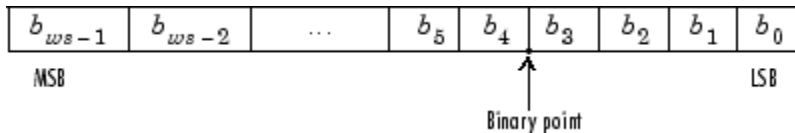
- “Notation” on page 41-82
- “Signed Fixed-Point Numbers” on page 41-83
- “Q Format Notation” on page 41-84

Notation

In digital hardware, numbers are stored in binary words. A binary word is a fixed-length sequence of binary digits (1s and 0s). How hardware components or software functions interpret this sequence of 1s and 0s is defined by the data type.

Binary numbers are used to represent either fixed-point or floating-point data types. A fixed-point data type is characterized by the word size in bits, the binary point, and whether it is signed or unsigned. The position of the binary point is the means by which fixed-point values are scaled and interpreted.

For example, a binary representation of a fractional fixed-point number (either signed or unsigned) is shown below:



where

- b_i is the i th binary digit.
- ws is the word size in bits.
- b_{ws-1} is the location of the most significant (highest) bit (MSB).
- b_0 is the location of the least significant (lowest) bit (LSB).
- The binary point is shown four places to the left of the LSB. In this example, therefore, the number is said to have four fractional bits, or a fraction length of 4.

Note: For Embedded Coder, the results of fixed-point and integer operations in MATLAB/Simulink match the results on the hardware target down to the least significant bit (bit-trueness). The results of floating-point operations in MATLAB/Simulink do not match those on the hardware target, because the libraries used by the third-party compiler may be different from those used by MATLAB/Simulink.

Signed Fixed-Point Numbers

Signed binary fixed-point numbers are typically represented in one of three ways:

- Sign/magnitude
- One's complement
- Two's complement

Two's complement is the most common representation of signed fixed-point numbers and is used by TI digital signal processors.

Negation using signed two's complement representation consists of a bit inversion (translation to one's complement representation) followed by the binary addition of a 1. For example, the two's complement of 000101 is 111011, as follows:

000101 ->111010 (bit inversion) ->111011 (binary addition of a 1 to the LSB)

Q Format Notation

The position of the binary point in a fixed-point number determines how you interpret the scaling of the number. When it performs basic arithmetic such as addition or subtraction, hardware uses the same logic circuits regardless of the value of the scale factor. In essence, the logic circuits do not have knowledge of a binary point. They perform signed or unsigned integer arithmetic — as if the binary point is to the right of b_0 . Therefore, you determine the binary point.

In the IQmath Library, the position of the binary point in the signed, fixed-point data types is expressed in and designated by Q format notation. This fixed-point notation takes the form

$Qm.n$

where

- Q designates that the number is in Q format notation — the Texas Instruments representation for signed fixed-point numbers.
- m is the number of bits used to designate the two's complement integer portion of the number.
- n is the number of bits used to designate the two's complement fractional portion of the number, or the number of bits to the right of the binary point.

In Q format, the most significant bit is designated as the sign bit. Representing a signed fixed-point data type in Q format requires $m+n+1$ bits to account for the sign.

Note: The range and resolution varies for different Q formats. For specific details, see Section 3.2 in the *Texas Instruments C28x Foundation Software, IQmath Library Module User's Guide*.

When converting from Q format to floating-point format, the accuracy of the conversion depends on the values and formats of the numbers. For example, for single-precision floating-point numbers that use 24 bits, the resolution of the corresponding 32-bit number cannot be achieved. The 24-bit number approximates its value by truncating the lower end. For example:

32-bit integer 11110000 11001100 10101010 00001111

Single-precision float +1.1110000 11001100 10101010 x 231

Corresponding value 11110000 11001100 10101010 00000000

Expressing Q Format – Q.15

For example, a signed 16-bit number with $n = 15$ bits to the right of the binary point is expressed as

Q0.15

in this notation. This is (1 sign bit) + ($m = 0$ integer bits) + ($n = 15$ fractional bits) = 16 bits total in the data type. In Q format notation, the $m = 0$ is often implied, as in

Q.15

In Fixed-Point Designer software, this data type is expressed as

sfrac16

or

sfix16_En15

In DSP System Toolbox software, this data type is expressed as

[16 15]

Expressing Q Format – Q1.30

Multiplying two Q0.15 numbers yields a product that is a signed 32-bit data type with $n = 30$ bits to the right of the binary point. One bit is the designated sign bit, thereby forcing m to be 1:

$$m+n+1 = 1+30+1 = 32 \text{ bits total}$$

Therefore, this number is expressed as

Q1.30

In Fixed-Point Designer software, this data type is expressed as

sfix32_En30

In DSP System Toolbox software, this data type is expressed as

[32 30]

Expressing Q Format – Q-2.17

Consider a signed 16-bit number with a scaling of $2^{(-17)}$. This requires $n = 17$ bits to the right of the binary point, meaning that the most significant bit is a *sign-extended* bit.

Sign extension fills additional bits with the value of the MSB. For example, consider a 4-bit two's complement number 1011. When this number is extended to 7 bits with sign extension, the number becomes 1111101 and the value of the number remains the same.

One bit is the designated sign bit, forcing m to be -2:
 $m+n+1 = -2+17+1 = 16$ bits total

Therefore, this number is expressed as

Q-2.17

In Fixed-Point Designer software, this data type is expressed as

`sfix16_En17`

In DSP System Toolbox software, this data type is expressed as

[16 17]

Expressing Q Format – Q17.-2

Consider a signed 16-bit number with a scaling of 2^2 or 4. This means that the binary point is implied to be 2 bits to the right of the 16 bits, or that there are $n = -2$ bits to the right of the binary point. One bit must be the sign bit, thereby forcing m to be 17:
 $m+n+1 = 17+(-2)+1 = 16$

Therefore, this number is expressed as

Q17.-2

In Fixed-Point Designer software, this data type is expressed as

`sfix16_E2`

In DSP System Toolbox software, this data type is expressed as

[16 -2]

Building Models

- “Overview” on page 41-87
- “Converting Data Types” on page 41-87
- “Using Sources and Sinks” on page 41-87

- “Choosing Blocks to Optimize Code” on page 41-87
- “Double and Single-Precision Parameter Values” on page 41-88

Overview

You can use IQmath Library blocks in models along with certain core Simulink, Fixed-Point Designer, and other blockset blocks. This section discusses issues you should consider when building a model with blocks from these different libraries.

Converting Data Types

It is vital to make sure that blocks you connect in a model have compatible input and output data types. In most cases, IQmath Library blocks handle only a limited number of specific data types. You can refer to the block reference page for a discussion of the data types that the block accepts and produces.

When you connect IQmath Library blocks and Fixed-Point Designer blocks, you often need to set the data type and scaling in the block parameters of the Fixed-Point Designer block to match the data type of the IQmath Library block. Many Fixed-Point Designer blocks allow you to set their data type and scaling through inheritance from the driving block, or through backpropagation from the next block. This can be a good way to set the data type of a Fixed-Point Designer block to match a connected IQmath Library block.

Some DSP System Toolbox blocks and core Simulink blocks also accept fixed-point data types. Choose the right settings in these blocks' parameters when you connect them to an IQmath Library block.

Using Sources and Sinks

The IQmath Library does not include source or sink blocks. Use source or sink blocks from the core Simulink library or Fixed-Point Designer in your models with IQmath Library blocks.

Choosing Blocks to Optimize Code

In some cases, blocks that perform similar functions appear in more than one blockset. For example, the IQmath Library and Fixed-Point Designer software have a Multiply block. When you are building a model to run on C2000 DSP, choosing the block from the IQmath Library yields better optimized code. You can use a similar block from another library if it gives you functionality that the IQmath Library block does not support, but you will generate code that is less optimized.

Double and Single-Precision Parameter Values

When you enter double-precision floating-point values for parameters in the IQ Math blocks, the software converts them to single-precision values that are compatible with the behavior on c28x processor. For example, with the **Ramp Generator** block, the software converts the value of the **Maximum step angle** parameter to a single-precision value.

Programming Flash Memory

In this section...

“Introduction” on page 41-89

“Installing TI Flash APIs” on page 41-89

“Configuring the DSP Board Bootloader” on page 41-90

“Configuring the Software for Automatic Flash Programming” on page 41-90

“Selectively Erase, Program, or Verify Specific Flash Sectors” on page 41-91

“Placing Additional Code or Data on Unused Flash Sectors” on page 41-91

Introduction

The Embedded Coder software includes a feature for programming Flash memory on the target hardware. You can configure this feature to automatically program Flash memory when you build and execute models for DSP boards. You can also use the Flash programming feature to selectively erase, program, or verify specific sectors of Flash memory.

Note: Reprogramming Flash memory thousands of times may deplete its ability to hold data. Consult the manufacturer's documentation for specifications.

Requirements:

- A F2812, F2808, or F28335 eZdsp board
- A working Simulink model
- The TI Flash API for your specific target

Installing TI Flash APIs

- 1 Visit the Texas Instruments Web site and download the TI Flash API installation software for your target:
 - F281x: <http://focus.ti.com/docs/toolsw/folders/print/sprc125.html>
 - F280x: <http://focus.ti.com/docs/toolsw/folders/print/sprc193.html>
 - F2802x: <http://focus.ti.com/docs/toolsw/folders/print/sprc848.html>

- F2804x: <http://focus.ti.com/docs/toolsw/folders/print/sprc325.html>
 - F2823x: <http://focus.ti.com/docs/toolsw/folders/print/sprc665.html>
 - F2833x: <http://focus.ti.com/docs/toolsw/folders/print/sprc539.html>
- 2** Start the TI Flash API installation software (.exe) contained in the ZIP file.
 - 3** During installation, *use the default folder location* for **Location to Save Files**.

Otherwise, each time you create a model, you must configure **Specify API Location**, located under the **Peripherals** tab of the Target Hardware Resources tab.

- 4** Complete the installation process.

Configuring the DSP Board Bootloader

Configure the bootloader switch or jumper on the DSP board so that, upon startup, the DSP board executes the program from Flash memory. Consult the manufacturer's hardware documentation to identify the specific switch and settings.

Typically, you can enable the bootloader switch or jumper by moving it from the factory default position (Flash disabled) to the opposite position (enabled). For example:

- On the F2812 eZdsp, change jumper JP7 from the factory default setting.
- On the F2808 eZdsp, change switches 1 and 3 on bank SW1 from the factory default settings.
- On F28335 eZdsp, change switch 3 on bank SW1 from the factory default setting.

Configuring the Software for Automatic Flash Programming

Configure Embedded Coder software to program Flash memory on the target board when you build and execute a model.

- 1** On your keyboard, press Ctrl+E to open the Configuration Parameters dialog, select the Code Generation > Coder Target pane.
- 2** On the Tool Chain Automation tab, set **Build Action** to **Build_and_execute**.
- 3** On the Target Hardware Resources tab, set **Board** to a board whose name includes (boot from flash).
- 4** Select the **Peripherals** tab, and then select **Flash_loader**
- 5** Set **Enable flash programmer** to **Erase, Program, Verify**.

- 6 Click **OK** to save and close the new configuration.

When you build the model, the software automatically erases, programs, and verifies Flash memory. When the DSP board restarts, it loads and executes the program from Flash memory.

Selectively Erase, Program, or Verify Specific Flash Sectors

You can manually erase, program, and verify specific sectors of Flash memory:

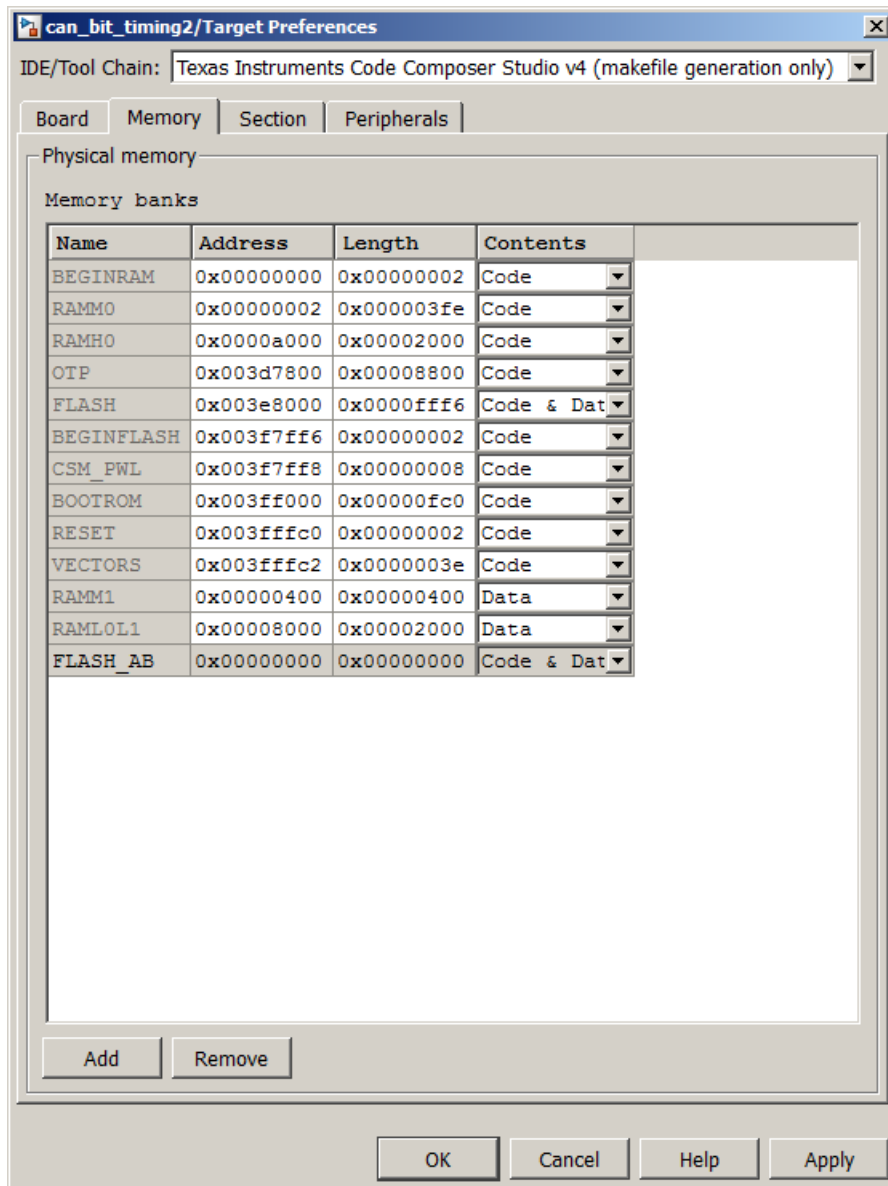
- 1 Open the Target Hardware Resources tab for your model, and select the **Peripherals** tab.
- 2 Select **Flash_loader** from the **Peripherals** list.
- 3 Set **Enable flash programmer** to erase, program, or verify flash.
- 4 (Optional) To protect specific Flash sectors:
 - a Disable **Detect Flash sectors to erase from COFF file**.
 - b Clear the flash sectors you want to protect.
- 5 Click **Execute**. The software performs the action you specified upon the unprotected flash sectors.

Note: Erase Flash sectors before programming them.

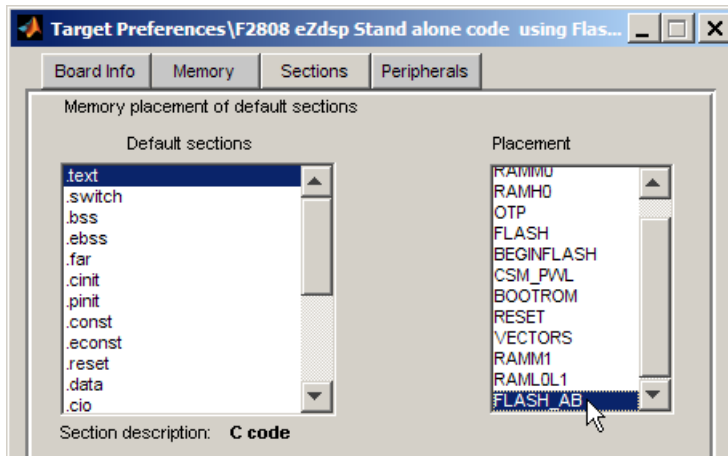
Placing Additional Code or Data on Unused Flash Sectors

To place additional code or data on unused Flash sectors:

- 1 Determine the address and length of the individual Flash sectors. You may need to refer to the manufacturer's specifications.
- 2 Determine the size of the primary C code program and the number of Flash sectors it occupies.
- 3 Determine the size of the additional code or data and the number of Flash sectors it will occupy.
- 4 Under the Target Hardware Resources tab, on the **Memory** tab, click **Add** to create two or more new memory banks; one for the primary C code program (e.g., FLASH_AB) and one or more for the additional code or data (e.g., FLASH_CD). The address and length of each memory bank must align with those of the flash sectors.



- 5 Under the **Sections** tab, under **Default sections**, select **.text**. Then, under **Placement**, select the new memory bank (e.g., FLASH_AB) you created for the primary C code program. The next time you program the Flash memory, the software places the **.text** C code file in the new memory bank.



- 6 Similarly, select items from the **Default sections** or **Custom sections list**, and place them in the new memory banks (e.g., FLASH_CD) for the previously unoccupied Flash sectors.

Configuring LIN Communications

In this section...
“Overview” on page 41-94
“Configuring Your Model” on page 41-94

Overview

The LIN communications architecture supports a single master node and up to 16 slave nodes on a LIN network.

LIN nodes use message frames to exchange data. The message has two parts:

- Frame header, generated by the Master node.
- Frame response, which contains data generated by either Slave node or a slave task on a Master node (but not both).

Configuring Your Model

First, study, and understand the LIN addressing system. See the “Message Filtering and Validation” topic in the TMS320F2803x Piccolo Local Interconnect Network (LIN) Module, Literature Number: SPRUGE2B.

Configure the LIN node in your model as a master or slave node:

- 1 Configure the Target Hardware Resources tab, as described in “Configure Target Hardware Resources”.
- 2 In the Target Hardware Resources tab, select the **Peripherals** tab, and then select **LIN**.
- 3 Set **LIN mode** to **Master** or **Slave**.

If the LIN node is a Master node:

- Add a LIN Transmit block to the model. This block enables the Master to generate message headers.
- To send data, set the **ID** input and **Tx ID Mask** input to make Tx ID Match happen on this node.

- To receive data, place LIN Receive block in the model. Set the **Rx ID Mask** input to make Rx ID Match happen on this node.

For example, to configure a model with a master node that receives data from a slave node:

- Add a LIN Transmit block and a LIN Receive block to the model.
- In the Target Hardware Resources tab, configure the **ID Slave Task Byte**.
- For the LIN Transmit block, set the **ID** input.
- For the LIN Receive block, set the **Rx ID Mask** input so that: **Rx ID Mask = ID XOR Slave Task ID Byte**.

If the LIN node is a Slave node:

- To send data, place LIN Transmit block in the model. Set the **ID** input to match the LIN frame header issued by the remote Master. Set **Tx ID Mask** to make a Tx ID Match happen on this node.
- To receive data, place LIN Receive block in the model. Set the **Rx ID Mask** input to make an Rx ID Match happen on this node.

For example, to configure a model with a slave node that transmits data to a master node:

- Add a LIN Transmit block to the model.
- In the Target Hardware Resources tab, configure the **ID byte** or **ID Slave Task Byte** (depending on the **ID filtering** option).
- In the LIN Transmit block, set the **ID** input and **Tx ID Mask** input so that: **Tx ID Mask = ID XOR (ID Byte or ID Slave Task Byte)**.

Set the **Data type** and **Data length** values in your LIN Receive blocks to match the type and length of the transmitted data. These values enable the receive block reconstruct the data from the message frames.

Note: The LIN Transmit block inherits the data type and length from its input.

Open Examples for Embedded Coder Texas Instruments C2000

The examples available for Embedded Coder Texas Instruments C2000 are as follows:

Demos based on `ert.tlc`

- ADC-PWM Synchronization via ADC Interrupt — `c280x_2833x_adcpwmasynctest_ert.mdl`
- SPI-Based Control of PWM Duty Cycle — `c280x_2833x_spitest_ert.mdl`
- Asynchronous Scheduling — `c280x_2833x_asynscheduling_ert.mdl`
- CAN Calibration Protocol and External Mode — `c28x_ccp_ert.mdl`
- Permanent Magnet Synchronous Motor Field-Oriented Control — `c28069pmsmfoc_ert.mdl`
- Using the Control Law Accelerator (CLA).mdl
- Schedule a Multi-Rate Controller for a Permanent Magnet Synchronous Machine — `c28335_pmsmfoc_script.mdl`

Demos based on `eidelink_ert.tlc` and `idelink_grt.tlc`

- ADC-PWM Synchronization via ADC Interrupt — `c280x_2833x_adcpwmasynctest.mdl`
- CAN-Based Control of PWM Duty Cycle — `c280x_2833x_cantest.mdl`
- SPI-Based Control of PWM Duty Cycle — `c280x_2833x_spitest.mdl`
- Using the I2C Bus to Access a Connected EEPROM — `c28x_i2ctest.mdl`
- LIN-Based Control of PWM Duty Cycle — `c28035lintest.mdl`
- Asynchronous Scheduling — `c280x_2833x_asynscheduling.mdl`
- DC Motor Speed Control via RTDX — `c2812speedcontrolDC.mdl`
- CAN Calibration Protocol and External Mode — `c28x_ccp.mdl`
- Real-Time Data Exchange via RTDX — `c28x_rtdxtest.mdl`
- HIL Verification of IIR Filter via SCI — `c2000scitest.mdl`
- Simulation of FOC Using PMSM Model — `c2812pmsmsim.mdl`
- Permanent Magnet Synchronous Motor Field-Oriented Control — `c2812pmsmfoc.mdl`
- Motor Control
- DC/DC Buck Converter

Note Make sure you first download and install Embedded Coder Texas Instruments C2000 support package before trying to open the examples.

Tips and Limitations

RTDX Simulation Blocks

RTDX™ Simulation Blocks Contents RTDX™ Simulation Blocks Library Host-Target Communication RTDX™ Read Block RTDX™ Write Block Relationship Between RTDX™ Blocks in Simulation and Target Application RTDX™ Simulation Blocks Library The RTDX Simulation Blocks Library contains two blocks: RTDX Read - Read from RTDX™ channel in simulation RTDX Write - Write to RTDX channel in simulation close open_system('rtdxsimlib'); The 'HOST' notation on the blocks indicates that these blocks work only in simulation. Note: To open the block library from MATLAB®, enter rtdxsimlib at the command prompt. Host-Target Communication When you add one of the blocks to a model, Simulink® adds a Host-Target Communication configuration option to the Configuration Parameters select tree. This new configuration option enables you to set up the target and RTDX. It is important that you enter the required target and RTDX information here before you run the model. The Host-Target Communication option is where you enter target set-up information: Target Information: Board number - board number of target Processor number - processor number on target board Target Application: Project name - CCS project file (optional) Program name - CCS program to load The RTDX Configuration option (under Host-Target Communication) is where you enter RTDX set-up information: RTDX mode - set to Continuous mode Number of buffers - number of RTDX buffers Buffer size - size (in bytes) of each RTDX buffer RTDX™ Read Block Description In simulation, this block reads a message from the specified RTDX channel and outputs the message to downstream blocks in the model. When your simulation uses RTDX to communicate with a target, the RTDX Read block gets messages from the RTDX channel that your target application writes to. If your target model is running code you generated from a Simulink model that used RTDX blocks from the Target Support Package component software, the To RTDX block you included in your target model configures and writes messages to the RTDX channel from which the RTDX Read block in your simulation is reading messages. Block Dialog Channel name Name of a read RTDX channel. The RTDX Read block in the host model reads data from the target through this channel. The name specified here must match an output channel defined in the target code. Sample time Specifies the time between samples of the signal. The default is 1 second between samples, for a sample rate of one sample per second (1/Sample time). Output dimensions Defines a matrix for the output signal from the block, where the first value is the number of rows and the second is the number of columns in the output matrix. For example, the default setting [1 64] represents a 1-row by 64-column matrix of output values. Enter a 1 by 2 vector of doubles for the dimensions. Frame-based Sets a flag at the block output that directs downstream blocks to use frame-based processing on the data from this block. In frame-based processing, the samples in a frame are processed simultaneously. In sample-based processing,

samples are processed one at a time. Frame-based processing can increase the speed of your application running on your target. Note that throughput in samples per second processed remains the same. Frame-based operation is the default. Data type Sets the type for the data output from the block. Select one of the following types: Double - double-precision floating-point numbers. Values range from -1 to 1. This is the default setting. Single - single-precision floating-point numbers. Values range from -1 to 1. Int16 - 16-bit signed integers. Values range from -32768 to 32767. Int32 - 32-bit signed integers. Values range from -2147483648 to -2147483647. Uint8 - 8-bit unsigned integers. Values range from 0 to 255. RTDX™ Write Block Description In simulation, this block writes the block input data to the specified RTDX channel. When your simulation uses RTDX to communicate with a target, the RTDX Write block writes messages to the RTDX channel that your target application reads. If your target model is running code you generated from a Simulink model that used RTDX blocks from the Target Support Package component software, the From RTDX block you included in your target model configures and reads messages from the RTDX channel to which the RTDX Write block in your simulation is sending messages. Block Dialog Channel name Name of a write RTDX channel. The RTDX Write block in the host model writes data to the target through this channel. The name specified here must match an input channel defined in the target code. Relationship Between RTDX™ Blocks in Simulation and Target Application To make the correspondence between the RTDX simulation blocks and the RTDX code generation blocks clear: Using the RTDX Read block in simulation provides one end of a RTDX channel. Using the To RTDX block in a model generates code to implement the other end. Together they form one complete RTDX communications path from the target application to the host simulation. Using the RTDX Write block in simulation provides one end of a RTDX channel. Using the From RTDX block in a model generates code to implement the other end. Together they form one complete RTDX communications path from the host simulation to the target application. `close_system('rtdxsimlib', 0);`
Copyright 2006-2011 The MathWorks, Inc. Published with MATLAB® R2013b

Code Generation from MATLAB Code

Build Configuration for Code Generation from MATLAB Code

- “Specify Comment Style” on page 42-2
- “Generate Custom File and Function Banners for C and C++ Code” on page 42-4
- “Code Generation Template (CGT) Files for MATLAB” on page 42-7
- “Customize Generated Identifiers” on page 42-18

Specify Comment Style

In this section...

“Specify Comment Style Using the Project Interface” on page 42-2

“Specify Comment Style at the Command-Line Interface” on page 42-2

If you have an Embedded Coder license, you can specify the comment style for C and C++ code generated from MATLAB code. Specify single-line style to generate single-line comments preceded by `//`. Specify multi-line style to generate single-line or multi-line comments delimited by `/*` and `*/`. Single-line style is the default for C++ code generation. Multi-line style is the default for C code generation. For C code generation, select single-line comment style only if your compiler supports it.

Specify Comment Style Using the Project Interface

- 1 In the Project Settings dialog box, click the **Code Appearance** tab.
- 2 Select the **Include Comments** check box if it is not already selected. By default, the **Include Comments** check box is selected.
- 3 Set **Comment Style** to one of the following options.

Value	Description
Auto(Use standard comment style of the target language)	For C, generate multi-line comments. For C++, generate single-line comments. (default)
Single-line (Use C++-style comments)	Generate single-line comments preceded by <code>//</code> .
Multi-line (Use C-style comments)	Generate single or multi-line comments delimited by <code>/*</code> and <code>*/</code> .

Specify Comment Style at the Command-Line Interface

- 1 Create a code configuration object for C/C++ code generation. For example, create a configuration object for C/C++ static library generation:

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

- 2 Set the **CommentStyle** property to one of the following values:

Value	Description
'Auto'	For C, generate multi-line comments. For C++, generate single-line comments. (default)
'Single-line'	Generate single-line comments preceded by //.
'Multi-line'	Generate single or multi-line comments delimited by /* and */.

For example, this code sets the comment style to single-line style:

```
cfg.CommentStyle='Single-line';
```

Generate Custom File and Function Banners for C and C++ Code

When you generate C and C++ code from MATLAB code, you can use a code generation template (CGT) file to specify custom:

- File banners
- Function Banners
- File trailers
- Comments before code sections

This example shows how you can create your own CGT file and customize it to generate your own file and function banners.

- 1 Create a local copy of the default CGT file for MATLAB Coder and rename it. The default CGT file is `matlabcoder_default_template.cgt` in the `matlabroot/toolbox/coder/matlabcoder/templates/` folder.
- 2 Store the copy in a folder that is outside of the MATLAB folder structure, but on the MATLAB path. If required, add the folder to the MATLAB path. If you intend to use the CGT file with a custom target, locate the CGT file in a folder under your target root folder. If the file is not on the MATLAB path, specify a full path to the file when adding the file to your configuration.
- 3 View the default template and generated output. For example, here is the default File Banner section:

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Custom File Banner section (optional)
%% Customize File banners by using either custom tokens or the following
%% predefined tokens:
%% <FileName>, <MATLABCoderVersion>, <EmbeddedCoderVersion>
%% <SourceGeneratedOn>, <HardwareSelection>, <OutputType>
%%
%% You can also use "custom tokens" in all of the sections below. See the
%% documentation center for more details.
%%
<FileBanner style="classic">
File: <FileName>

MATLAB Coder version      : <MATLABCoderVersion>
C/C++ source code generated on : <SourceGeneratedOn>
</FileBanner>

```

When you generate code using this default, the file banner looks similar to this file banner:

```
/*
```

```
* File: coderand.c
*
* MATLAB Coder version      : 2.7
* C/C++ source code generated on : 06-Apr-2014 14:34:15
*/
```

- 4** Edit your local copy of the CGT file. You can change the default values and add your own custom tokens. For example, here is the File Banner section with the style changed to box and a custom token `myCustomToken` :

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Custom File Banner section (optional)
%% Customize File banners by using either custom tokens or the following
%% predefined tokens:
%% %<FileName>, %<MATLABCoderVersion>, %<EmbeddedCoderVersion>
%% %<SourceGeneratedOn>, %<HardwareSelection>, %<OutputType>
%%
%% You can also use "custom tokens" in all of the sections below. See the
%% documentation center for more details.
%%
<FileBanner style="box">
File: %<FileName>

My custom token          : %<myCustomToken>

MATLAB Coder version    : %<MATLABCoderVersion>
C/C++ source code generated on : %<SourceGeneratedOn>
</FileBanner>
```

For more information, see “Code Generation Template (CGT) Files for MATLAB”.

- 5** Create a configuration object for generation of a C static library for an embedded target.

```
% Create configuration object for an embedded target
cfgObj = coder.config('lib','ecoder',true);
```

- 6** Create a `MATLABCodeTemplate` object from your CGT file and add it to the configuration object.

```
% Specify the custom CGT file
CGTFile = 'myCGTFile.cgt';
% Use custom template
cfgObj.CodeTemplate = coder.MATLABCodeTemplate(CGTFile);
```

- 7** Assign values for custom tokens that you added to the template. For example, assign the value `'myValue'` to the `myCustomToken` token that you added in a previous step.

```
cfgObj.CodeTemplate.setTokenValue('myCustomToken','myValue');
```

- 8** Generate code using the configuration object that you just created.

```
codegen -config cfgObj coderand
```

- 9 View the changes to the generated file banner. For example, here is the file banner for `coderand.c` using the customized CGT file:

```

/*****
/* File: coderand.c                                     */
/*                                                     */
/* My custom token           : myValue                 */
/*                                                     */
/* MATLAB Coder version      : 2.7                    */
/* C/C++ source code generated on : 06-Apr-2014 14:42:55 */
*****/

```

Changes to a CGT file do not affect the generated code unless you create a `MATLABCodeTemplate` object from the modified CGT file, and then add it to the configuration object. If you modify the CGT File, `myCGTFile.cgt`, used in the previous example, you must repeat these steps:

- 1 Create a `MATLABCodeTemplate` object from `myCGTFile.cgt` and add it to the configuration object.

```
CGTFile = 'myCGTFile.cgt';
cfgObj.CodeTemplate = coder.MATLABCodeTemplate(CGTFile);
```

- 2 Assign the value 'myValue' to the `myCustomToken` token.

```
cfgObj.CodeTemplate.setTokenValue('myCustomToken', 'myValue');
```

- 3 Generate code.

```
codegen -config cfgObj coderand
```

Code Generation Template (CGT) Files for MATLAB

In this section...

“Default CGT file” on page 42-7

“CGT File Structure” on page 42-7

“Components of the CGT File Sections” on page 42-9

A code generation template (CGT) file defines the sections in generated code that you can customize using comments and tokens. Using a code generation template (CGT) file for the generation of C and C++ code from MATLAB, you can specify custom file banners and function banners for generated code. File banners are comment sections in the header and trailer sections of a generated file. Function banners are comment sections for each function in the generated code. You can also customize comments before code sections. Use these banners to:

- Add a company copyright statement.
- Specify a special version symbol for your configuration management system.
- Remove time stamps.
- Add other custom information to your generated files.

For information on creating, customizing, and using a CGT file, see “Generate Custom File and Function Banners for C and C++ Code” on page 42-4.

Default CGT file

You can base your custom template on the default CGT file, `matlabcoder_default_template.cgt`, in the `matlabroot/toolbox/coder/matlabcoder/templates/` folder.

Note: If you choose not to customize banners for your generated code, the default template is used for code generation.

CGT File Structure

A CGT file consists of thirteen optional sections.

File Banner Section

Contains comments and tokens for use in generating a custom file banner.

Function Banner Section

Contains comments and tokens for use in generating a custom function banner.

Shared Utility Function Banner

Contains comments and tokens for use in generating custom banners for shared utility functions.

File Trailer Section

Contains comments for use in generating a custom trailer banner.

Include Files Banner

Contains comments for use in generating a custom banner for the include files section.

Type Definitions

Contains comments for use in generating a custom banner for the type definitions section.

Named Constants

Contains comments for use in generating a custom banner for the named constants section.

Variable Declarations

Contains comments for use in generating a custom banner for the variable declarations section.

Variable Definitions

Contains comments for use in generating a custom banner for the variable definitions section.

Function Declarations

Contains comments for use in generating a custom banner for the function declarations section.

Function Definitions

Contains comments for use in generating a custom banner for the function definitions section.

Custom Source Code

Contains comments for use in generating a custom banner for the custom source code section.

Custom Header Code

Contains comments for use in generating a custom banner for the custom header code section.

Components of the CGT File Sections

Each CGT file section is defined by open and close tags.

CGT File Section	Open Tag	Close Tag
“File Banner” on page 42-12	<FileBanner>	</FileBanner>
“Function Banner Section” on page 42-8	<FunctionBanner>	</FunctionBanner>
“Shared Utility Function Banner” on page 42-8	<SharedUtilityBanner>	</SharedUtilityBanner>
“File Trailer Section” on page 42-8	<FileTrailer>	</FileTrailer>
“Include Files Banner” on page 42-8	<IncludeFilesBanner>	</IncludeFilesBanner>
“Type Definitions” on page 42-8	<TypeDefinitionsBanner>	</TypeDefinitionsBanner>
“Named Constants” on page 42-8	<NamedConstantsBanner>	</NamedConstantsBanner>
“Variable Declarations” on page 42-8	<VariableDeclarationsBanner>	</VariableDeclarationsBanner>

CGT File Section	Open Tag	Close Tag
“Variable Definitions” on page 42-8	<VariableDefinitionsBanner>	</VariableDefinitionsBanner>
“Function Declarations” on page 42-8	<FunctionDeclarationsBanner>	</FunctionDeclarationsBanner>
“Function Definitions” on page 42-9	<FunctionDefinitionsBanner>	</FunctionDefinitionsBanner>
“Custom Source Code” on page 42-9	<CustomSourceCodeBanner>	</CustomSourceCodeBanner>
“Custom Header Code” on page 42-9	<CustomHeaderCodeBanner>	</CustomHeaderCodeBanner>

You can customize your banners by including tokens and comments between the open and close tags for each section. Tokens are replaced with values in the generated code. The following rules apply to tokens in your CGT file:

- You can have only one token per line.
- Token values must not contain a ‘\t’ for formatting.

Note: In the contents of your banner, C comment indicators, ‘/*’ or ‘*/’, can introduce an error in the generated code.

An open tag includes tag attributes. Enclose the value of the attribute in double quotes. The attributes available for an open tag are:

- **width:** specifies the width of the file or function banner comments in the generated code. The default value is 80.
- **style:** specifies the boundary for the file or function banner comments in the generated code.

The open tag syntax is:

```
<OpenTag style = "style_value" width = "num_width">
```

There are five options for the banner style. The `CommentStyle` and `TargetLang` configuration object properties determine the use of C or C++ comment style. The built-in style options for the `style` attribute are:

- classic

Using C style comments

```
/* single line comments */

/*
 * multiple line comments
 * second line
 */
```

Using C++ style comments

```
// single line comments

//
// multiple line comments
// second line
//
```

- box

Using C style comments

```
*****
/* banner contents */
*****
```

Using C++ style comments

```
////////////////////////////////////
// banner contents //
////////////////////////////////////
```

- open_box

Using C style comments

```
*****
 * banner contents
*****
```

Using C++ style comments

```
////////////////////////////////////
// banner contents
```

```
////////////////////////////////////////////////////////////////
```

- **doxygen**

Using C style comments

```
/** single line comments */
```

```
/**  
 * multiple line comments  
 * second line  
 */
```

Using C++ style comments

```
///single line comments
```

```
///  
/// multiple line comments  
///second line  
///
```

- **doxygen_qt**

Using C style comments

```
/*! single line comments */
```

```
/*!  
 * multiple line comments  
 * second line  
 */
```

Using C++ style comments

```
/*!single line comments
```

```
/*!  
/*! multiple line comments  
/*!second line  
/*!
```

File Banner

This section contains comments and tokens for use in generating a custom file banner that precedes the generated C and C++ code. If you omit the file banner section from the

CGT file, the code generation software does not generate a file banner in the generated code. The file banner section provided in the default CGT file is:

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Custom File Banner section (optional)
%% Customize File banners by using either custom tokens or the following
%% predefined tokens:
%% %<FileName>, %<MATLABCoderVersion>, %<EmbeddedCoderVersion>
%% %<SourceGeneratedOn>, %<HardwareSelection>, %<OutputType>
%%
%% You can also use "custom tokens" in all of the sections below. See the
%% documentation center for more details.
%%
<FileBanner style="classic">
File: %<FileName>

MATLAB Coder version      : %<MATLABCoderVersion>
C/C++ source code generated on : %<SourceGeneratedOn>
</FileBanner>

```

Summary of Tokens for File Banner Generation

FileName	Name of the generated file (for example, "kalman.c")
SourceGeneratedOn	Time stamp of generated file
MATLABCoderVersion	Version of MATLAB Coder
EmbeddedCoderVersion	Version of Embedded Coder
HardwareSelection	Selected target
OutputType	Type of output (for example, lib, exe, or dll)

Function Banner

This section contains comments and tokens for use in generating a custom function banner that precedes a generated C or C++ function. If you omit the function banner section from the CGT file, the code generation software does not generate function banners. The function banner section provided in the default CGT file is:

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Custom function banner section (optional)
%% Customize function banners by using the following predefined tokens:
%% %<FunctionName>, %<FunctionDescription>
%% %<Arguments>, %<ReturnType>
%%
<FunctionBanner style="classic">
%<FunctionDescription>
Arguments      : %<Arguments>
Return Type   : %<ReturnType>
</FunctionBanner>

```

Summary of Tokens for Function Banner Generation

FunctionName	Name of function
FunctionDescription	Short abstract about the function
Arguments	List of function arguments
ReturnType	Return type of function

Shared Utility Banner

This section contains comments and tokens for use in generating a custom shared utility function banner that precedes a generated C or C++ shared utility function. If you omit the shared utility function banner section from the CGT file, the code generation software does not generate shared utility function banners. The shared utility function banner section provided in the default CGT file is:

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Custom Shared Utility Function Banner section (optional)
%% Customize shared utility function banners by using the following
%% predefined tokens:
%% %<FunctionName>, %<FunctionDescription>
%% %<Arguments>, %<ReturnType>
%%
<SharedUtilityBanner style="classic">
Arguments      : %<Arguments>
Return Type    : %<ReturnType>
</SharedUtilityBanner>
    
```

Summary of Tokens for Shared Utility Function Banner Generation

FunctionName	Name of function
FunctionDescription	Short abstract about the function
Arguments	List of function arguments
ReturnType	Return type of function

File Trailer

The file trailer section contains comments for generating a custom file trailer that follows the generated C or C++ code. If you omit the file trailer section from the CGT file, the code generation software does not generate a file trailer. The file trailer section provided in the default CGT file is:

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Custom file trailer section (optional)
%% You can use any of the predefined tokens used for File Banner
    
```

```
%%
<FileTrailer style="classic">
File trailer for %<FileName>

[EOF]
</FileTrailer>
```

Tokens for the file banner are available for the file trailer. See Summary of Tokens for File Banner Generation.

Include Files Banner

The include files banner section contains comments for generating a custom banner that precedes the include files section in the generated code. If you omit the include files banner section from the CGT file, the code generation software does not generate a banner for this section. The include files banner section provided in the default CGT file is:

```
<IncludeFilesBanner style="classic">
Include Files
</IncludeFilesBanner>
```

Type Definitions Banner

The type definitions banner section contains comments for generating a custom banner that precedes the type definitions section in the generated code. If you omit the type definitions banner section from the CGT file, the code generation software does not generate a banner for this section. The type definitions banner section provided in the default CGT file is:

```
<TypeDefinitionsBanner style="classic">
Type Definitions
</TypeDefinitionsBanner>
```

Named Constants Banner

The named constants banner section contains comments for generating a custom banner that precedes the named constants section in the generated code. If you omit the named constants banner section from the CGT file, the code generation software does not generate a banner for this section. The named constants banner section provided in the default CGT file is:

```
<NamedConstantsBanner style="classic">
Named Constants
</NamedConstantsBanner>
```

Variable Declarations

The variable declarations banner section contains comments for generating a custom banner that precedes the variable declarations section in the generated code. If you omit the variable declarations banner section from the CGT file, the code generation software does not generate a banner for this section. The variable declarations banner section provided in the default CGT file is:

```
<VariableDeclarationsBanner style="classic">  
Variable Declarations  
</VariableDeclarationsBanner>
```

Variable Definitions

The variable definitions banner section contains comments for generating a custom banner that precedes the variable definitions section in the generated code. If you omit the variable definitions banner section from the CGT file, the code generation software does not generate a banner for this section. The variable definitions banner section provided in the default CGT file is:

```
<VariableDefinitionsBanner style="classic">  
Variable Definitions  
</VariableDefinitionsBanner>
```

Function Declarations

The function declarations banner section contains comments for generating a custom banner that precedes the function declarations section in the generated code. If you omit the function declarations banner section from the CGT file, the code generation software does not generate a banner for this section. The function declarations banner section provided in the default CGT file is:

```
<functionDeclarationsBanner style="classic">  
Function Declarations  
</FunctionDeclarationsBanner>
```

Function Definitions

The function definitions banner section contains comments for generating a custom banner that precedes the function definitions section in the generated code. If you omit the function definitions banner section from the CGT file, the code generation software does not generate a banner for this section. The function definitions banner section provided in the default CGT file is:

```
<FunctionDefinitionsBanner style="classic">  
Function Definitions  
</FunctionDefinitionsBanner>
```

Custom Source Code

The custom source code banner section contains comments for generating a custom banner that precedes the custom source code section in the generated code. If you omit the custom source code banner section from the CGT file, the code generation software does not generate a banner for this section. The custom source code banner section provided in the default CGT file is:

```
<CustomSourceCodeBanner style="classic">  
Custom Source Code  
</CustomSourceCodeBanner>
```

Customer Header Code

The custom header code banner section contains comments for generating a custom banner that precedes the custom header code section in the generated code. If you omit the custom header code banner section from the CGT file, the code generation software does not generate a banner for this section. The custom header code banner section provided in the default CGT file is:

```
<CustomHeaderCodeBanner style="classic">  
Custom Header Code  
</CustomHeaderCodeBanner>
```

Customize Generated Identifiers

In this section...

“Customize Identifiers Using the Project Interface” on page 42-18

“Customize Generated Identifiers Using the Command Line Interface” on page 42-19

If you have an Embedded Coder license, you can customize the identifiers that the MATLAB Coder software generates in the C/C++ code. To customize generated identifiers, specify the identifier format parameters in the project or the embedded code configuration object. For each parameter, enter a macro string. The code generation software expands the macro string and includes it in the generated identifiers.

The macro string can include:

- Valid C or C++ language identifiers (a-z, A-Z, _, 0–9).
- The tokens listed in the following table. \$M is required.

Token	Description
\$M	Insert name mangling string to avoid naming collisions. Required.
\$N	Insert name of the object (global variable, global type, local function, local temporary variable, or constant macro) for which the identifier is generated. Improves readability of generated code.
\$R	Insert root project name into identifier, replacing unsupported characters with the underscore (_) character.

Customize Identifiers Using the Project Interface

In a project, on the **Code Appearance** tab, enter macros for the parameters that you want to customize.

Project Parameter	Default Macro
Global variables	\$M\$N

Project Parameter	Default Macro
Global types	\$M\$N
Field name of global types	\$M\$N
Local functions	\$M\$N
Local temporary variables	\$M\$N
Constant macros	\$M\$N
EMX Array Types	emxArray_ \$M\$N
EMX Array Utility Functions	emx\$M\$N

For example, suppose **Global variables** has the value `glob_ MN`. For a global variable named `g`, when name mangling is not required, the generated identifier is `glob_g`. If name mangling is required, the generated identifier includes the name mangling string.

Customize Generated Identifiers Using the Command Line Interface

At the command line, create a code configuration object. Define macros for the parameters that you want to customize.

Embedded Code Configuration Object Parameter	Description	Default Macro
CustomSymbolStrGlobalVar	Global variables	' \$M\$N '
CustomSymbolStrType	Global types	' \$M\$N '
CustomSymbolStrField	Field name of global types	' \$M\$N '
CustomSymbolStrFcn	Local functions	' \$M\$N '
CustomSymbolStrTmpVar	Local temporary variables	' \$M\$N '
CustomSymbolStrMacro	Constant macros	' \$M\$N '
CustomSymbolStrEMXArray	EMX Array Types	' emxArray_ \$M\$N '
CustomSymbolStrEMXArrayFcn	EMX Array Utility Functions	' emx\$M\$N '

For example:

```
cfg = coder.config('lib','ecoder',true);
cfg.CustomSymbolStrGlobalVar = 'glob_ $M$N';
```

For a global variable named `g`, when name mangling is not required, the generated identifier is `glob_g`. If name mangling is required, the generated identifier includes the name mangling string.

Code Replacement for MATLAB Code

- “What Is Code Replacement?” on page 43-2
- “Code You Can Replace from MATLAB Code” on page 43-4
- “Code Replacement Libraries” on page 43-15
- “Code Replacement Terminology” on page 43-17
- “Code Replacement Limitations” on page 43-20
- “Replace Code Generated from MATLAB Code” on page 43-21
- “Choose a Code Replacement Library” on page 43-24

What Is Code Replacement?

Code replacement is a technique to change the code that the code generator produces for functions and operators to meet application code requirements. For example, you can replace generated code to meet requirements such as:

- Optimization for a specific run-time environment, including, but not limited to, specific target hardware.
- Integration with existing application code.
- Compliance with a standard, such as AUTOSAR.
- Modification of code behavior, such as enabling or disabling nonfinite or inline support.
- Application- or project-specific code requirements, such as:
 - Elimination of `math.h`.
 - Elimination of system header files.
 - Elimination of calls to `memcpy` or `memset`.
 - Use of BLAS.
 - Use of a specific BLAS.

To apply this technique, configure the code generator to apply a code replacement library (CRL) during code generation. By default, the code generator does not apply a code replacement library. You can choose from the following libraries that MathWorks provides:

- GNU C99 extensions—GNU⁶ gcc math library, which provides C99 extensions as defined by compiler option `-std=gnu99`.
- Intel IPP for x86-64 (Windows)—Generates calls to the Intel Performance Primitives (IPP) library for the x86-64 Windows platform.
- Intel IPP/SSE with GNU99 extensions for x86-64 (Windows)—GNU libraries for Intel Performance Primitives (IPP) and Streaming SIMD Extensions (SSE), with GNU C99 extensions.
- Intel IPP for x86/Pentium (Windows)—Generates calls to the Intel Performance Primitives (IPP) library for the x86/Pentium Windows platform.

6. GNU is a registered trademark of the Free Software Foundation.

- Intel IPP/SSE with GNU99 extensions for x86/Pentium (Windows)—Generates calls to the GNU libraries for Intel Performance Primitives (IPP) and Streaming SIMD Extensions (SSE), with GNU C99 extensions, for the x86/Pentium Windows platform.
- Intel IPP for x86-64 (Linux)—Generates calls to the Intel Performance Primitives (IPP) library for the x86-64 Linux platform.
- Intel IPP/SSE with GNU99 extensions for x86-64 (Linux)—Generates calls to the GNU libraries for Intel Performance Primitives (IPP) and Streaming SIMD Extensions (SSE), with GNU C99 extensions, for the x86-64 Linux platform.

Libraries that include GNU99 extensions are intended for use with the GCC compiler. If use one of those libraries with another compiler, generated code might not compile.

Depending on the product licenses that you have, other libraries might be available. If you have an Embedded Coder license, you can view and choose from other libraries and you can create custom code replacement libraries.

Code replacement is a technique to change the code that the code generator produces for functions and operators to meet application code requirements. For example, you can replace generated code to meet requirements such as:

- Optimization for a specific run-time environment, including, but not limited to, specific target hardware.
- Integration with existing application code.
- Compliance with a standard, such as AUTOSAR.
- Modification of code behavior, such as enabling or disabling nonfinite or inline support.
- Application- or project-specific code requirements, such as:
 - Elimination of `math.h`.
 - Elimination of system header files.
 - Elimination of calls to `memcpy` or `memset`.
 - Use of BLAS.
 - Use of a specific BLAS.

To apply this technique, configure the code generator to apply a code replacement library (CRL) during code generation. By default, the code generator does not apply a code replacement library. You can choose from the following libraries that MathWorks provides:

- GNU C99 extensions—GNU⁷ gcc math library, which provides C99 extensions as defined by compiler option `-std=gnu99`.
- Intel IPP for x86-64 (Windows)—Generates calls to the Intel Performance Primitives (IPP) library for the x86-64 Windows platform.
- Intel IPP/SSE with GNU99 extensions for x86-64 (Windows)—GNU libraries for Intel Performance Primitives (IPP) and Streaming SIMD Extensions (SSE), with GNU C99 extensions.
- Intel IPP for x86/Pentium (Windows)—Generates calls to the Intel Performance Primitives (IPP) library for the x86/Pentium Windows platform.
- Intel IPP/SSE with GNU99 extensions for x86/Pentium (Windows)—Generates calls to the GNU libraries for Intel Performance Primitives (IPP) and Streaming SIMD Extensions (SSE), with GNU C99 extensions, for the x86/Pentium Windows platform.
- Intel IPP for x86-64 (Linux)—Generates calls to the Intel Performance Primitives (IPP) library for the x86-64 Linux platform.
- Intel IPP/SSE with GNU99 extensions for x86-64 (Linux)—Generates calls to the GNU libraries for Intel Performance Primitives (IPP) and Streaming SIMD Extensions (SSE), with GNU C99 extensions, for the x86-64 Linux platform.

Libraries that include GNU99 extensions are intended for use with the GCC compiler. If use one of those libraries with another compiler, generated code might not compile.

Depending on the product licenses that you have, other libraries might be available . If you have an Embedded Coder license, you can view and choose from other libraries and you can create custom code replacement libraries.

Related Examples

- “Replace Code Generated from MATLAB Code”
- “Choose a Code Replacement Library”

More About

- “Code You Can Replace from MATLAB Code”
- “Code Replacement Libraries”
- “Code Replacement Terminology”
- “Code Replacement Limitations”

7. GNU is a registered trademark of the Free Software Foundation.

Code You Can Replace from MATLAB Code

In this section...

“About Code You Can Replace” on page 43-4

“Math Functions” on page 43-4

“Memory Functions” on page 43-9

“Operators” on page 43-9

About Code You Can Replace

Code that the code generator replaces depends on the code replacement library (CRL) that you use. By default, the code generator does not apply a code replacement library. Your choice of libraries is dependent on product licensing and whether you have access to custom libraries.

For information on how to explore functions and operators that a code replacement library supports, see “Choose a Code Replacement Library”. If you have an Embedded Coder license and want to develop a custom code replacement library, see Code Replacement Customization.

Math Functions

When generating C/C++ code from MATLAB code, depending on code replacement libraries available in your development environment, you can configure the code generator to replace instances of the following math functions with application-specific implementations.

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
abs ¹	Floating point	Scalar	Real
acos	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
acosd	Floating point	Scalar Vector Matrix	Real Complex

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
acot	Floating point	Scalar Vector Matrix	Real Complex
acotd	Floating point	Scalar Vector Matrix	Real Complex
acoth	Floating point	Scalar Vector Matrix	Real Complex
acsc	Floating point	Scalar Vector Matrix	Real Complex
acscd	Floating point	Scalar Vector Matrix	Real Complex
acsch	Floating point	Scalar Vector Matrix	Real Complex
asec	Floating point	Scalar Vector Matrix	Real Complex
asecd	Floating point	Scalar Vector Matrix	Real Complex
asech	Floating point	Scalar Vector Matrix	Real Complex
asin	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
asind	Floating point	Scalar Vector Matrix	Real Complex
atan	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
atan2	Floating point	Scalar Vector Matrix	Real
atan2d	Floating point	Scalar Vector Matrix	Real
atand	Floating point	Scalar Vector Matrix	Real Complex
cos	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
ceil	<ul style="list-style-type: none"> • Floating-point • Scalar 	<ul style="list-style-type: none"> • Floating-point • Scalar 	<ul style="list-style-type: none"> • Floating-point • Scalar
cosd	Floating point	Scalar Vector Matrix	Real Complex
cosh	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
cot	Floating point	Scalar Vector Matrix	Real Complex

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
cotd	Floating point	Scalar Vector Matrix	Real Complex
coth	Floating point	Scalar Vector Matrix	Real Complex
csc	Floating point	Scalar Vector Matrix	Real Complex
cscd	Floating point	Scalar Vector Matrix	Real Complex
csch	Floating point	Scalar Vector Matrix	Real Complex
exp	Floating point	Scalar	Real
fix	Floating point	Scalar	Real
floor	<ul style="list-style-type: none"> • Floating-point • Scalar 	<ul style="list-style-type: none"> • Floating-point • Scalar 	<ul style="list-style-type: none"> • Floating-point • Scalar
hypot	Floating point	Scalar Vector Matrix	Real
ldexp	Floating point	Scalar	Real
log	Floating point	Scalar Vector Matrix	Real Complex
log10	Floating point	Scalar Vector Matrix	Real Complex
log2	Floating point	Scalar Vector Matrix	Real Complex

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
max	Integer Floating point	Scalar	Real
min	Integer Floating point	Scalar	Real
pow	Floating point	Scalar	Real
rem	Floating point	Scalar	Real
round	Floating point	Scalar	Real
sec	Floating point	Scalar Vector Matrix	Real Complex
secd	Floating point	Scalar Vector Matrix	Real Complex
sech	Floating point	Scalar Vector Matrix	Real Complex
sign	Floating point	Scalar	Real
sin	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
sind	Floating point	Scalar Vector Matrix	Real Complex
sinh	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
sqrt	Floating point	Scalar	Real

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
tan	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
tand	Floating point	Scalar Vector Matrix	Real Complex
tanh	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output

¹ Wrap on integer overflow only

Memory Functions

Depending on code replacement libraries available in your development environment, you can configure the code generator to replace instances of the following memory functions with application-specific implementations.

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
memcmp	Void pointer (void*)	Scalar Vector Matrix	Real Complex
memcpy	Void pointer (void*)	Scalar Vector Matrix	Real Complex
memset	Void pointer (void*)	Scalar Vector Matrix	Real Complex
memset2zero	Void pointer (void*)	Scalar Vector Matrix	Real Complex

Some target processors provide optimized functions to set memory to zero. Use the code replacement library programming interface to replace the `memset2zero` function with more efficient target-specific functions.

Operators

When generating C/C++ code from MATLAB code, depending on code replacement libraries available in your development environment, you can configure the code generator to replace instances of the following operators with application-specific implementations.

Mixed data type support indicates you can specify different data types of different inputs.

Operator	Key	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
Addition (+)	RTW_OP_ADD	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Subtraction (-)	RTW_OP_MINUS	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Multiplication (*) ¹	RTW_OP_MUL	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Division (/)	RTW_OP_DIV	Integer Floating point Fixed-point Mixed	Scalar	Real Complex
Data type conversion (cast)	RTW_OP_CAST	Integer Floating point ² Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Shift left (<<)	RTW_OP_SL	Integer	Scalar	Real

Operator	Key	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
		Fixed-point Mixed	Vector Matrix	
Shift right arithmetic (>>) ³	RTW_OP_SRA	Integer Fixed-point Mixed	Scalar Vector Matrix	Real
Shift right logical (>>)	RTW_OP_SRL	Integer Fixed-point Mixed	Scalar Vector Matrix	Real
Element-wise matrix multiplication (.*) ⁴	RTW_OP_ELEM_MUL	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Complex conjugation	RTW_OP_CONJUGATE	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Transposition (.')	RTW_OP_TRANS	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Hermitian (complex conjugate) transposition (')	RTW_OP_HERMITIAN	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Multiplication with transposition ¹	RTW_OP_TRMUL	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Multiplication with Hermitian transposition ¹	RTW_OP_HMMUL	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex

Operator	Key	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
Greater than (>)	RTW_OP_GREATER_THAN	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Greater than or equal(>=)	RTW_OP_GREATER_THAN_OR_EQUAL	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Less than (<)	RTW_OP_LESS_THAN	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Less than or equal (<=)	RTW_OP_LESS_THAN_OR_EQUAL	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Equal (==)	RTW_OP_EQUAL	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Not equal (!=)	RTW_OP_NOT_EQUAL	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
<p>¹ Can map to Basic Linear Algebra Subroutine (BLAS) multiplication functions.</p> <p>² Scaled floating point is not supported.</p> <p>³ Code replacement libraries that provide arithmetic shift right implementations should also provide logical shift right implementations, because some arithmetic shift rights are converted to logical shift rights during code generation.</p> <p>⁴ Use the multiplication (*) operator (RTW_OP_MUL) for scalar multiplication.</p>				

Related Examples

- “Choose a Code Replacement Library”

More About

- “Code Replacement Libraries”
- “Code Replacement Terminology”
- “Code Replacement Limitations”

Code Replacement Libraries

A *code replacement library* consists of one or more code replacement tables that specify application-specific implementations of functions and operators. For example, a library for a specific embedded processor specifies function and operator replacements that optimize generated code for that processor.

A *code replacement table* contains one or more *code replacement entries*, with each entry 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.

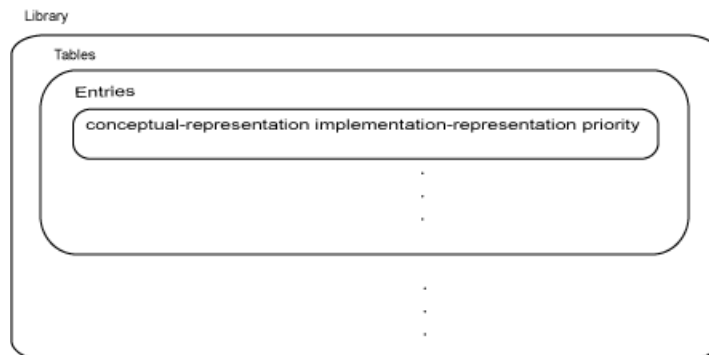


Table Entry Component	Description
Conceptual representation	<p>Identifies the table entry and contains match criteria for the code generator. Consists of:</p> <ul style="list-style-type: none"> • Function name or a key. The function name identifies most functions. For operators and some functions, a string called a key identifies a function or operator. For example, function name 'COS' and operator key 'RTW_OP_ADD'. • Conceptual arguments that observe code generator naming ('y1', 'u1', 'u2', ...), with corresponding I/O types (output or input) and data types. • Other attributes, such as an algorithm, fixed-point saturation, and rounding modes, which identify matching criteria for the function or operator.

Table Entry Component	Description
Implementation representation	Specifies replacement code. Consists of: <ul style="list-style-type: none"> • Function name. For example, 'cos_dbl' or 'u8_add_u8_u8') • Implementation arguments, with corresponding I/O types (output or input) and data types. • Parameters that provide additional implementation details, such as header and source file names and paths of build resources.
Priority	Defines the entry priority relative to other entries in the table. The value can range from 0 to 100, with 0 being the highest priority. If multiple entries have the same priority, the code generator uses the first match with that priority.

When the code generator looks for a match in a code replacement library, it creates and populates a *call site object* with the function or operator conceptual representation. If a match exists, the code generator uses the matched code replacement entry populated with the implementation representation and uses it to generate code.

The code generator searches the tables in a code replacement library for a match in the order that the tables appear in the library. If the code generator finds multiple matches within a table, the priority determines the match. The code generator uses a higher-priority entry over a similar entry with a lower priority.

Related Examples

- “What Is Code Replacement?”
- “Replace Code Generated from MATLAB Code”
- “Choose a Code Replacement Library”

More About

- “Code You Can Replace from MATLAB Code”
- “Code Replacement Terminology”

Code Replacement Terminology

Term	Definition
Cache hit	A code replacement entry for a function or operator, defined in the specified code replacement library, for which the code generator finds a match.
Cache miss	A conceptual representation of a function or operator for which the code generator does not find a match.
Call site object	Conceptual representation of a function or operator that the code generator uses when it encounters a call site for a function or operator. The code generator uses the object to query the code replacement library for a conceptual representation match. If a match exists, the code generator returns a code replacement object, fully populated with the conceptual representation, implementation representation, and priority, and uses that object to generate replacement code.
Code replacement library	One or more code replacement tables that specify application-specific implementations of functions and operators. When configured to use a code replacement library, the code generator uses criteria defined in the library to search for matches. If a match is found, the code generator replaces code that it generates by default with application-specific code defined in the library.
Code replacement table	One or more code replacement table entries. Provides a way to group related or shared entries for use in different libraries.
Code replacement entry	Represents a potential replacement for a function or operator. Maps a conceptual representation of a function or operator to an implementation representation and priority.
Conceptual argument	Represents an input or output argument for a function or operator being replaced. Conceptual arguments observe naming conventions ('y1',

Term	Definition
	'u1', 'u2', ...) and data types familiar to the code generator.
Conceptual representation	<p>Represents match criteria that the code generator uses to qualify functions and operators for replacement. Consists of:</p> <ul style="list-style-type: none"> • Function or operator name or key • Conceptual arguments with type, dimension, and complexity specification for inputs and output • Attributes, such as an algorithm and fixed-point saturation and rounding modes
Implementation argument	Represents an input or output argument for a C or C++ replacement function. Implementation arguments observe C/C++ name and data type specifications.
Implementation representation	<p>Specifies C or C++ replacement function prototype. Consists of:</p> <ul style="list-style-type: none"> • Function name (for example, 'cos_db1' or 'u8_add_u8_u8') • Implementation arguments specifying type, type qualifiers, and complexity for the function inputs and output • Parameters that provide build information, such as header and source file names and paths of build resources and compile and link flags
Key	A string that identifies a function or operator that is being replaced. A function name or key appears in the conceptual representation of a code replacement entry. The key RTW_OP_ADD identifies the addition operator.

Term	Definition
Priority	Defines the match priority for a code replacement entry relative to other entries, which have the same name and conceptual argument list, within a code replacement library. The priority can range from 0 to 100, with 0 being the highest priority. The default is 100. If a library provides two implementations for a function or operator, the implementation with the higher priority shadows the one with the lower priority.

More About

- “What Is Code Replacement?”
- “Code Replacement Libraries”

Code Replacement Limitations

Code replacement verification — It is possible that code replacement behaves differently than you expect. For example, data types that you observe in code generator input might not match what the code generator uses as intermediate data types during an operation. Verify code replacements by examining generated code.

Related Examples

- “Replace Code Generated from MATLAB Code”

More About

- “Code You Can Replace from MATLAB Code”
- “Code Replacement Libraries”

Replace Code Generated from MATLAB Code

This example shows how to replace generated code, using a code replacement library. Code replacement is a technique you can use to change the code that the code generator produces for functions and operators to meet application code requirements.

Prepare for Code Replacement

- 1 Make sure required software is installed. You need MATLAB, MATLAB Coder, and a C compiler. Some code replacement libraries available in your development environment might also require Embedded Coder.

For instructions on installing MathWorks products, see the MATLAB installation documentation. If you have installed MATLAB and want to check which other MathWorks products are installed, in the MATLAB Command Window, enter `ver`.

- 2 Identify an existing or create a new MATLAB function for which you want the code generator to replace code.

Choose a Code Replacement Library

If you are not sure which library to use, explore available libraries.

Configure Code Generator To Use Code Replacement Library

- 1 Configure the code generator to apply a code replacement library during code generation for the MATLAB function.
 - In a project, on the **Hardware** tab, set the **Code Replacement Library** parameter.
 - In a code configuration object, set the `CodeReplacementLibrary` parameter.
- 2 Configure the code generator to produce code only so you can verify your code replacements before building an executable.
 - In a project, on the **Build** tab, set the **Generate code only** parameter.
 - In a code configuration object, set the `GenCodeOnly` parameter.

Include Code Replacement Information In Code Generation Report

If you have an Embedded Coder license, you can configure the code generator to include a code replacement section in the code generation report. The additional information can help you verify code replacements.

- 1 Configure the code generator to generate a report.
 - In a project, on the **Debugging** tab, set the **Always create a code generation report** parameter.
 - In a code configuration object, set the `GenerateReport` parameter.
- 2 Include the code replacement section in the report.
 - In a project, on the **Debugging** tab, set the **Code replacements** parameter.
 - In a code configuration object, set the `GenerateCodeReplacementReport` parameter.

Generate Replacement Code

Generate C/C++ code from the MATLAB code and, if you configured the code generator accordingly, a code generation report. For example, on the **Build** tab, click **Build**. Or, at the command prompt, enter:

```
codegen -report myFunction -args {5} -config cfg
```

The code generator produces the code and displays the report.

Verify Code Replacements

Verify code replacements by examining the generated code. Code replacement might behave differently than you expect, for example, data types that you observe in the code generator input might not match what the code generator uses as intermediate data types during an operation.

Related Examples

- “Choose a Code Replacement Library”
- “Build Setting Configuration”
- “Review and Test Code Replacements”

More About

- “What Is Code Replacement?”
- “Code You Can Replace from MATLAB Code”
- “Code Replacement Libraries”
- “Code Replacement Terminology”

- “Code Replacement Limitations”

External Web Sites

- Supported Compilers

Choose a Code Replacement Library

In this section...

“About Choosing a Code Replacement Library” on page 43-24

“Explore Available Code Replacement Libraries” on page 43-24

“Explore Code Replacement Library Contents” on page 43-32

About Choosing a Code Replacement Library

By default, the code generator does not use a code replacement library.

If you are considering using a code replacement library:

- 1 Explore available libraries. Identify one that best meets your application needs.
 - Consider the lists of application code replacement requirements and libraries that MathWorks provides in “What Is Code Replacement?”.
 - See “Explore Available Code Replacement Libraries”.
- 2 Explore the contents of the library. See “Explore Code Replacement Library Contents”.

If you do not find a suitable library and you have an Embedded Coder license, you can create a custom code replacement library.

Explore Available Code Replacement Libraries

You can select the code replacement library to use for code generation in a project, on the **Hardware** tab, by setting the **Code Replacement Library** parameter. Alternatively, in a code configuration object, set the `CodeReplacementLibrary` parameter.

Explore Code Replacement Library Contents

Use the Code Replacement Viewer to explore the content of a code replacement library.

- 1 At the command prompt, type `RTW.viewTf1`.

```
>> RTW.viewTf1
```

The viewer opens. To view the content of a specific library, specify the name of the library as an argument in single quotes. For example:

```
>> RTW.viewTf1('GNU C99 extensions')
```

- 2 In the left pane, select the name of a library. The viewer displays information about the library in the right pane.
- 3 In the left pane, expand the library, explore the list of tables it contains, and select a table from the list. In the middle pane, the viewer displays the function and operator entries that are in that table, along with abbreviated information for each entry.
- 4 In the middle pane, select a function or operator. The viewer displays information from the table entry in the right pane.

If you select an operator entry that specifies net slope fixed-point parameters (instantiated from entry class `RTW.Tf1COperationEntryGenerator` or `RTW.Tf1COperationEntryGenerator_NetSlope`), the viewer displays an additional tab that shows fixed-point settings.

See [Code Replacement Viewer](#) for details on what the viewer displays.

Related Examples

- [“Replace Code Generated from MATLAB Code”](#)

More About

- [“What Is Code Replacement?”](#)
- [“Code You Can Replace from MATLAB Code”](#)
- [“Code Replacement Libraries”](#)
- [“Code Replacement Terminology”](#)
- [“Code Replacement Limitations”](#)

Verification of Code Generated from MATLAB Code

- “Highlight Potential Data Type Issues in a Report” on page 44-2
- “Find Potential Data Type Issues in Generated Code” on page 44-5

Highlight Potential Data Type Issues in a Report

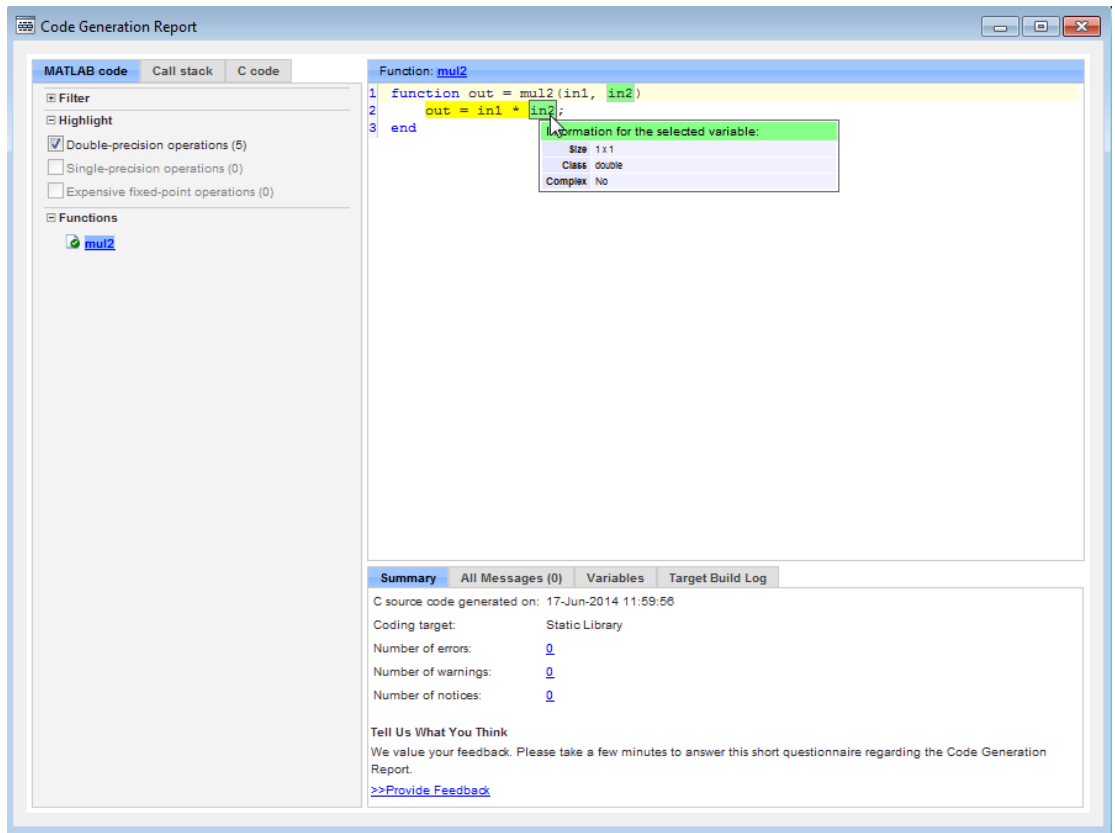
In this section...
“Enable Highlight Option Using a Project” on page 44-3
“Enable Highlight Option Using the Command Line Interface” on page 44-4

If you have an Embedded Coder license, you have the option to highlight potential data types issues in the code generation report for standalone code generated from MATLAB code. If you enable this option, the **Highlight** section on the **MATLAB Code** tab lists the number of single-precision and double-precision operations in the generated C/C++ code. If you have a Fixed-Point Designer license, it also lists the number of expensive fixed-point operations.

To highlight the MATLAB code that corresponds to the potential data type issues:

- 1 Select the check box for the type of operation that you want to highlight.
- 2 Select the function that you want to highlight.

The report highlights the operations in the selected function. The following example report highlights MATLAB code that results in double-precision operations in the generated code.



The option to highlight potential data type issues is disabled by default.

Enable Highlight Option Using a Project

- 1 On the **Build** tab, set the **Output Type** to one of the following:
 - C/C++ Static Library
 - C/C++ Dynamic Library
 - C/C++ Executable
- 2 Click the **More settings** link to open the **Project Settings** dialog box.

- 3 On the **Debugging** tab, select the **Always create a code generation report** and **Highlight potential data type issues** check boxes.

Enable Highlight Option Using the Command Line Interface

- 1 Create an embedded code configuration object for 'lib', 'dll', or 'exe':

```
cfg = coder.config('lib','ecoder',true); % or dll or exe
```

- 2 Set the **GenerateReport** and **HighlightPotentialDataTypeIssues** configuration object properties to **true**:

```
cfg.GenerateReport = true;  
cfg.HighlightPotentialDataTypeIssues = true;
```

Related Examples

- “Find Potential Data Type Issues in Generated Code”

Find Potential Data Type Issues in Generated Code

In this section...

“Data Type Issues Overview” on page 44-5

“Enable Highlighting of Potential Data Type Issues” on page 44-5

“Find and Address Cumbersome Operations” on page 44-6

“Find and Address Expensive Rounding” on page 44-8

“Find and Address Expensive Comparison Operations” on page 44-9

Data Type Issues Overview

When you generate C code from MATLAB code, you can highlight potential data type issues in the C code generation report. The report highlights MATLAB code that requires single-precision, double-precision, or expensive fixed-point operations. The expensive fixed-point operations checks require a Fixed-Point Designer license.

- The double-precision check highlights expressions that result in a double-precision operation. Manual inspection of code to find unwanted doubles can be time-consuming and error prone.
- The single-precision check highlights expressions that result in a single operation.
- The expensive fixed-point operations check identifies optimization opportunities for fixed-point code. It highlights expressions in the MATLAB code that require cumbersome multiplication or division, or expensive rounding. For more information on optimizing generated fixed-point code, see “Tips for Making Generated Code More Efficient”.

Enable Highlighting of Potential Data Type Issues

Procedure 44.1. Enable the highlight option in a project

- 1 On the **Build** tab, set the **Output Type** to one of the following:
 - C/C++ Static Library
 - C/C++ Dynamic Library
 - C/C++ Executable
- 2 Click the **More settings** link to open the Project Settings dialog box.

- 3 On the **Debugging** tab, select the **Always create a code generation report** and **Highlight potential data type issues** check boxes.

Procedure 44.2. Enable the highlight option using the command line interface

- 1 Create an embedded code configuration object for 'lib', 'dll', or 'exe':

```
cfg = coder.config('lib','ecoder',true); % or dll or exe
```

- 2 Set the `GenerateReport` and `HighlightPotentialDataTypeIssues` configuration object properties to true:

```
cfg.GenerateReport = true;  
cfg.HighlightPotentialDataTypeIssues = true;
```

Find and Address Cumbersome Operations

Cumbersome operations usually occur due to an insufficient range of output. Avoid inputs to a multiply or divide operation that have word lengths larger than the base integer type of your processor. Software can process operations with larger word lengths, but this approach requires more code and runs slower.

This example requires Embedded Coder and Fixed-Point Designer licenses to run.

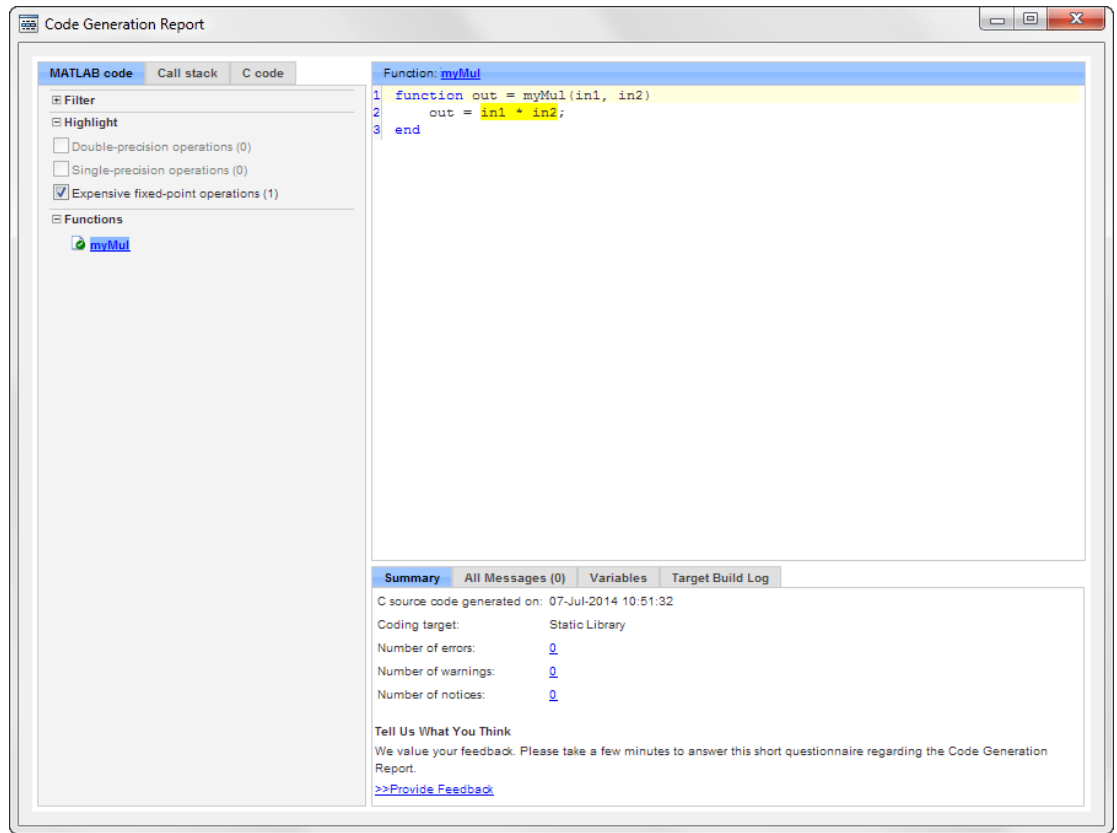
- 1 Create the function `myMul`.

```
function out = myMul(in1, in2)  
    out = in1 * in2;  
end
```

- 2 Generate code for `myMul`.

```
cfg = coder.config('lib');  
cfg.GenerateReport = true;  
cfg.HighlightPotentialDataTypeIssues = true;  
codegen -config cfg myMul -args {fi(1, 1, 33, 4), fi(1, 1, 32, 4)}
```

- 3 Click **View report**.
- 4 In the Code Generation Report, on the left pane, click the **MATLAB code** tab.
- 5 Expand the **Highlight** section and select the **Expensive fixed-point operations** check box.



The `in1 * in2` operation is highlighted in the HTML report. On the bottom pane, click the **Variables** tab. The word length of `in1` is 33, and the word length of `in2` is 32. Hovering over the highlighted expression reveals that the product has a word length of 65, which is larger than the target word length of 64. Therefore, the software detects a cumbersome operation.

To resolve this issue, modify the data types of `in1` and `in2` so the word length of the product does not exceed the target word length.

Find and Address Expensive Rounding

Traditional handwritten code, especially for control applications, almost always uses "no effort" rounding. For example, for unsigned integers and two's complement signed integers, shifting right and dropping the bits is equivalent to rounding to floor. To get results comparable to, or better than, what you expect from traditional handwritten code, use the `floor` rounding method.

This example requires Embedded Coder and Fixed-Point Designer licenses to run.

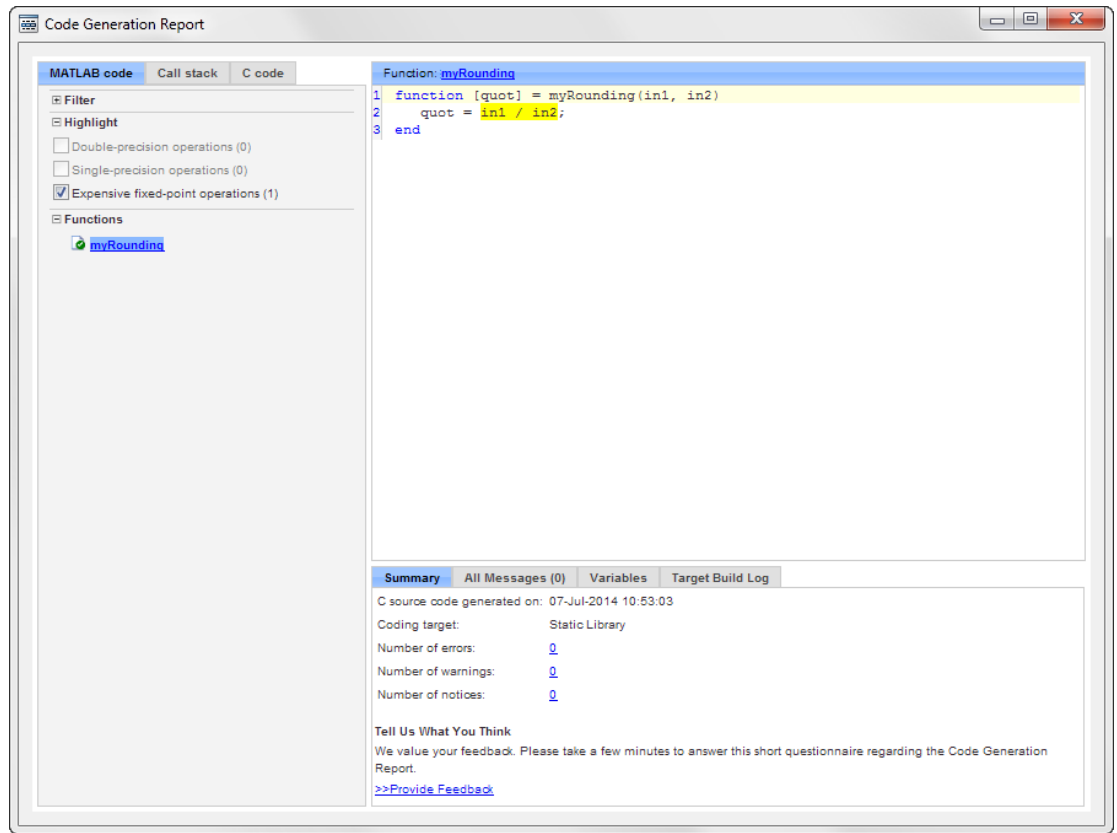
- 1 Create the function `myRounding`.

```
function [quot] = myRounding(in1, in2)
    quot = in1 / in2;
end
```

- 2 Generate code for `myRounding`.

```
cfg = coder.config('lib');
cfg.GenerateReport = true;
cfg.HighlightPotentialDataTypeIssues = true;
codegen -config cfg myRounding -args {fi(1, 1, 32, 2), fi(1, 1, 32, 4)}
```

- 3 Click **View report**.
- 4 In the Code Generation Report, on the left pane, click the **MATLAB code** tab.
- 5 Expand the **Highlight** section and select the **Expensive fixed-point operations** check box.



This division operation uses the default rounding method, `nearest`. Changing the rounding method to `Floor` provides a more efficient implementation.

Find and Address Expensive Comparison Operations

Comparison operations generate extra code when a casting operation is required to do the comparison. For example, before comparing an unsigned integer to a signed integer, one of the inputs must be cast to the signedness of the other. Consider optimizing the data types of the input arguments so that a cast is not required in the generated code.

This example requires Embedded Coder and Fixed-Point Designer licenses to run.

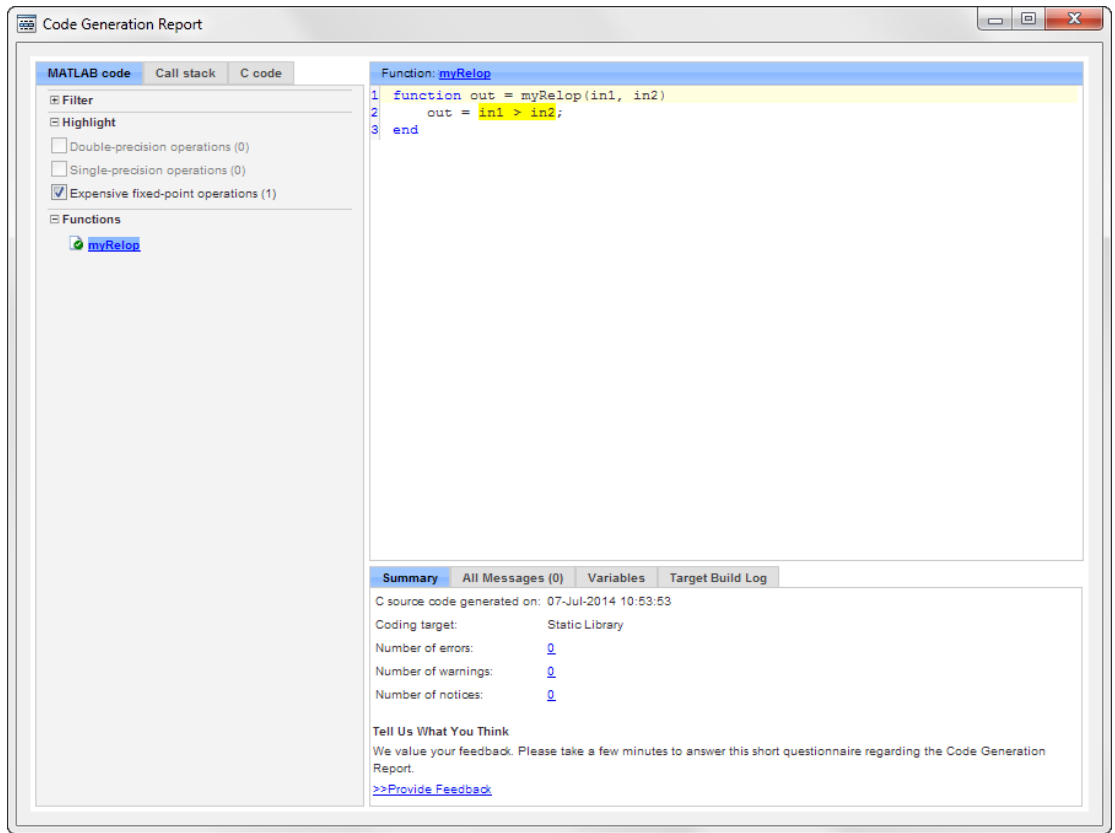
- 1 Create the function myRelop.

```
function out = myRelop(in1, in2)
    out = in1 > in2;
end
```

- 2 Generate code for myRelop.

```
cfg = coder.config('lib');
cfg.GenerateReport = true;
cfg.HighlightPotentialDataTypeIssues = true;
codegen -config cfg myRelop -args {fi(1, 1, 32, 1.5, 9, 17), fi(1, 0, 32, 16)}
```

- 3 Click **View report**.
- 4 In the Code Generation Report, on the left pane, click the **MATLAB code** tab.
- 5 Expand the **Highlight** section and select the **Expensive fixed-point operations** check box.



The first input argument, `in1`, is signed with slope bias scaling, while `in2` is unsigned with binary point scaling. Extra code is generated because a cast must occur before the two inputs can be compared.

Change the signedness and scaling of one of the inputs to generate more efficient code.

